3D Kadaster of the Netherlands

Arumoy Shome
Vrije Universiteit

Cees Portegies
Vrije Universiteit

Shruti Rao
Vrije Universiteit

1 Introduction

With growing population, human migration and urbanization, governments face an increasing need to manage urban resources such as traffic flows, energy consumption, storm water management and waste disposal. Many agencies especially in the Netherlands have begun dealing with some of these problems by using 3D models to recreate issues in a testing space and thus better understand them [1].

Modeling entire cities has gained a lot of popularity due to its ability to give governments detailed overviews of the existing cities and enable better functioning in terms of construction, geological survey and municipal administration. It is a step in the direction of the popular concept of *"smart cities"* [2]. Based on this motivation, we aim to build a 3D model of all the existing buildings in the Netherlands.

Netherlands is a small country with land shortage. It also faces an influx of human and corporate migration that puts further pressure on its land and urban resources. There is a great need for Netherlands to efficiently use all of its available land to accommodate for its growing cities and industries. Using a 3D model of the existing buildings, city planners can better visualize and plan new development in relation to the existing urban environment [3]. This allows for efficient use of the available land space. Additionally, the 3D city model can be used by monitoring agencies to detect and prevent illegal establishments [4].

Evaluate areas in 3D dimensional space, for example a city, point cloud data is used. This data maps a space in 3D dimensions through the means of many *points*. An example of a large scale point cloud is the AHN2 point cloud data set for the Netherlands. However, rendering such data is computationally expensive and thus hard to wowrk with[5]. To this end, point cloud data is adapted to mesh data, which is far less computationally expensive to render and can present an enclosed area, making it more suitable to represent geometry. However, this means that the point cloud data needs to be converted to such a mesh datastructure, which is an area of on going research. Various considerations regarding scale and level of detail have to be made when constructing the meshes. This research is focused on creating meshes on a large scale, for an entire country, to illustrate the feasibility and increased usability over point cloud data.

2 Related Work

3D model reconstruction from point cloud data is a well established field, with a few well established problems as well. The first main problem is that often the point cloud data used is not classified. It is not known whether the points describe a building, foliage or otherwise. This is an important point, since one might want to consider different reconstruction methods for the 3D geometry methods for foliage versus buildings. This classification problem was one of the key focus points of research by Lafarge and Mallet[5]. For our research however, the classification problem is somewhat solved by an additional dataset which describes land use, thus serving as an external classification tool.

The next important point in their research is related to the actual construction of a 3D model. Depending on the mesh construction algorithm, the constructed polygonal mesh can either be accurate or inaccurate. An accurate construction is well detailed and thus large in size. An inaccurate construction has a smaller footprint however, the trade off here is the lack of detail in the construction. As the focus of this project is on building reconstruction, trees are simply represented as ellipsoidal shapes to lower the number of vertices and faces needed to describe the foliage. Similarly, in research by Vosselman, Dijkman, et al. the focus was on determining how much point cloud data is required to reconstruct a geometry accurately. It was observed that the density of the point cloud data was directly proportional to the accuracy of the generated model.

A program has been constructed by the Technical University of Delft (TU Delft), to operate on similar datasets [7]. The focus of the software they created - the *3dfier*, was not simply on buildings but also to generate accurate terrain. In a related research by Kumar, Ledoux, and Stoter, it was

demonstrated that terrain can be the most challenging part of the reconstruction, where just for a single city it could be in excess of 700GB, but with the right algorithms this could be significantly reduced. The software's capabilities were demonstrated by using two datasets similar to the ones used in our research. First, they used the AHN3 dataset as that is the next generation of the AHN2 dataset and as such it is more accurate. The other dataset is the BGT dataset which describes land usage in terms of vegetation, buildings, etc. This dataset is somewhat similar to the BAG dataset used in our research, but contains more information. By combining these two datasets the created software was capable of reconstructing a city center with acceptable detail for terrain and highly simplified buildings. This is illustrated in 1. Again it is noteworthy to see that the buildings are comprised of few vertices and faces and again the terrain seems to consist of more data points. This is an important factor in the size of the final comprised dataset, especially since terrain will be omitted in our research.

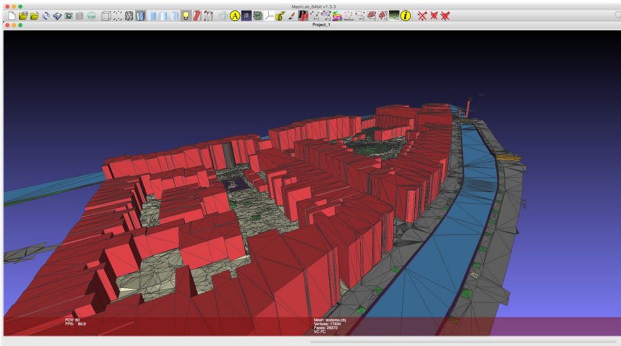

Figure 1: Example of the output of the 3dfier software, with terrain and buildings

3 Research Questions

The main question this research trives to answer is related to the feasibility of mesh reconstruction. Can polygonal meshes be constructed for buildings on a large scale from point cloud data?

4 Data

The project uses two datasets to convert 2D polygons to 3D buildings. The Actueel Hoogtebestand Nederland (AHN2) contains LiDAR point cloud data and is used to obtain height points for all the buildings in the Netherlands. Next, the BAG XML dataset contains geodata and is used to extract the 2D house surface polygons for every building in Netherlands.

4.1 AHN2 Dataset

The AHN2 dataset is a point cloud dataset representing an accurate height map of the Netherlands [1]. Point cloud data consists of a *cloud* of points that each represent an x,y,z location within a certain coordinate system. For the AHN2 dataset this means that for the entirety of the Netherlands such measurements have been taken, with an average of 8 datapoints per square meter. The resulting dataset is stored in compressed form in *LAZ* files, which can be uncompressed to *LAS* files, yielding workable point cloud data. The dataset is 1.6TB in compressed form, consisting of 41,000 LAZ files. These LAZ files have been divided into 143 *tiles* that describe the total surface area of the Netherlands.

This dataset is also visualized in a web application `http://ahn2.pointclouds.nl/`. The application is able to demonstrate the challenge in working with such a large dataset as the website takes a significant amount of time to load. The application is also difficult to navigate due to the computational load it places on the machine viewing the data.

4.2 BAG Dataset

The BAG dataset contains information about registered textitobjects in the Netherlands. The dataset registers many attributes about buildings, such as when a building was built, permits granted as well as their ground floor outlines. This ground floor outline is used in this research as it gives the basic shape of a building.

The BAG dataset is 17GB in size and contains data for 16 million buildings in the Netherlands. Each XML [9] file contains roughly 10,000 houses where each house is described by an average of 8 points (x,y coordinates).

4.2.1 Structure

The BAG dataset is categorized by 7 criterias [10] as stated below

- *Nummeraanduiding* represents each house or building by a unique identification number [10]
- *Openbare Ruimte* contains data on public spaces such as streets and parks [10]
- *Woonplaats* contains data on all municipalities in the Netherlands [10]
- *Pand* contains data on all legally registered buildings in the Netherlands [10]
- *Verblijfsobject* contains data on all spaces situated within a building [10]

- *Standplaats* contains data on spaces that are not directly connected to the Earth but are suitable for residential or commercial purposes [10]

- *Ligplaats* contains data on water bodies [10]

Since this project concerns itself with the generation of 3D models of the buildings in the Netherlands, the data under *Pand* is the only relevant data source.

5 Processing Pipeline

The final pipeline to combine the two datasets resulting in the final 3D models consists of several stages that will be outlined in this section. The key focus in the development of this pipeline is trying to minimize the operations needing to be done on the point cloud data as that is the largest in size and thus the most computationally intensive to work with.

The pipeline has been developed entirely in Python and some of the stages have been executed on the SURFsara Hadoop cluster[11] with PySpark. The choice for Python was mostly related to the possibility to use a Python library called LASpy[12] to work with the point cloud data. This library allows the loading of LAZ/LAS files and working with the contained point cloud data in native Python and Numpy[13], a well known Python library, primitives. Additionally this library relies on the laszip which is part of the widely used LAStools toolkit for point cloud data [14]. The other possibility would have been to use an older point cloud data frame work for Hadoop, IQmulus [15], however since the used cluster uses a newer version of Scala this would incur many technical challenges in getting it to work correctly.

For the actual generation of the 3D models, the 3dfier software from the TU Delft was used as it seemed to be suited to our purposes. Contrary to their documentation, direct classification of the point cloud data did not seem to be needed to create 3D models. That One key factor in the use of this program is that the software also uses LASzip internally to process LAZ files, meaning that the same storage/compute issues regarding this point cloud data would still remain relevant. The software seemed to be able to process about 20GB of LAZ files in an hour [1] with the limiting factor being the single threaded nature of the program. Additionally, it was observed that all of the point cloud data first seems to be held in main memory, as the memory usage increased with each of the LAZ files read. This also sets a limit on large of an area (or the detail of said area) can be processed at once. The software `las2tin` was also tested, as it is part of LAStools. However, this software created meshes that were too detailed and contained weird artifacting as visible in figure 2. Such a mesh would not be suitable for representation and additionally consists of too many data points.

[1] As measured on Windows 10, with an i7-4790, 32GB of RAM and an NVMe SSD

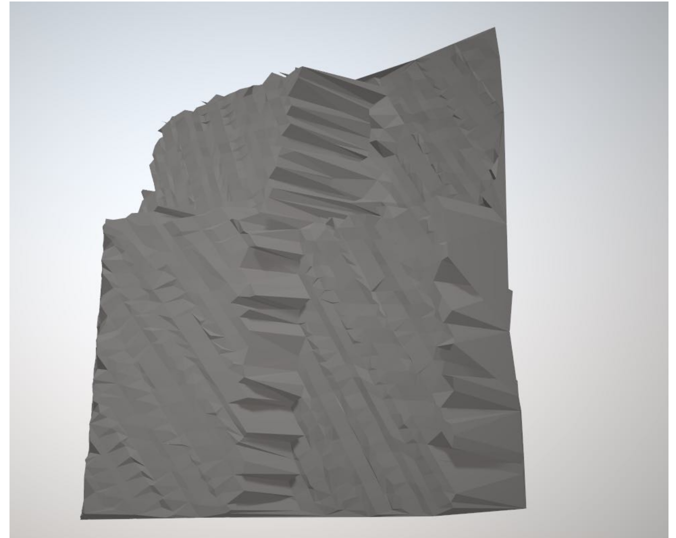

Figure 2: Resultant mesh from las2tin

5.1 Stage 1: Metadata extraction

The BAG data was analyzed in several iterations before relevant data was extracted (referred to as features from here on) from it. This section describes all experiments that were carried out, what was achieved from them and finally what their outcomes were.

5.1.1 Iteration 1: Exploration of dataset

To get a better understanding of the dataset and it's structure, a single xml file was observed. `ElementTree` [16], a Python [17] module for working with xml files was used to first determine the structure of the files. From this exploration, two xml tags namely `identificatie` and `polygon` were determined to be of interest. The `identificatie` tag contains a unique id for each building and the `polygon` tag contains the 2D surface coordinates.

5.1.2 Iteration 2: Extraction of polygon coordinates

The next iteration focused on the extraction of the features mentioned above. This was implemented using `ElementTree`. A test was carried out on 5 xml files and the features were successfully extracted and saved to disk.

5.1.3 Iteration 3: Parallelization & final outcomes

Finally, the above feature extraction code was implemented using Pyspark [18] primitives such as Row [19] and

Dataframe [20] and parallelized. With 12 executors, features from the entire pand dataset was extracted in 30 minutes. The operation produced a 1.5GB compressed parquet dataset [**parquet**] from the original 18GB xml dataset.

5.1.4 AHN2 Point cloud data

For the AHN point cloud data, information was gathered about the shape of each of the 41 thousand files. In principle, each LAZ file is defined as a rectangle with a minimum and maximum x and y coordinate. The main challenge for this step is that the compressed LAZ data cannot easily be directly evaluated. This was also the first test for the software library used, LASpy. This also exposed a first important limitation with the chosen software. Firstly, since Spark has no primitives for LAZ data it needs to be evaluated as pure binary data. Secondly, the used library, LASpy, only works with files and not Spark data primitives.This means that all LAZ are first written to disk on the cluster before being evaluated with LASpy. This evaluation then simply yields the X and Y coordinates defining the rectangle of each LAZ file.

5.2 Stage 2: Coupling buildings and LAZ files

With the reduced datasets a coupling can be made between the two. That is, for each building the corresponding LAZ point cloud files were found. To search efficiently in spatial data, one option to use is R-Trees as an indexing structure [21]. A R-Tree indexes based on rectangular structures, which matches the shape of the LAZ files. The R-Tree consists of all LAZ files, which then allows it to be searched. However, a building is not always a rectangle, nor does it always align with the grid of the LAZ files. A single building might also span multiple LAZ areas.

A naive solution would be to take each point in the outline of a building and search for the corresponding LAZ file. However, since established before, each building is defined on average by 8 points, meaning that a total of 80 million search would have to be conducted.

To reduce the number of searches and be more efficient, each building was represented as a rectangle, which optimizes the search structure. This is visualized in 3. The blue area represents buildings, and the green squares the area that each LAZ file spans. To optimize, first the minimum and maximum coordinates are searched in the R-Tree of the LAZ files. If they yield the same result, then the entire building is in a single LAZ file, otherwise the other 2 coordinates are also searched to see in what LAZ files they might be.

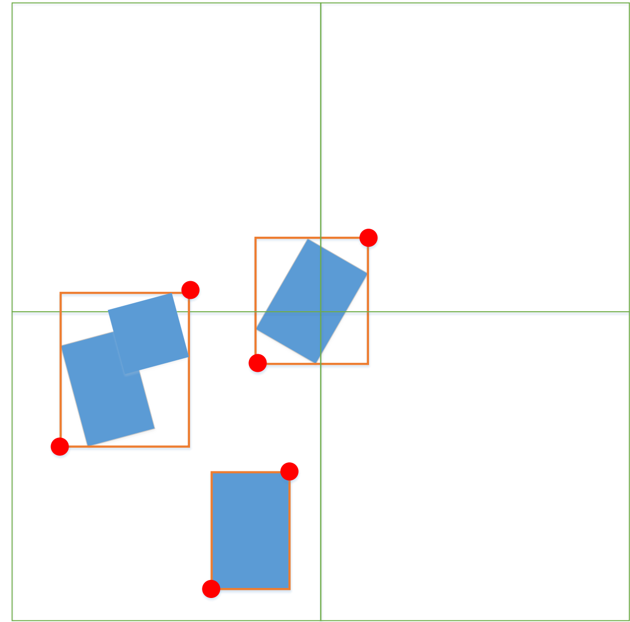

Figure 3: Abstracted representation of the search methodology, blue is the buildings, orange the rectangle containing the building. Green defines the outline of each LAZ file

This finally yields a dataset where each building is linked with the corresponding LAZ files. This stage was done in a partially distributed fashion. The R-Tree creation was done without distribution, merely inserting the data about each of the 41 thousand LAZ files in the R-Tree in an iterative fashion. It would potentially be possible to do this in an distributed fashion by partitioning the data and constructing multiple R-Trees and then merging them afterwards. However, the library used, `PyRtree` [**pyr**] did not support merging multiple R-Trees.

However, the searching for the buildings could be done in an distributed fashion as the R-Tree was fixed at that point. To this end, the dataset containing the buildings was partitioned and each worker had an instance of the created R-Tree for searching. The final resultant dataset has each pnd with the corresponding LAZ files for each of the corners, or only for the left bottom corner if all are in the same LAZ file.

5.3 Stage 3: Creating masks for the buildings

As much of the point cloud data is not related to the buildings, this data is not needed for the creation of the building models. To optimize the final creation of the polygons, as much of the not needed point cloud data was removed from the dataset. As a rough estimate, considering around 8% of the land is used for buildings [22], this should result in a dataset of some 140-150GB of compressed point cloud

data, reduced from the original 1600GB of compressed data. As each building is represented as a rectangle, some extraneous pointcloud data will be included, increasing the size somewhat.

The first part of this is changing the dataset generated in Stage 2. Instead of a building with each LAZ file on a row, each LAZ file is listed with all corresponding buildings. From the buildings only the rectangular shape is retained, defined by the four corners. This set of rectangles that show which area of the LAZ file contains buildings will be referred to as the "mask", *masking* the area that is important. For the example in 3 this would result in the abstracted mask for each of the tiles as shown in 4. The orange areas illustrate the portion of each green LAZ file that would remain of the point cloud data.

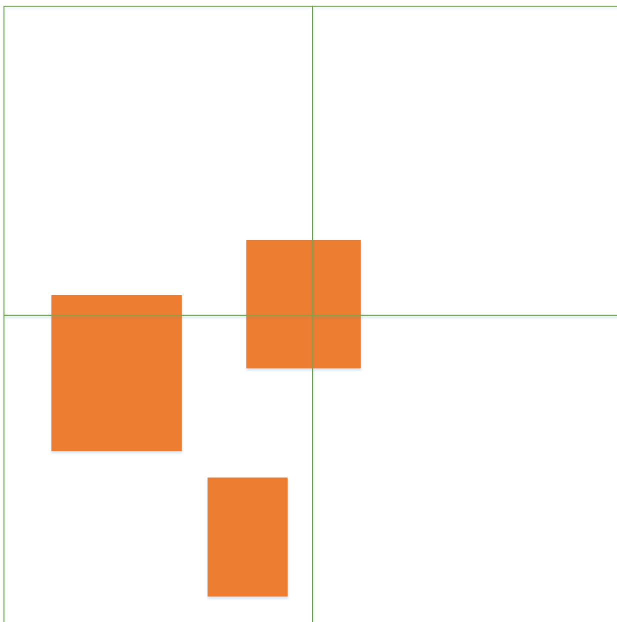

Figure 4: Abstracted representation of the LAZ *masks*, with the orange rectangles being the masks. Green defines the outline of each LAZ file

This creation of the masks was done without Spark due to the limited size of the dataset, reducing the complexity of the code. The resultant data file was a single compressed parquet file of only 413MB.

5.4 Stage 4: Masking the LAZ data

After the masks has been created, the actual processing of the LAZ data can be executed. This process was done on the cluster, by partitioning all the LAZ files that need to be processed. The worker nodes could then download the corresponding files and executing the masking. This is computationally intensive operationg since the LAZ files first need to be decompressed, then loaded into main memory for masking. The masking itself is also costly, since all there can be many small masks, for each building one, that need to be applied to the set of points before the final result is reached. After all masks are applied to the set of points, only the the points that make up the rectangle around the buildings remain. These points are then written to a new LAS file, which is compressed to a new LAZ file and stored in HDFS. This last operation is also costly, but the smaller file sizes help this operation.

The main challenge with regards to this operation is again that the point cloud data is not natively supported in Spark, and in contrast to the metadata extraction in section 5.1.3 multiple LAZ files are processed together. The resultant algorithm needs to first aggregate multiple LAZ files, and their related point cloud data before applying the masking operation. The aggregation step the most time consuming, as each LAZ file first needs to be decompressed before the point cloud data can be manipulated.

This stage finally results in a set of reduced LAZ files for each tile. Due to time constraints on the used cluster not all LAZ files were processed in this manner. However, for a few tiles for which all LAZ files were processed some notable reductions in size were observed. For example, `tile_4_15` was reduced from about 1GB of LAZ files to 1MB, as it contained few buildings. A tile in the center of Amsterdam featured predictably less benefit. The `tile_5_9` which contains the center of Amsterdam went from 11GB originally to about 2.5GB in size, far less of a reduction.

5.5 Stage 4: Generation of polygons

The final actual creation of the buildings works with the previously created subset of the point cloud data as well as the extracted coordinates from the BAG dataset. For this, the 3dfier software developed at the TU Delft was used. This software normally combines point cloud data with more than just building information to also create polygonal information about the terrain. This software was chosen since it alleviated the need to create our own software/algorithm to create 3D models.

The software itself has a fairly naive reconstruction result for buildings, with little detail regarding roofs and such. Only the exact outline of the building is preserved, with the rest of the 3D shape being simplified. Roofs are flattened, creating buildings with a more box-like shape. This has the benefit of reducing the polygons needed to comprise a building and thus the final size of the created dataset. This is especially important for this research as the goal was to

create models of all buildings in the Netherlands. Since the entire data set comprises the around 10 million buildings, thus high detail models would be too large and too expensive to render. In non-compressed OBJ format a single building is around 1KB as presented by the 3dfier software. This would result in about 10GB of 3D models for the entire Netherlands.

The software itself takes two types of input files to generate the 3D models. Firstly it takes in 2D data describing the land usage at ground level. In normal usage of the software this not only includes the outline of buildings, but also roads, water, vegetation. This dataset needs to be in *OGR* format [23]. The second source of data it uses is the pointcloud data. For more precise model creation it requires the pointcloud data to be classified, that is, each point is classified as building, vegetation, etc. The software then uses the two datasets to create a detailed model of an area.

For the purposes of this research, the software was simply provided with 2D data regarding the buildings, with no information about the surrounding area nor were any of the points in the pointcloud dataset classified. With this reduced dataset, the software only produces 3D models for the buildings, with terrain being entirely omitted.

The software was ran separately for each tile, producing a single polygon file for each tile. To this end, all reduced LAZ files for a single tile were given as input for the height data. For the 2D outline data, the original extracted building outline coordinates with the unique identifiers were used. As the 3dfier software expects OGR compliant data, the extracted data from the BAG dataset was encoded in OGR compliant XML files. A sample of the resultant XML data is shown in 5, where the `buildingpart` is a single building, defined with the same GML coordinates as in the original BAG dataset.

```
<ogr:FeatureCollection
    xmlns:gml="http://www.opengis.net/gml"
    xmlns:ogr="http://ogr.maptools.org/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://ogr.maptools.org/ bgt_pand.xsd">
  <gml:featureMember>
    <ogr:buildingpart>
      <ogr:geometryProperty>
        <gml:Polygon>
          <gml:outerBoundaryIs>
            <gml:LinearRing>
              <gml:coordinates>148902.085 524238.491,148909.678 52
            </gml:LinearRing>
          </gml:outerBoundaryIs>
        </gml:Polygon>
      </ogr:geometryProperty>
      <ogr:gml_id>0388100000207658</ogr:gml_id>
    </ogr:buildingpart>
```

Figure 5: OGR XML data suitable for 3dfier

The software was ran on a single machine for all processed tiles as it proved to be difficult to run the program in a distributed fashion. This was mainly due to the program being fairly new and recompiling it to run on the older CentOS 6.9 that is used on the Spark cluster used in this research proved impossible. This is another reason why the previous stage to reduce the size of the point cloud data was vital. Although even if recompilation would be possible, not all tiles could likely be processed on the cluster. The maximum possible main memory usage is about 7GB, which means that for some of the larger tiles it would be likely execution time would greatly suffer due to overflow (and paging to the hard disk). As measured before, on a single machine around 20GB of data cloud be processed in an hour, meaning it would be unfeasible to process the entire original 1.6TB of point cloud data without having it reduced. However, with the reduced data it was possible to run it on a single machine, especially since not all tiles were masked in the previous step and thus also not processed in this step. For the tiles processed, the pipeline does demonstrate its functionality.

The visualization also yielded some interesting artifacts. One such is shown in figure 6. The big white blob is a mesh structure larger than a city, one of the smaller boxes visible is a single building. It seems to correspond to no buildings, in fact there are buildings underneath it. We are not sure where it came from, it is likely an artifact in the BAG dataset that has resulted in an area that is not actually a building being presented as such. The 3dfier has simply lifted the terrain and treated it as a building, as the point cloud data has no classification in our dataset. There might be more instances of this, but it would be hard to check all 10 million buildings. Point cloud classification might help with this problem, if it would be more wide spread.

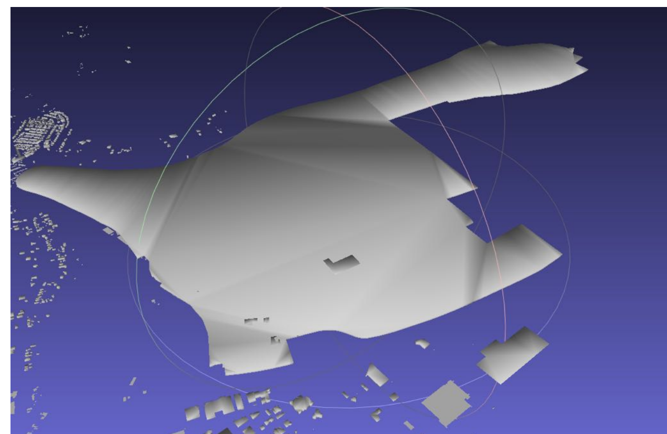

Figure 6: A weird mesh found in tile 5 6 after rendering

Finally, as the constructed dataset is not complete, some of the tiles contain “holes”’, where the suitable LAZ files were not processed in the masking fases. These holes manifest themselves as 1 dimensional or partially one dimensional

buildings, where the 3dfier software simply did not have the point cloud data to *lift* them into a 3d shape and thus only created the outline.

6 Visualization

In order to best visualize the entire 3D dataset, we created a map of the Netherlands using the tile files that were generated from the *3dfier* on the cluster (see Figure 7). The tiles are colour coded with green representing a fully loaded tile; orange representing a partially loaded tile; and grey indicating a lack of tile. When clicked, the relevant building files get rendered in the canvas below (see Figure 8).

Figure 7: The map of Netherlands represented by tiles

Figure 8: The Canvas renders the buildings based on the tile clicked

6.1 Creating the Visualization

The main goal in terms of visualization was to effectively display buildings on the web. The user must also be able to have access to basic navigation through the buildings - zooming in and out and panning. The website would then be divided into two main sections in terms of design: an upper interactive map area followed by a lower `canvas` with the viewable buildings. This `canvas` that would be rendered based on the tile clicked.

For the interactive map, a simple SVG image would be displayed where a valid tile would call upon a specific file and pass it onto the canvas below. However, deciding on the methodology for the 3D rendering was harder. The project scope was set to allow only the use of server-static *HTML* and *JavaScript*; and so, the usual techniques of displaying large files with 3D data on the web had to be dismissed. Based on initial research ([24], [25], [26]), it was decided that the actual rendering of the 3D buildings would be done using *WebGL* [27]. *WebGL* directly accesses a computer's graphics hardware and then renders the output onto the `canvas` element without the use of any plugins [28]. Further, as it uses *OpenGL* [27] as its framework, we were assured that most of the popular browsers would have support for it [28]. There was no clear answer as to which *JavaScript* library would be best to use on top of *WebGL*, so experiments had to be conducted to determine that. The entire process of visualization occurred in several stages wherein, in each stage, a portion of the visualization was attempted using a specific tool and technique.

6.1.1 Iteration 1: Initial Look at a 3D Building

The *3dfier* generated the building tile files in an *OBJ* format so it was essential to get a sense of what a tile looked like. A test file with a few buildings was visualized locally using *MeshLab*[29], and a few points were noted. First, the software rendered the building clusters far away from the origin. This made it hard to locate them at first sight (see Figure 9). Moreover, our data lacks terrain and other ground features so a lot of empty spaces got rendered in areas. Next, all the buildings in the file are stored as a single 3D object rather than individual buildings so it was not possible to zoom into a particular building and that made navigation through the "city" rather inefficient. This initial assessment helped us set expectations in terms of what limitations we had and, give us an idea of what we could expect in terms of rendering on the web.

Figure 9: MeshLab rendered a test tile file with hard to see buildings. The building clusters are marked in red.

6.1.2 Iteration 2: Rendering a Single Building with three.js

Based on a recent report that modeled 3D cities [30], it was decided that *three.js* would be our best option as a *JavaScript* framework for rendering *WebGL*. The biggest reason was that we required a way to gain maximum control over displaying our buildings, given the restrictions we saw in our local test on *MeshLab*. *three.js* requires three main components to work - a `scene`, `camera` and a `light` [27]. So now this meant that we could manipulate each component individually to best showcase the buildings, with `camera` being the most important one. Another reason for choosing *three.js* was that it could be initialized with web renderers other than *WebGL* as a fallback to account for older browsers. Finally, to counter the large size of our data files, we had the option to set resolution of our buildings to a lower value and speed up our loading time.

The implementation was done in small pieces so the test tile file was used to try and generate a 3D model. However, despite several attempts, while `console.log()` confirmed that the buildings had rendered, they were just not being displayed on the canvas.

6.1.3 Iteration 3: Rendering a Single Building with babylon.js

babylon.js is another popular but newer alternative to *three.js* that caught our attention. *babylon.js* is very performance focused and we wanted to try and take advantage of that [31]. A major performance boost that *babylon.js* provides is that it can reuse a mesh's properties and geometry. As our data contains buildings wherein several of them have identical meshes, we could potentially reduce the amount of information sent to the GPU for rendering [31]. However, implementing the 3D buildings with *babylon.js* proved to be a problem. As *babylon.js* focuses on game development, it has several strictly defined methods to enable easy development with fixed parameter requirements. For instance, while trying to use `sceneLoader` to load a scene, we had to pass a texture file which our data not have. There was no option to skip that parameter. So while *babylon.js* would have been a good choice on paper, in reality, it was not flexible enough for our needs.

6.1.4 Iteration 4: Using 3DViewer and JSModeler

In order to have a suitable and working visualization, we decided to incorporate the source code from an online 3D image rendering software *Online 3Dviewer* [32] that used *three.js* and *JSModeler* [33]. This worked fairly well as a temporary setup. The only problem now was that it would take approximately 35 seconds to render and display a single tile file on the `canvas`. The conventional standards for loading times on the web are defined to be anything less than 3 seconds so in comparison, our loading time was very poor [34]. With the *OBJ* files being around 80MB in size and, with the additional time required by our stack to create materials, apply them and set up a scene, altogether, it was a rather slow process. To counter this, *ctm* - a compressed 3D format for the web was discovered as a much more efficient alternative [35]. The next hurdle was to incorporate a *ctm* loader within the current implementation we had. The source code that we had incorporated used two different libraries to render 3D images so incorporating a new format loader would not be possible given the time limitations. It would require us to comb through two different libraries and try to incorporate *JavaScript* to make both libraries compatible a *ctm* file.

The original implementation of *three.js* from the second iteration was revisited. The issue at the time prevented our buildings from being displayed on the `canvas`. It was realized that the buildings were being originated very far from the origin and hence were not being displayed on the limited `canvas`. With *js-openctm* - a *JavaScript* library for reading *ctm* files, we were able to create the required data structure for *three.js* (see Figure 10) and then incorporate a *ctm* loader. From the original 80MB, *ctm* files got compressed down to 8MB, a big 90% reduction in size. This resulted in significantly faster loading time of 2.5 seconds for the files on the web.

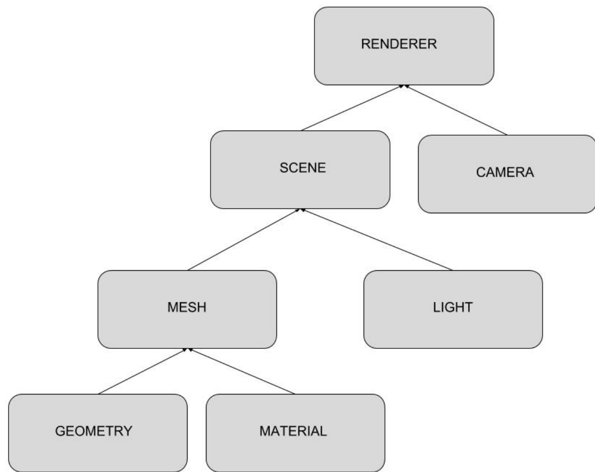

Figure 10: The three.js Data Struture that was created to render 3D buildings

6.1.5 Iteration 5: Bringing Everything Together

For the interactive map of Netherlands, we had to generate a custom SVG image. All the maximum and minimum x and y coordinates from the *LAZ* tile files were calculated and plotted using *matplotlib* [36] and *matplotlib.patches* [37]. *matplotlib* is a popular Python library that enables easy 2D plotting among other things [36].*matplotlib.patches* further allows for rectangles to be plotted. By mapping the coordinates from each and every file in each and every tile, we ended up with a very dense tile map (see Figure 11).

Figure 11: The map plotted with all the x and y coordinates from all the laz files in all the tile files

To simplify the image, we instead grouped all the files under each tile and then took the maximum and minimum x and y coordinates from each of the tile files. This resulted in a cleaner SVG (see Figure 12).

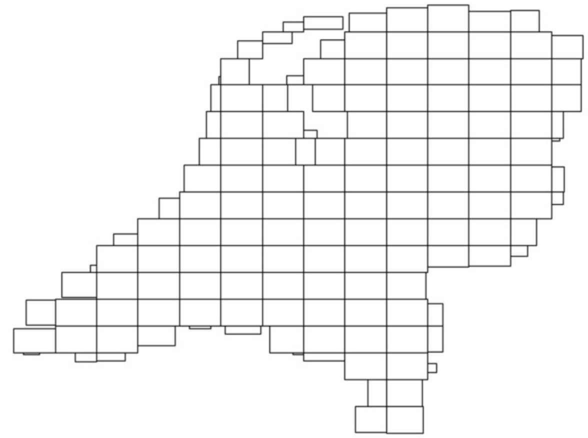

Figure 12: The map plotted again with x and y coordinates from the tile files

We embedded an `onclick` event on relevant tiles such that on choosing the tile, the `main` function gets called. This function accepts a string parameter and passes it along to the `loader` which then loads the appropriate file and renders it on the `canvas`. Due to our restrictions in generating all the tile files from the cluster, we decided to colour-code our tiles to depict the data it represented - green for a fully loaded tile, orange for a partially generated tile and grey for an unavailable tile. Our final visualization architecture can be seen in Figure 13.

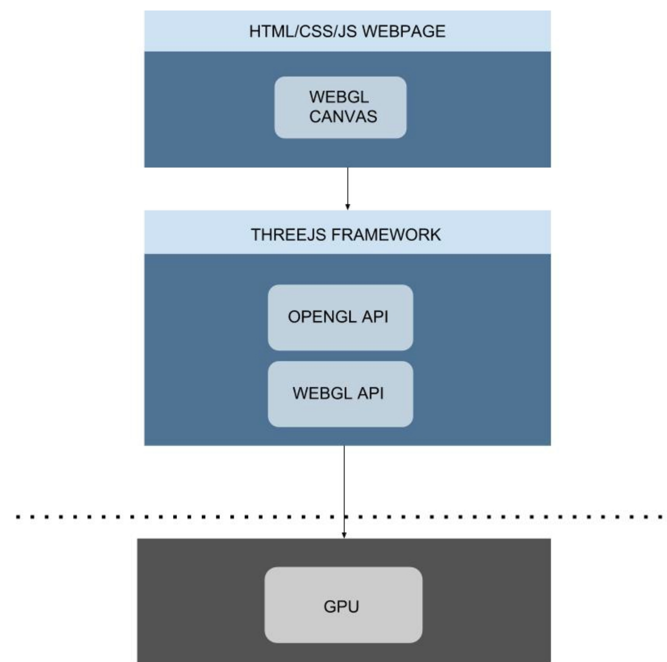

Figure 13: The final visualization architecture

6.2 Technical Challenges

The biggest challenge in terms of visualization was to display the buildings such that they could all be rendered given the technical constraints of the project. The ideal way to render such large 3D files on the web is to convert real-time 3d rendered images to a streaming video. The browser is then effectively a video player. [38]

With less background in JavaScript, following through some of the complex examples in *three.js* and debugging was not easy. The *js-openctm* is not well documented and well maintained on GitHub. Thus implementing some of the functions of this library was messy with no examples or documentation to fall back on. Finally, much of the technology that renders large 3D files to generate complex cities are proprietary and thus not possible to learn from. The entire process of using *three.js* and *js-openctm* was more of a trial and error type of coding.

7 Conclusions & Future research

While the current project manages to convert 2D plans to 3D buildings, there are a few things that could be improved. First, our data is lacking all terrain and geological features. Including it in our data would result in a far more cohesive and complete 3D model. As the datasets do not necessarily have data from the exact period, buildings may differ. For example, the BAG dataset could contain newer buildings that we did not filter out, for which the point cloud data set has no corresponding data. This could be overcome by making sure the data used is from the same period or having a query that selects on this. Also, while the building outlines in our final data are fairly accurate, we lack information regarding roofs, windows and doors. Thus the complexity of our building shapes overall can certainly be improved. The buildings in the tile file got generated as a single object. Ideally, each building would have been saved as a single object. This would have allowed for a much involved navigation in the visualization phase. Currently, our visualization only allows us to move across the space of the file, rather than through it. The developed pipeline shows that it is possible to generate 3D models for an entire country. The created visualization shows that it allows the dense point cloud data for an entire country to be visualized in more light weight manner. This gives far greater insight into the urbanization and spread of buildings in a country.

For future research it would be useful to apply the pipeline employed on this dataset on a larger scale on the Netherlands. As demonstrated, it is possible to create reasonably sized 3D models of the buildings in a city, allowing for rendering in a web application. However, as stated, no terrain was processed. It would be useful to broaden the scope of the developed software to similarly to the 3dfier software also include terrain in some form. At the very least some sort of overlay showing the terrain from satellite images would be a useful addition to understand where the buildings are located. A key point of potential improvement would be to make the software into better integrated pipeline. For instance, working with the point cloud data in Spark was not ideal and can be optimized. Additionally it would be useful to adapt the 3dfier software to serve as a library so it too can be employed in a distributed fashion. As the level of detail from the created meshes is low, it is likely also possible to strike some sort of other balance here. Either creating the buildings with more detail, with terrain but then going for smaller areas of the Netherlands to still yield small and thus web application deliverable file sizes.

References

[1] *Actueel Hoogtebestand Nederland*. URL: `http://www.ahn.nl/index.html`.

[2] O. Oguz et al. "Production and Visualization of 3D City Models from Building Allocation Plans". In: *2006 IEEE 14th Signal Processing and Communications Applications* (). DOI: `10.1109/siu.2006.1659911`.

[3] Yan Liu, Iderlina Mateo-Babiano, and Sebastien Darchen. *How virtual 3D modelling and simulation can help us create better cities*. Sept. 2018.

[4] .

[5] Florent Lafarge and Clément Mallet. "Creating large-scale city models from 3D-point clouds: a robust approach with hybrid representation". In: *International journal of computer vision* 99.1 (2012), pp. 69–85.

[6] George Vosselman, Sander Dijkman, et al. "3D building model reconstruction from point clouds and ground plans". In: *International archives of photogrammetry remote sensing and spatial information sciences* 34.3/W4 (2001), pp. 37–44.

[7] *TU Delft 3dfier*. URL: `https://github.com/tudelft3d/3dfier`.

[8] Kavisha Kumar, Hugo Ledoux, and Jantien Stoter. "Compactly representing massive terrain models as TINs in CityGML". In: *Transactions in GIS* (2018). In press.

[9] Tim Bray et al. "Extensible Markup Language (XML)." In: *World Wide Web Journal* 2.4 (1997), pp. 27–66.

[10] *Basisregistratie Adressen en Gebouwen*. URL: `https://www.kadaster.nl/basisregistratie-adressen-en-gebouwen`.

[11] *Description of the Hadoop cluster*. URL: `https://userinfo.surfsara.nl/systems/hadoop/description`.

[12] *LASpy*. URL: https://github.com/laspy/laspy/.

[13] *NumPy*. URL: http://www.numpy.org/.

[14] *LAStools*. URL: https://rapidlasso.com/lastools/.

[15] *IQmulus*. URL: https://github.com/IGNF/spark-iqmulus.

[16] *The ElementTree XML API*. URL: https://docs.python.org/2/library/xml.etree.elementtree.html.

[17] *Python*. URL: https://www.python.org/.

[18] *Spark Overview*. URL: https://spark.apache.org/docs/2.1.1/.

[19] *Spark API reference: pyspark.sql row*. URL: https://spark.apache.org/docs/2.1.1/api/python/pyspark.sql.html#pyspark.sql.Row.

[20] *Spark API reference: pyspark.sql dataframe*. URL: https://spark.apache.org/docs/2.1.1/api/python/pyspark.sql.html#pyspark.sql.DataFrame.

[21] Antonin Guttman. *R-trees: A dynamic index structure for spatial searching*. Vol. 14. 2. ACM, 1984.

[22] *Bodemgebruik; uitgebreide gebruiksvorm, per gemeente*. URL: https://opendata.cbs.nl/statline/#/CBS/nl/dataset/70262ned/table?dl=3D8B.

[23] GDAL/OGR contributors. *GDAL/OGR Geospatial Data Abstraction software Library*. Open Source Geospatial Foundation. 2018.

[24] Bartosz Sawicki and Bartosz Chaber. "Efficient visualization of 3D models by web browser". In: *Computing* 95.S1 (Aug. 2013), 661fffdfffdfffd673. DOI: 10.1007/s00607-012-0275-z.

[25] Jurgen Discher Dollner. "A scalable webGL based approach for visualizing massive 3D point clouds using semantics-dependent rendering techniques". In: *Proceedings of the 23rd International ACM Conference on 3D Web Technology* (2018). DOI: 10.1145/3208806.3208816.

[26] Lifeng Liu and Jinyun Fang. "An accelerated approach to create 3D Web cities". In: *2009 17th International Conference on Geoinformatics* (2009). DOI: 10.1109/geoinformatics.2009.5293461.

[27] *WebGL-OpenGL ES for the Web*. July 2011.

[28] Khronos Group. *The Industry's Foundation for High Performance Graphics*.

[29] *MeshLab*.

[30] *3D building threejs*.

[31] 2018.

[32] Kovacsv. *kovacsv/Online3DViewer*. Jan. 2017.

[33] Kovacsv. *kovacsv/JSModeler*. Nov. 2016.

[34] Dr Manhas. "A Study of Factors Affecting Websites Page Loading Speed for Efficient Web Performance". In: *International Journal of Computer Sciences and Engineering* (Dec. 2013).

[35] *OpenCTM - Compression of 3D triangle meshes*.

[36] *Installationfffdfffd*.

[37] .

[38] Dante Abate et al. "Remote rendering and visualization of large textured 3D models". In: *2012 18th International Conference on Virtual Systems and Multimedia* (2012). DOI: 10.1109/vsmm.2012.6365951.