Abstract

The aim of this project is to create a three dimensional (3D) model of a part of a city from a single two dimensional (2D) image. The goal of this study is to have an idea of what kind of result is achievable, what kinds of problems arise while implementing it and how to overcome them. This is fulfilled by choosing and implementing an algorithm that can extract buildings from a 2D satellite image from Google Earth and recreate the extracted buildings in three dimensions.

The project is undertaken because of the emergence of application mixing 3D models and 2D images like Google SketchUp for example or applications in virtual reality. Existing methods use stereoscopic systems, images from video or maps to create a realistic 3D model of a city. This project is challenging because generating a 3D model from a single 2D image encounters the problem of the accuracy of the building extraction and above all of the depth inference. However having this kind of software would enable the general public to access 3D models on a large scale, directly from Google Earth for example, and would reduce the existing costs for 3D related industry like the video games and special effects industry or the architecture and civil engineering.

The project is divided into several steps: the first one is to extract the edges of the buildings. The next one is to use these edges to create building hypotheses by splitting the image in homogeneous polygons. The following step selects the polygons that are more likely to represent actual buildings. Finally the height of the buildings is inferred from the shadow they cast and the final buildings are modelled in 3D from the previously extracted data.

The main achievements of this project are:

- The implementation and adaptation of Boldt algorithm (Boldt M., 1989) to extract building edges.
- The implementation of the idea developed in Sohn & Dowman's paper (Sohn, et al., 2001) which uses a binary tree to split the image in homogeneous polygons and group them to have building hypotheses.
- The design and implementation of several selection criteria to keep the most likely polygons.
- The design and implementation of a method to compute the height of a building based on cast shadows and the creation a simple 3D model in VRML to visualize the result.
- The research and use of the optimal parameters for different types of images: different scales and different contents, and the running of the algorithm to analyze the differences of result, the cases where it performed well and the limitations of the algorithm.

Acknowledgements

I would like to thank my supervisor Erik Reinhard for his availability and his advice throughout all the steps of the project.

I wish to express all my affection and gratitude for my family who encouraged me and supported me financially during the last year.

I would also like to thank Aracely Trevino Vazquez for her support and friendship all along the year.

Table of contents

1 - Introduction

The objective of the project is to have an idea of what kind of accuracy and performance is achievable while trying to reconstruct a 3D model of a city from a single satellite image.

1.1 – Aims and Objective

This assessment is done by breaking down the problem into several parts and, for each part, by finding the best algorithm among solutions that are described in the scientific literature or by using some techniques or criteria that would fit the purpose of this project. After implementing the resulting algorithm, tests are carried out to assess the result and solutions are suggested to enhance the algorithm or overcome some problems.

The aims of this project are:

- Choosing, implementing and testing an algorithm to extract building edges from aerial images.
- Choosing and implementing the feature processing to create building hypotheses.
- Implementing and testing the building selection step to have the final outline of the building roofs.
- Inferring the height of each building from the image.
- Generating the corresponding 3D model.
- Testing the algorithm on several kinds of images and assessing the result.

1.2 – Organisation of the dissertation

This dissertation consists of three main parts.

The first part focuses on the methods used in each step of the algorithm as they are described in articles. I describe these algorithms and explain why they are chosen as well as their pros and cons.

The second part describes the implementation of these methods, including the design and development of the software, as well as encountered problems and the solutions proposed.

The third part of the dissertation tackles the results of the tests that are carried out and give a global analysis of the strengths and limitations of the algorithm.

2 - Background and previous work

Modelling cities in 3D is a very active research field because these models are a great source of knowledge for areas such as town-planning, the military or in the entertainment industry for example. The availability of a wide range of data: high-resolution satellite images, depth information from Light Detection And Ranging (LIDAR) or Digital Elevation Model (DEM) has led to very accurate and complex models. Most of the methods found in the literature use several kinds of input to extract information from images, either to locate the buildings or to infer depth. The specificity of this project lies in the input used to generate the 3D model which is a unique 2D satellite picture. The different techniques or algorithms used in this project are either: parts of previously used algorithms that have been implemented and adapted to this application, or techniques inspired by some of the articles found during my background research.

This part focuses on the solutions available to achieve each single part of the project and on the chosen techniques. I describe every technique that is used as well as its strengths and shortcomings, and provide a description of the constraints induced by the context and the result I want to achieve to explain my choice.

2.1 – Extracting building edges

Creating a 3D model of a city can be broken down into two main steps: getting the footprint of the buildings and finding their height. To detect the footprint of the constructions, the usual way is to detect the roof of the blocks, there are several methods to do that: detecting the edges of the roofs and processing them to build polygons that are likely to be buildings, detecting regions with similar textures that are features of constructions roof or detecting corners and processing them to reconstruct blocks.

2.1.1 – Edge-based methods

This part focuses on techniques that detect edges. I deal with Canny algorithm, the Hough transform, the Nevatia-Babu line finder, the Burns algorithm and the Boldt algorithm.

The Canny edge detector (Canny, 1986) is a widely used edge detector that is as a basis for more complex algorithms. There are several steps in the algorithm:

- First the noise in the image is reduced to avoid detecting lines where there is only noise, this is done with Gaussian filtering.
- The gradient intensity and direction are computed. The gradient angle is rounded to one of the four directions 0°, 45°, 90° or 135°.
- Thresholding: the algorithm uses two thresholds to detect weak and strong edges. The weak edges are detected only if they are connected to strong edges to reduce the possibility to include noise in the final edges.

Canny is an efficient, adaptable edge detector, it can detect a lot of edges and locate them properly, the edges are thin and not very sensitive to noise. However, it gives very fragmented edges and does not result in straight line segments. Moreover, in very complex scenes like in the images coming from Google Earth it needs post-processing to get rid of edges that are not part of buildings. That is why it can only be used as a basis for another algorithm that would select the most likely edge buildings for example.

The Hough Transform is a common method used to detect shapes in images. The detection of lines is based on the fact that a line can be described in two ways (Duda, et al., 1972). On one hand it can be described by its slope *a* and y-intersect *b* with its equation $y = ax + b$. On the other hand it can be described by two parameters *r* and *theta* that represent the distance of the line to the origin and its angle. This second equation can be written: $r = x \cos \vartheta + y \sin \vartheta$.

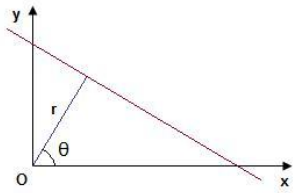


Figure 1 - The two ways of describing a line

In the (r,θ) plane which is also called Hough Space, each line can be represented as a point. The line that goes through point (x_0,y_0) in image plane is a sine curve of equation $r = x_0 \cos \theta + y_0 \sin \theta$. If some points are aligned in the original image, all the corresponding sine curves in the Hough Space cross at a unique point (see Figure 2). This point (r_0, θ_0) defines the line that goes through all these points in image space.

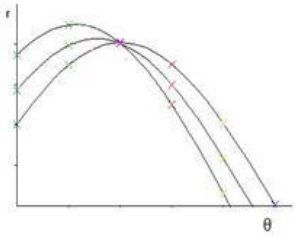


Figure 2 – Three aligned points correspond to three curves in the Hough space. These curves cross at the point (r_0, θ_0) that defines the line these points lie on.

The inconvenient of this method is that it does not take into account the continuity of the edges (Venkateswar, et al., 1992); pixels can increase the probability to have a line even if they are not connected. On the other hand this can be an advantage as this can detect lines with gaps due to shadows or vegetation for example.

The Nevatia-Babu line finder is another technique to extract edges from images. It appears a lot in the literature, partly because Ramakant Nevatia, who first described it in 1980, has written a lot of papers about building detection and extraction since. He uses his line finder as a basis to get information about edges (Nevatia, et al., 1998) (Nevatia, et al., 1989). But this method has also been used by other people, like in (McGlone, et al., 1994) to detect horizontal and vertical lines.

The Nevatia-Babu line finder uses six 5-by-5 masks to find edges in six directions : 0°, 30°, 60°, 90°, 120° and 150° (Nevatia, 1982) here is an example of some masks.

-100	32	100	100	100
-100	-78	92	100	100
-100	-100	0	100	100
-100	-100	-92	78	100
-100	-100	-100	-32	100

100	100	100	100	100
100	100	100	100	100
0	0	0	0	0
-100	-100	-100	-100	-100
-100	-100	-100	-100	-100

Figure 3 - Left: edge mask in direction 30° - Right: edge mask for direction 90°

The image is convolved with these masks, for each pixel the convolution with the mask that gives the maximum output gives the orientation of the edge pixel and its magnitude. Then the edges detected are thinned: a pixel can be an edge pixel only if the pixels next to it in the direction normal to the edge have smaller magnitude than its magnitude. Like the canny edge detector, this gives accurate and thin edges but still separated edges. Nevatia and Babu link these edges by looking at the eight neighbours that surround each edge pixel, and link two edge pixels based on simple rules like the orientation of the edge pixels.

The Burns algorithm is another edge-based method used to extract straight lines. It has been used in (S. U. Zheltov, 2001) before linking collinear segments into longer lines. It is also used in (Sohn, et al., 2001) where the orientation of segments extracted by the algorithm are adjusted to the main buildings directions in the image found with a Fourier Transform.

The Burns algorithm (Burns, et al., 1986) extracts first the gradient magnitude and direction with 2x2 masks to find the gradient in the vertical and horizontal directions.

1	1	1	-1
-1	-1	1	-1

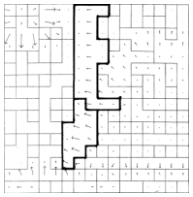


Figure 4 – Left: horizontal and vertical masks - Right: segmentation of the gradients based on their orientation from (Burns, et al., 1986)

The second step is to group pixels with similar gradient orientation, to do that the gradient orientations are quantized and gradients with the same quantification create “line support regions” (LSR). Now, each of these regions can be approximated by a planar surface (figure 5). This planar region is then intersected with a horizontal plan corresponding to the average gradient intensity. The resulting straight line is the detected edge that corresponds to the support region.

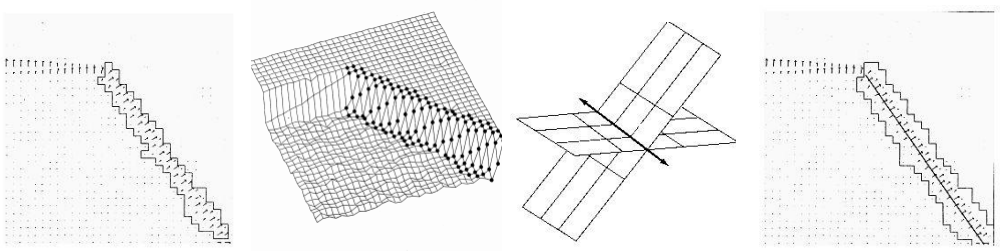


Figure 5 - From left to right: LSR from gradient orientation segmentation, the surface corresponding to the LSR, intersection of the previous surface and the plan of average gradient intensity, the resulting straight line corresponding to the LSR. From (Burns, et al., 1986)

The Burns algorithm has a good robustness to noise, it can extract lines with low contrast and result in an accurate straight line but it seems quite complex to implement.

Another solution that fits our problem is the Boldt algorithm (Boldt M., 1989). It is used in (Woo, et al., 2008) and in (Jaynes, et al., 1994). It is based on grouping edge segments in longer lines following some simple constraints. The first step of the algorithm is an edge detection step. The algorithm in itself focus on the grouping of edge pixels in straight lines and it can be used with any edge detector as long as the location of the detected pixels is accurate and the detector is robust. In his paper, Boldt experimented with a Laplacian operator and an edge detector similar to Canny. The contrast of the edge is also needed in the next steps of the algorithm.

After extracting edges, the grouping process is made of three steps: linking, optimization and replacement. This is done by creating a graph where lines are node and are linked if they satisfy the following constraints:

- Proximity: the line intersects a circle which centre is the end point of another line and which radius depends on the length of this line.
- Collinearity: the difference of orientation of the lines and the distance between the lines must be less than a threshold.
- Proximity of end-points: the distance between the end-points of the lines must be less than a threshold.
- Overlap: the overlap of the lines must be less than a percentage of the length of the lines.
- Contrast : the contrasts of the lines must be similar

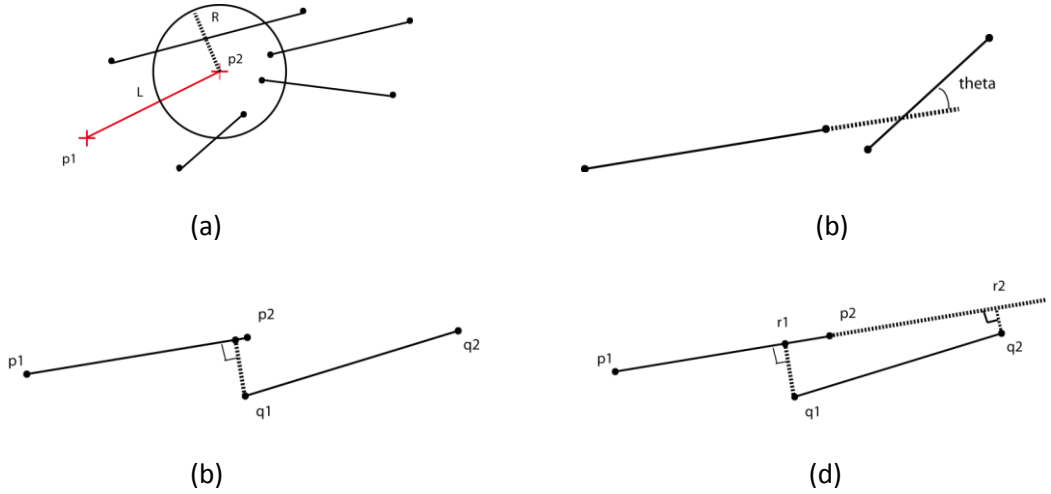


Figure 6 – (a) The proximity constraint. (b) The collinearity constraint. (c) The proximity of endpoints (q1 and p2) constraint. (d) The overlap constraint: the distance r1 p2 must be small compared to p2 r2.

In the optimization step, the length of a line sequence is first limited, then for a token every possible line sequence that goes through it is created. For each possible path a straightness test is done based on least-squares method. The line which score is the best and above a threshold is selected and replaces the lines involved in its creation. These steps are iterated several times with an increasing threshold of proximity as the length of the tokens increases. At the same time, very small tokens that are not linked to any other tokens are deleted because they are unlikely to belong to any significant building.

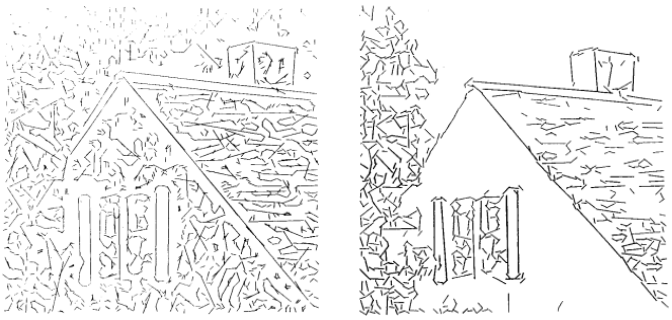


Figure 7 – Left: result of Boldt algorithm - Right: result of Burns algorithm, there are less lines that are more fragmented but the location of the common lines are similar
From (Boldt M., 1989)

2.1.2 – Region-based methods

Contrary to edge-based methods, region-based methods try to delineate areas based on their inside properties like texture or intensity and not on their boundaries. This is a way to detect buildings in aerial images without looking for a shape but for uniform colour or texture areas, it adds information to building edges detection. Region-growing and split-and-merge techniques are briefly explained as they are the most well-known techniques and are used in several of the papers that were mentioned before.

Region-growing is a technique based on similarity between pixels. The first step of the algorithm (Petrou, et al., 2004) is to choose a set of seeds that are the starting points of the growth of the regions. The neighbouring pixels are checked to see if they verify a similarity relationship with the seed. This criterion can be pixel intensity or colour difference for example. If the condition is verified the pixel is added to the region. Once no pixel can be added any more, we have the final area and the process is iterated over another region.

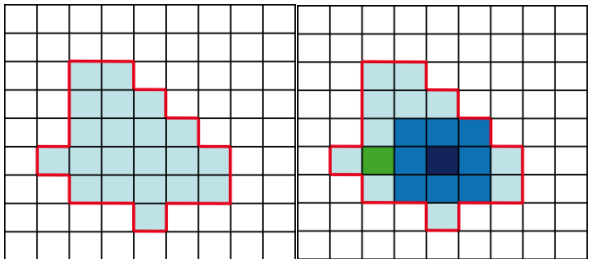


Figure 8 – Left: The original image with the outline of the shape to detect in red. Right: The seed is in dark blue, the pixels that are already part of the detected region are in blue, the tested pixel is in green.

This algorithm is simple and works well when the edges of the regions are well defined, however it requires someone to choose the seeds. We cannot automate this part if we do not have any clues about the location of the regions to segment. It would require previous knowledge about the location of the buildings. Moreover, initializing randomly the seeds can lead to very bad result. As a consequence, we could possibly use this method if we had hints about where to put the seeds.

The split-and-merge method uses a technique similar to the previous one for the merging part but it first splits the image based on its homogeneity. The algorithm is divided in two parts: the splitting part and the merging part. A criterion of homogeneity is first decided based on colour, intensity or texture. If an area (beginning with the whole image) is not homogeneous according to this criterion, the region is split in four parts. The splitting part is recursively repeated on every resulting region. At the end of this process the image is split in parts that are all homogeneous. Then adjacent regions that are pairwise homogeneous are merged.

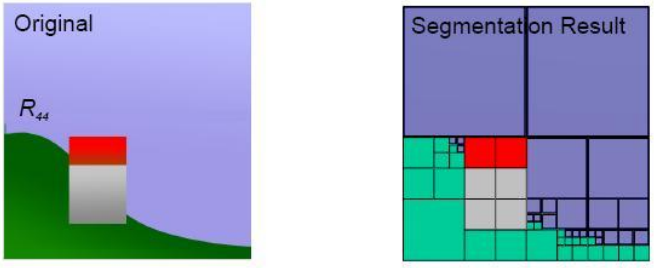


Figure 9 – Left: the original image. Right: the result of the algorithm, the image is first split in a grid and then areas are merged like in the region-growing algorithm. Areas that have the same colour are homogeneous. From (Burghardt, 2010)

2.1.3 – Corner-based methods

Another way to detect buildings can be to detect corners as we can assume that buildings are mostly aggregations of rectangular shapes. Besides, corners are more robust than edges as they are defined by two edges orientations.

In (Jaynes, et al., 1994) the authors use masks to find the corners in the images.

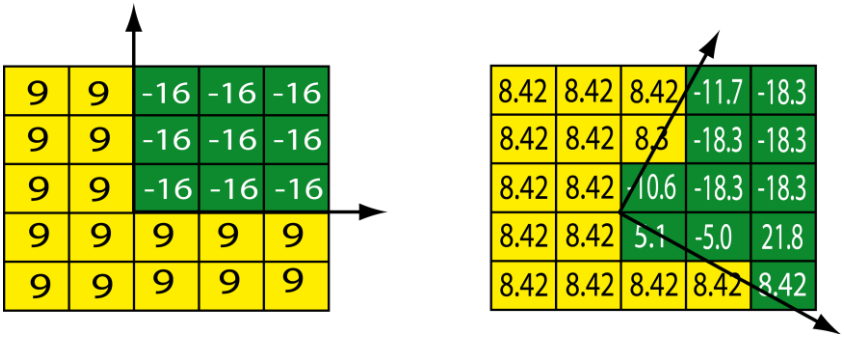


Figure 10 – The original mask on the left and the transformed one that will be convolved with the image from (Jaynes, et al., 1994)

The authors assume that the city they are modelling is grid-aligned, which reduces the different number of corners types to four. They transform the mask to convolve with the image so that it is aligned with the buildings, as the edges of the buildings are probably not vertical or horizontal in the image.

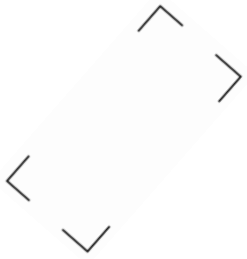


Figure 11 – the masks enable the detection of four kinds of corners

The detected corners are then used to rebuild buildings by comparing their orientations and linking them with one another.

2.1.4 – Chosen algorithm

From the previous part we can first see that the corner-based method cannot be applied to our problem because it assumes that every building in the image has the same orientation, and this is a simplification that cannot be used on images of European cities.

Then, region-based techniques work well when the edges of the regions are well defined and this is not always the case with satellite images. Another reason not to choose this kind of technique to detect the building footprints is that I use a similar technique in the next stage (see 2.2.3) and that it is good to use both edge-based and region-based techniques to have a more robust result.

Among the remaining edge-based method, I decided to choose the Boldt algorithm because it includes efficient methods like the canny edge detector and the Hough transform that cannot be used alone to detect building edges. It is reliable and easier to implement than the Burns algorithm, and every parameter can be adjusted to have the expected result.

The aim of this stage is to have at the end some segments that represent the edges of the buildings in the image. The constraints are:

- Accuracy : detecting edges at the right location (edges of building roofs)
- Uniqueness : detecting one edge for every roof edge (no fragmented segments of the same edge and no duplicates)
- Future proof : having elements that can be processed in the next steps (segments and not only edge points)
- Length: segments of the right length.

The Boldt algorithm achieves these goals by using first the canny edge detector. The result of this first step is an image with edge pixels that need to be linked with one another. The edges are reliably detected; they are thin and not very sensitive to noise, which ensures the "accuracy constraint".

The next step in the Boldt algorithm is to turn the edge pixels into edge segments. This is done by applying the Hough transform to the image obtained with the canny edge detector. A probabilistic use of the Hough transform can give us segments instead of lines, these segments are called *tokens* in the following parts and represent the detected edges of buildings. This part of the algorithm ensures the "future-proof" constraint of the result.

The optimisation, linking and replacement steps insure the "uniqueness" and "length" constraints.

2.2 - Creating building hypotheses

This part of the algorithm focuses on turning the tokens found in the previous step into potential buildings. It relies heavily on the quality of the previous stage. Available techniques for this step include : aggregating tokens into parallel lines, U-structures and finally rectangles (Nevatia, et al., 1989) (Woo, et al., 2008) (Nevatia, et al., 1998) (S. U. Zheltov, 2001), or using tokens to find corners and linking these corners to find buildings (Woo, et al., 2008) (Jaynes, et al., 1994). The chosen method must lead to some buildings representation (polygons) that can be easily selected in the coming steps. Ideally the polygons would already fit the main buildings of the image.

I do not use a method that aggregates the tokens into more complex structures as mentioned earlier because it generates a high number of rectangle or parallelogram hypothesis that need to be selected and eliminated. Moreover it takes independently every problematic situation (like building overlapping or containment) and tries to find a solution for each problem. This is not the right approach for this project as I do not need to address most of these situations (the aim of the project is to build a rough model and have an idea of what is possible, not to address every particular case). Solutions using corner detection are not chosen either because of the complexity and the content of the images used for the project, a lot of corner direction and hypothesis would be created this way.

The method I use in this algorithm is presented in (Extraction of Buildings from High Resolution Satellite Data, 2001). It is made of different processes. First the tokens are rectified to fit the main tokens orientations in the image which enhances the robustness of the overall algorithm. Then the image is split into homogeneous regions using a binary tree and the remaining tokens. The detected regions are then grouped based on their similarity of texture. This stage uses a region-based method that completes the edge-based method used before.

2.2.1 – Tokens rectification

The goal of this step is to remove noise in the extracted tokens, some of them have an orientation that does not correspond to the main orientations in the image. Tokens with orientations close to the main ones need to be aligned to make the next steps easier, tokens with an orientation completely different are removed because they are likely not to correspond to any building edge.

This step is based on the assumption that buildings are usually aligned with their close neighbours which means that most of the extracted edges should have similar orientations. This is sometimes verified but not always as we will see in part 4.3.6. When the main angles are known, the direction of the extracted edges can be adjusted to fit them perfectly and delete noise. In (Extraction of Buildings from High Resolution Satellite Data, 2001), a kind of Fourier transform is applied on the image. This function derives the angles corresponding to each edge pixel by analyzing a small neighbourhood of the pixel. This angle is then quantized and for each edge pixel with this angle, its gradient magnitude is accumulated in a histogram (which makes pixels with high gradient magnitude more important to detect the main orientations). Local maxima are then extracted from the histogram and give the dominant orientations of the edges in the image. Each edge is then aligned according to the most likely dominant angle.

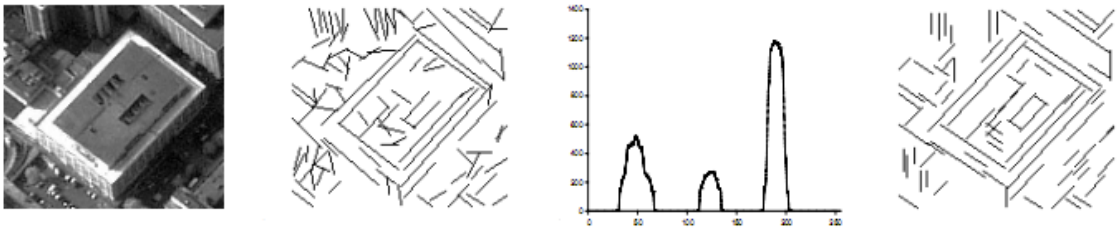


Figure 12 - Left: the original image and extracted edges – Right: histogram of the main orientations of edges and adjusted edges. From (Extraction of Buildings from High Resolution Satellite Data, 2001)

2.2.2 – Binary tree and image splitting

This part of the algorithm is the part where tokens are used to guess possible outlines of buildings. The quality of the previous step is very important here because the more tokens do not correspond to building edges, the worse the result is. The goal of this stage is to over-segment the image, to partition it in small homogeneous areas. Then we group these small polygons according to their similarity into larger ones that correspond to the building roofs. Therefore the main constraint here is to have small and uniform polygons.

In this step the adjusted tokens are used to split the original image in different regions. This splitting is done using a binary tree. The detected edges are sorted in a list according to their likelihood to stand for an edge. The edges with the highest scores are used first to split the image. It is recursively split until there is no more edge in the list. The criterion that decides how tokens are sorted is very important as it chooses how the image is split, hence whether polygons represent whole buildings or fragmented blocks (see part 4.3.5).

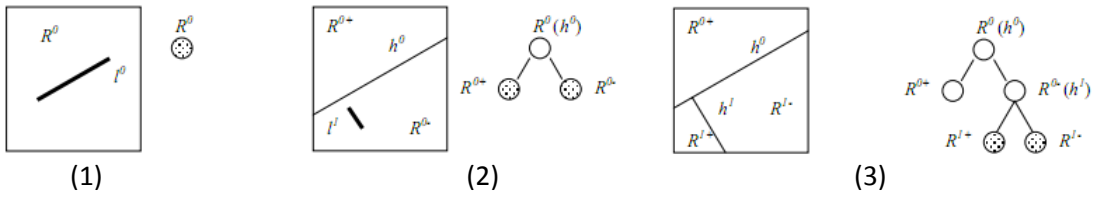


Figure 13 - (1) The original image and the creation of the root. (2) First partition and the corresponding tree. (3) Recursive partitioning. From (Extraction of Buildings from High Resolution Satellite Data, 2001)

This results in a partition of the image in polygons. Some of them correspond to building roofs or part of building roofs while others correspond to vegetation, rivers, parking lots or roads. Next a graph is built in which nodes are unpartitioned regions (called BUS for Unit Building Shape) and arcs are created between adjacent BUSes. This graph is then processed to group similar buses into buildings (next part).

2.2.3 – Buildings grouping

This step is inspired by some region-based segmentation algorithms and especially the region-growing and split and merge techniques described in 2.1.2.

The process described in the previous part can be seen as the splitting part of the split-and-merge algorithm where the image is over-segmented in some small and homogeneous areas. Some of these areas may be the same as their neighbouring areas in the original image, and be parts of the same roof. The next step is to group these areas in bigger, still homogeneous, areas that represent the buildings, this corresponds to the merging step of the previous algorithm. BUSes have well-delineated boundaries which makes it easy to use region-growing algorithm. A graph is built where one node is one BUS and links are created for adjacent polygons. The seed in the algorithm is a BUS. We look for homogeneous neighbouring BUSes thanks to the adjacency graph. The polygons are described by their intensity mean and variance and two regions are homogeneous if they have similar parameters. Homogeneous regions are then merged to create bigger regions.

The goal of this step is to go from an over-segmented image created in the previous step to an image where the segmentation actually fits the buildings. The threshold for the similarity measure has to be chosen carefully as it needs to allow the merging of similar parts of roofs while preventing the merging of roofs and any other area like parking lot, road, and vegetation. This region-based method completes the Boldt algorithm which is and edge detection method. It improves the global robustness of the algorithm.

The three previous steps are sum up in the following diagram:

Figure 14 – (a) The original image with buildings in grey, the background in white and detected tokens in black. (b) The rectified edges. (c) and (d) The splitting of the image in polygons with the binary tree. (e) The grouping of polygons

These three steps lead to a representation of the building roofs by polygons that can have various shapes, convex or concave and any number of vertices which gives flexibility and versatility to the algorithm. The representation is suitable for the selection step that comes next in which polygons that are more likely to stand for buildings are selected.

2.3 – Buildings selection

The building selection process depends on the previous steps and on the representation of constructions or their cues. With U-structures and parallel lines for example, geometrical evidence like parallel edges, corners and shadow corners are used (Nevatia, et al., 1998). With an algorithm that detects corners and edges, a graph that links these cues together to create a building representation can be used (Shiyong, et al., 2009).

My choice of selection criteria to differentiate buildings polygons and other polygons is made on simple observations. I choose five criteria.

- I first used the fact that buildings in an image must have a size bigger than a given threshold depending on the scale of the image, this deletes polygons that are too small to be buildings (trucks for example) or that can be created by the splitting of the image and do not have any meaning.

- The second criterion is the texture of the polygon. Usually, buildings have quite uniform roofs so the standard deviation of the colour of the pixels inside the polygon must be lower than a threshold. This threshold must be chosen carefully because roofs are still a bit textured, no roof is completely uniform. This technique also enables to delete polygons that cover different areas like: vegetation and roads or river and parking lot for example, that we are sure do not belong to the building category.

Figure 15 – We can see in this image from Google Earth the difference of size between buildings and trucks, as well as the fact that buildings have roofs with relatively uniform texture.

- This gives me polygons that are quite likely to be constructions. I then simplify the buildings to be sure that there are not several vertices for the same corner in the original image, or aligned vertices and to have a better representation for the rest of the process. Next, I check that the polygons have more than three vertices because it is quite unlikely to have triangular shaped buildings. I also check that the angle between every side of the polygon is not too small, in which case the polygon is probably not a building.

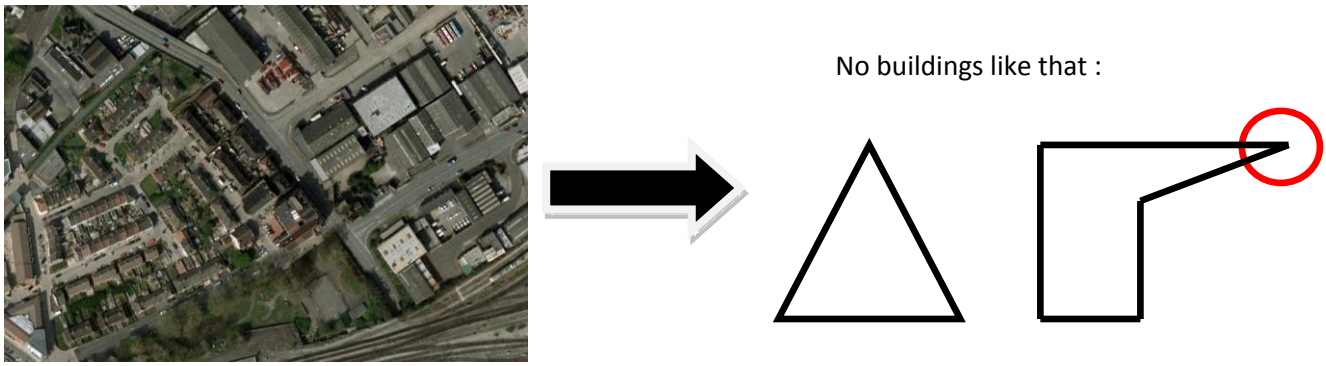


Figure 16 – Satellite Image of an area of Bristol with warehouses and a residential area and description of shapes that are not selected

- The next criterion is the difference between the polygon texture and the texture of the area surrounding it. For a block the difference of colour between these two areas is big, whereas false-positive polygons have a colour similar to the one of their surroundings.

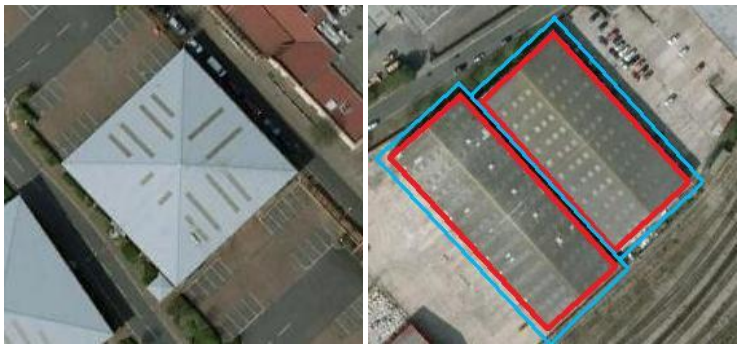


Figure 17 – The buildings usually stand out the background. Even with a ground colour similar to the one of the roof, the cast shadows give a clue that the detected polygon stands for a real building.

- The last criterion of this selection is the shade area that surrounds the building, it has to be bigger than a threshold. Polygons that are not buildings do not cast shadow, so this area is small for them whereas it is big for buildings (Bruce, et al., 1989).

Figure 18 – Buildings with their shade area in green and false buildings without a shade area in red.

These criteria turn out to be well-tailored to industrial areas but have some limitations for other kinds of landscape (see part 4). After applying all these constraints, only the polygons that are most likely to be buildings are left. The next step is to compute their height before displaying them in 3D.

2.4 – Height inference

The previous stages give us the outline of the roof of the buildings in the image. The next thing to do is to guess the heights of the different buildings. Several techniques have been mentioned in the literature to infer buildings heights from aerial or spatial images. Some of them use GIS or DEM or project the shape of hypothetical buildings with different heights and check for cues in the real image that could confirm the height (George Vosselman, 2004) (S. U. Zheltov, 2001). Others use DEM in addition with extracting vertical lines (McGlone, et al., 1994). A lot of techniques deal with stereo images to infer depth (Nevatia, et al., 1989) (George Vosselman, 2004) and papers that deal with single images usually use the sun elevation angle or the satellite viewing angles to infer the height of the buildings (Nevatia, et al., 1998) (Bruce, et al., 1989). All these techniques obviously cannot be applied to our case because we do not have access to other information than the image, however we can still use similar ideas to try to have a rough estimate of the height. The sun elevation angle is usually used to compute the height of a building from the shadow it casts, even without this data we can still have an idea of the height of the building from its shadow compared to other buildings.

2.4.1 – Idea

Shadows have been widely used to find information about buildings in images. We have already seen in the previous part that shadows can help use eliminate fake building hypotheses because buildings cast shadows. We can make the most of shadows to infer the height of the buildings. As explained in (Xiaojing Huang, 2008) the height of a building can be inferred from the length of its shadow with a simple trigonometric relation.

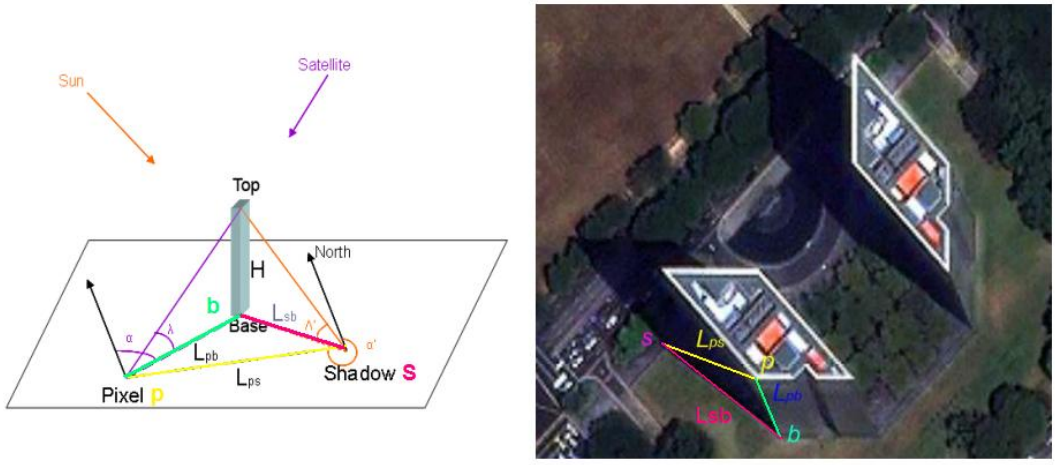


Figure 19 – Lengths and angles corresponding to the image on the right, the plan is the image plan. From (Xiaojing Huang, 2008)

From the previous figure we can infer that: $H = \tan \lambda . L_{pb}$ where λ is the satellite angle and L_{pb} is the segment linking the top p and the base b of the building.
We also have $H = \tan \lambda' . L_{sb}$ where λ' is the sun elevation angle and L_{sb} the distance between the pixel representing the base of the building b and the pixel representing the shadow s of the top of the building. With a little bit more of math we have $H = \alpha . L_{ps}$ where L_{ps} is the distance between the top of the building p and its corresponding shadow point s in the image and α depends on the sun elevation angle and the satellite angle (Xiaojing Huang, 2008).

Of course we do not have all these data but the important thing is that the height of a building is proportional to the length of its shadow in a given image. The previous parts of the algorithm give us

the shape of the building roof, if we manage to find the point corresponding to a roof corner in the shadow we will have L_{ps} in the previous equation. If we have L_1 and L_2 the shadow lengths of two buildings we know that $H_1 = (L_1 / L_2). H_2$. The last thing to do is to make a hypothesis about the height of one building, and the height of every other building are derived from this. To do that we can use statistical method and the scale of the image (see next part) or we can simply test with different value of the height and visually assess the resulting 3D model.

2.4.2 – Shadow extraction

In (Bruce, et al., 1989) the author presents a method to extract shadows from an image, this is used to find the shadow corresponding to a building and especially the point of the shadow corresponding to one or more corners of the extracted buildings.

The first step of the shadow extraction is to smooth the image while preserving the edges. There are several possibilities to do that: use a simple median filter, use conditional averaging filter or symmetrical nearest neighbour filter (Garnica, et al., 2000). Then a global threshold is applied on the image and dark regions are extracted. This global threshold is computed from pixels close to buildings edges which are supposed to be in shadow.

This method has the inconvenient of extracting shadows without any a priori knowledge about the buildings location. In our case we could still use the edge preserving smoothing and threshold but we do not need to apply it on the whole image: we could only apply it to a small neighbourhood of the detected building.

Another idea is to use corner masks as in (Jaynes, et al., 1994). We know that one of the shadow corners is going to be a straight angle which orientation is the same as one of the roof angle. We can simply use a mask that has the orientation of the building corner (mask in black and grey on the image) on a small neighbourhood of each roof corner to find the corresponding shadow corner. It would directly give us the corner location instead of having to detect the edges of the shadow and then the corresponding corners.

The idea with the previous method is to find the straight angle corners corresponding to a roof corner in the image (top-right in figure 20).

Figure 20 – Each area at each building corner is searched for the corresponding angle with the corresponding mask. This angle is found only in the top right area.

Once we have found one or several shadow corners for one building we can deduce the length of the shadow. By doing that for all the detected buildings in the image, we can now compute their shadow length ratio which is also their height ratio.

Figure 21 - Image of an industrial area in Bristol (UK), and the extracted shadow corners for each building. These lengths are then compared to infer the ratio in the buildings heights.

After this last step, we have the outline of the buildings in the image and the height of each building. The last thing to do is to be able to display the resulting 3D model.

2.5 – 3D model creation

Now that we have the outline of the buildings from the original image and their heights, we have to display the resulting 3D model. To do that, I choose to use the VRML language.

VRML is a file format that has been developed for the web (VRML, 2010). It was designed to become a standard in the 3D representation over the Internet. 3D objects are described by their edges and vertices and can be lit and textured as in any other 3D modelling package. Objects in VRML can be manipulated and scenes can be walked through. Besides, VRML files are simple text files which make it very easy to automatically create 3D models of our buildings from the previous step of the algorithm as all we need is to write the vertices coordinates in a text file.

Scenes in VRML can be seen in an Internet browser, it only requires to install a plug-in like Cortona 3DViewer (Cortona3D, 2009) or OpenVRML (OpenVRML, 2009) for example.

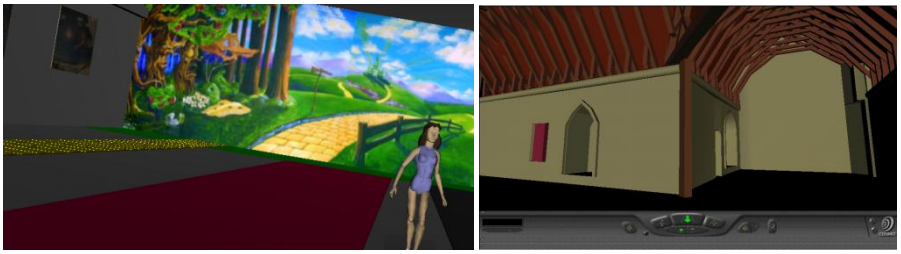


Figure 22 - Examples of worlds created in VRML

The VRML language uses a simple structure called `IndexedFaceSet` that stores the coordinates of the vertices and the set of faces depending on these vertices. It can also store colours and textures. A simple code that displays a cube is shown and the resulting 3D shape is displayed below. In the code, the coordinate of the points are set with the keywords `coord` Coordinate and point, the index in the previous array is used to create the faces that are separated by -1 in `coordIndex`.

```
#VRML V2.0 utf8

Shape {geometry
IndexedFaceSet {
color Color{color[0 1 0, 1 0 0, 1 1 1, 0 0 1 , 1 1 0, 0.5 0.5 0.5 ]}
coord Coordinate { point [1 1 1 , -1 1 1 , -1 -1 1, 1 -1 1 , 1 1 -1
, -1 1 -1 , -1 -1 -1 , 1 -1 -1]}
coordIndex [0 1 2 3 -1 0 3 7 4 -1 0 4 5 1 -1 7 6 5 4 -1 1 5 6 2 -1 6
7 3 2]
colorPerVertex FALSE}
}
```

Figure 23 – Code to display a cube with different colours for each face

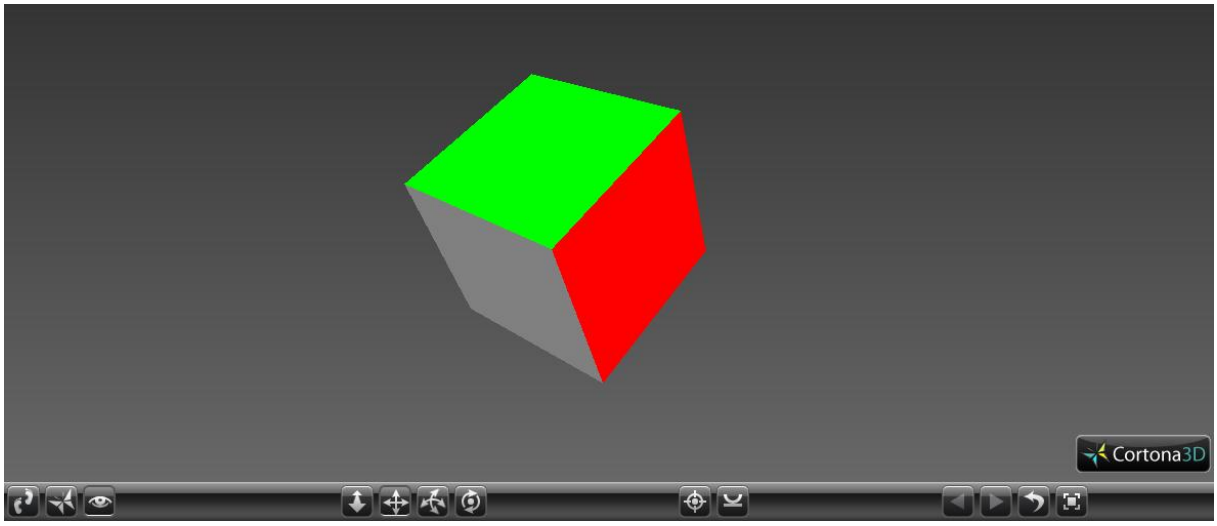


Figure 24 – The cube coded by the previous code and seen with the Cortona plug-in in Firefox

Using VRML seems to be a solution adapted to our needs because it is simple yet efficient and flexible.

The next part describes how this background work and the different techniques described in the literature are used to achieve the aims of the project and explains the details of their implementation.

3 – Design and development

This part of the dissertation tackles the implementation of the methods studied in the previous part. It focuses on the design of the application, on the implementation of each part of the algorithm and on the problems encountered and the solutions found during the development.

3.1 – Design and framework

I chose to carry out this project using the C++ language and the OpenCV library. The OpenCV (Open Source Computer Vision) is “a library of programming functions for real-time computer vision” (OpenCVWiki, 2010). This library gives basic computer vision tools to programmers. Examples are functions to: filter images, compute images histograms, binarize or segment images and more complex ones like Hough transform, movement detection, interest point detection, Delaunay triangulation or Voronoi diagrams. This library is available for C, C++ and Python. I chose to use C++ to use the object-oriented paradigm. I hesitated with an imperative paradigm because usually image processing is done in an imperative way as it can be seen as a sequence of actions that can be described by functions. Also I did not really feel the need to use polymorphism or inheritance at the beginning of the project. On the other hand I had objects : images, edges, points, lines, nodes of graphs, polygons and I needed to create some structures for all these things and some methods/functions to make them interact. Another reason was the size of the project and the need for arrays and structures that could be avoided with classes. The extensibility of the project was also important as I wanted to be able to change methods and parts of the algorithm easily and to have enough flexibility to add new modules to improve the algorithm. Finally I was curious to know more about C++ in an oriented-object paradigm and needed to practise.

In the implementation I use a lot of utility classes to give me tools and describe the objects I use in other, more complex classes. This includes a Colour class that stores three integer values for R, G and B components, a class Point that describes a point in the 2D image with two floats, an Image class that describe an image using the OpenCV element IplImage, a LineEq class that describes a 2D line with three parameters a, b and c for the line equation ax + by + c = 0, and a Token class that describes and edge or token with two points (and other attributes used in different other classes). These basic classes use OpenCV elements and have some basic methods that are largely used throughout all the project like operators (addition, subtraction of points or colour), *display* or *print* methods to show and save the content of objects (images or tokens for example) and results of methods to files, and *draw* methods to display objects on images.

The input of the program is the name of the image to be processed, that can be found in a predefined file. When the program begins to run, the user can choose whether to display and save the files or images created by the algorithm. This is done for each main step of the algorithm: detecting the tokens, creating the buses, grouping the buses, selecting the buses and finding their height. The program can save these files and pictures and always outputs a .vrml file that show the result of the algorithm.

3.2 – The edge detection step

This stage implements the Boldt algorithm described in part 2.1. There are made of three main parts: the initial edge detection to have the first tokens, the graph creation to know which tokens can be linked, and the actual linking and replacement.

3.2.1 - Initial edges detection

The first thing to do is to detect the initial edges. This is done by first loading the input image into the Image class. Then the canny and Hough algorithm are applied to the image using predefined functions of OpenCV. The parameters of these functions are the first parameters that need to be adapted to each image: each scale and every kind of content. The canny algorithm must detect the building edges and try not to detect edges of shadows or vehicles or too many details that would not be useful to have the outline of the buildings. Similarly, the Hough transform must detect edges that are long enough to be significant. The output of the Hough algorithm gives us a sequence of segments that are called tokens from now on. They are stored in an OpenCV structure and are used to build a new object VToken (vector of tokens) that stores the detected tokens.

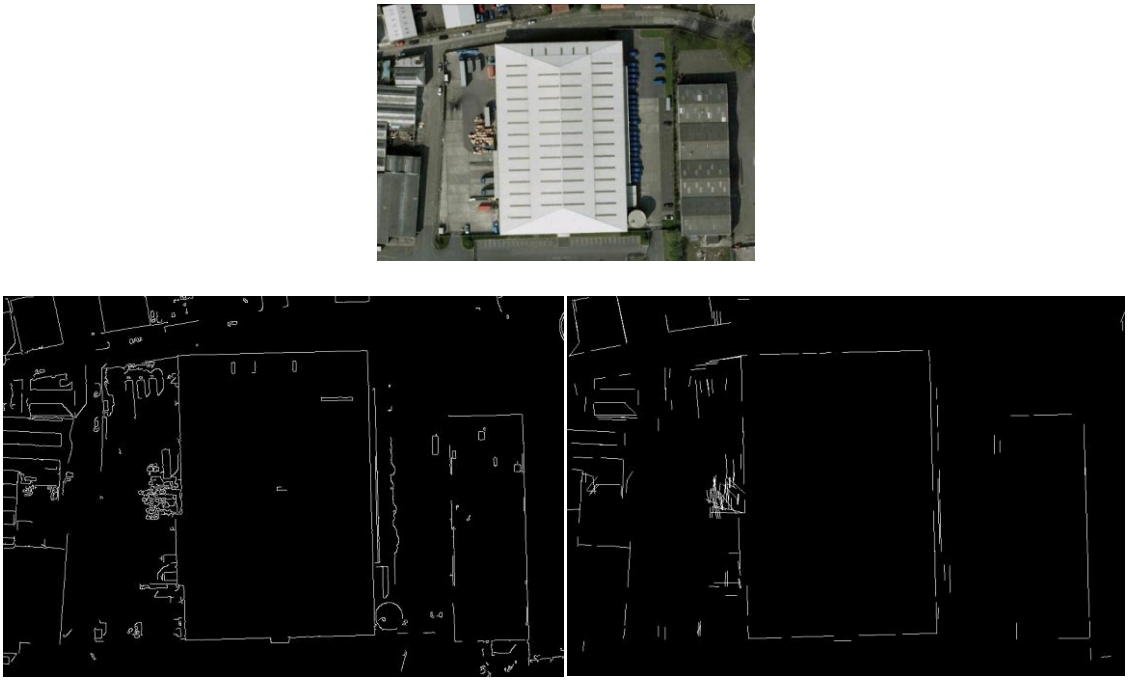


Figure 25 – Top: the original image (image 5). Left: result of the canny filter. Right: Result of the Hough algorithm.

3.2.2 – Graph creation

The next step in the Boldt algorithm is to link the tokens. To do that, a graph is created where the nodes are the tokens and edges are created between tokens that verify some constraints. These constraints are described part 2.1, they are: proximity, collinearity, proximity of end-points, overlap and same contrast. Some methods are added to the Token class to check that these constraints are verified:

- `IsWithinRadius(Token t, int i)` (i is the index of the endpoint of the token)
- `IsCollinearTo(Token t)`
- `EndPointsAreCloseTo(Token t, int i, int j)` (i and j are the indexes of the endpoints of the tokens)

- `OverlapWith(Token t, float threshold)`
- `HasSameContrast(Token t, Image Img)`

The graph is implemented with a class Graph that holds a vector of pointers on Node.
This Node class represents one node of the graph and is made of:

- one Token
- several vectors of Edge class: `vector<Edge> edgesP1toP1` that represent edges between one endpoint of this token and another endpoint of the other token.

The Edge class is made of two pointers on a Node class: the destination node and the origin node.

The first graph is created from the first VToken (vector of Tokens). For each token a node is created, then:

- I check whether it is within the radius of every other token.
- If yes I check whether the other constraints are verified.
- If yes an edge is added between the two tokens.

The situation in the left part of the figure below creates a graph like the one on the right part.

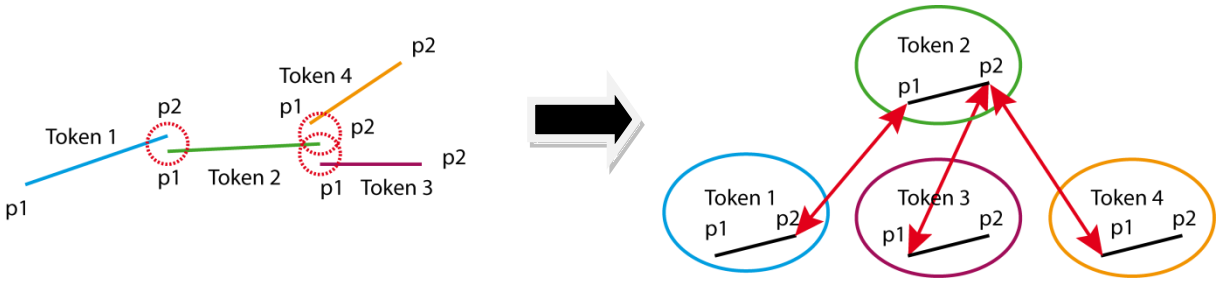


Figure 26 – From a set of tokens in an image (left) to the corresponding graph (right)

3.2.3 – Tokens linking

Now that the graph has been created the goal is to generate a new list of linked tokens. For each node, I first find all the paths which origin is the first endpoint of its token. To do that I use a recursive function that uses a depth-first search through the graph. Each token that is reached is added with the corresponding endpoint (with a class PathDir that stores both pieces of information). The function is called on the neighbouring nodes if they fall in a given radius that is bigger at each iteration. It is called with the endpoint that had not been reached previously.
I then do the same process with the second endpoint. A global variable called `Node::paths` is used to store the paths that are found with this function.

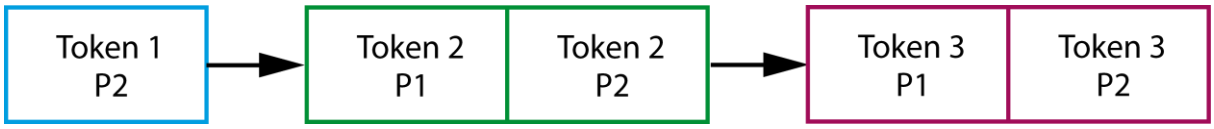
For the situation above (figure 26), let's begin the algorithm with Token 1.
There is no link from endpoint 1 so the path from endpoint 1 only contains its one element:

Path from endpoint 1:

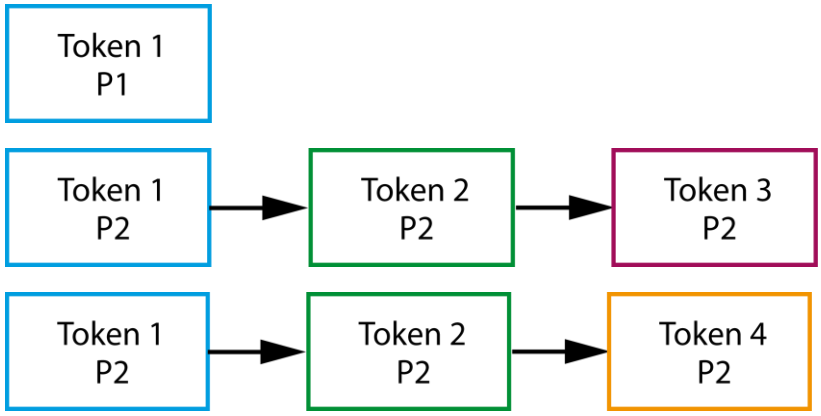
The only neighbour of endpoint 2 is Endpoint 1 of Token 2, so that the beginning of the path is:

Path from endpoint 2:

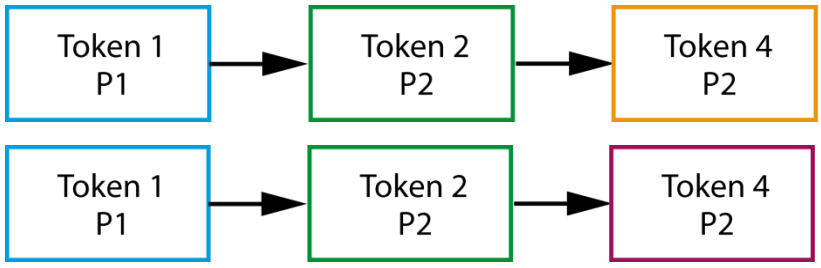
The function is called on the other endpoint of Token 2 which is linked to Point 1 of Token 3 and 4. Hence, the first path is:



The function is called on endpoint 2 of Token 3 which is not linked to any other token. This path stops here. The same thing is done on the Token 4. The resulting paths for token 1 are:



These paths are then put together and give two solutions:



These paths tell us which tokens are replaced. The next step is to compute the replacing token. For each path a straight line is created that fits the given tokens using the least-square method (2010). The straightness of the line compared to the original tokens is then computed. This measure is used in the original Boldt algorithm (Boldt M., 1989):

$$S = \frac{\sum_1^n d_i^2 . l_i}{(n-2).s^2}$$

where n is the number of token endpoints, d_i is the perpendicular distance from the endpoint to the approximated line, l_i is the length of the token which this endpoint belongs to, and s is the distance between the first endpoint of the first token and the last endpoint of the last token.

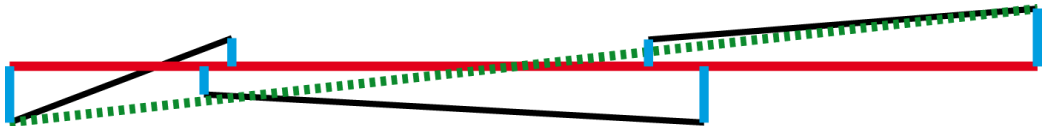


Figure 27 – Example of original tokens (in black) being approximated by a new token (in red)

Above, the original three tokens are in black. The approximated line is the horizontal thick red line. The d_i are the small vertical segments in blue, the l_i are simply the lengths of the tokens, s is the length of the green dotted line and n = 3.
Once the straightness has been computed for each path, the optimum one (the one with the smallest S) is compared to a threshold, if it is less than this threshold the tokens are replaced by the approximated line, if it is bigger the tokens are not changed.

After that, there can be several times the same token in the list of new, linked tokens, so duplicates are removed as well as tokens that are too small (and have not been linked). A new graph is created as in part 3.2.2 using a bigger radius to select the neighbouring nodes to increase the scale of the linking. Then I iterate on the resulting graph from part 3.2.3 to find the new paths and approximate them with a line, check their straightness and replace the old tokens by ones. Three to five iterations are usually enough to converge to a state where no improvements can be achieved.

3.2.4 – Problems

The steps at the beginning of this part that use canny algorithm and Hough transform and the creation of the graph do not require expensive computations but the replacement of the original tokens by new ones can be very expensive. The more tokens there are, the longer this step takes, because the number of paths between tokens increases hugely with the initial number of tokens. Most of the tokens have a very small number of corresponding paths and their replacement tokens are easily found but a few of them have huge numbers of paths that take a very long time to process. This happens when a lot of tokens, usually small ones are in the same area, and this is caused by some areas of images like trucks, cars or big boxes that have straight edges and are much contrasted.

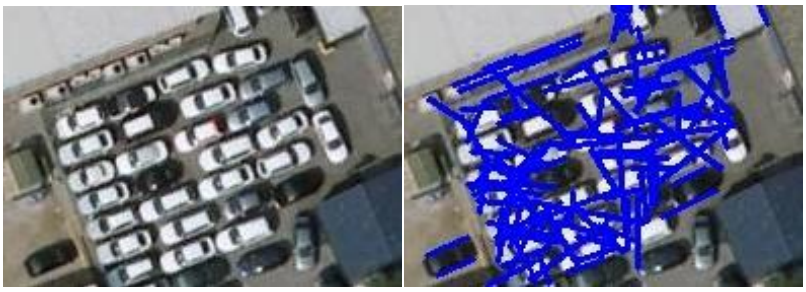


Figure 28 – Original image of cars and corresponding detected tokens

A way to reduce this processing time is to adjust the previous parameters of Canny and Hough algorithm to have less tokens, however this usually reduces the accuracy of the rest of the algorithm, this is discussed in more details in part 4.

Once this step has been done we have tokens that represent the edges of the buildings as well as some other parts of the urban environment. We can begin the building hypothesis creation from these extracted edges.

3.3 – Building hypothesis creation

This step is the one where we turn the tokens into actual building hypotheses. As explained in part 2.2, there are three main processes, the first one is to rectify the tokens that we got from the previous part, the second one is to segment the image into areas using these tokens and the last one is to group the previous polygons to give meanings to the resulting areas.

3.3.1 – Tokens rectification

The goal of this step is to align the tokens on the main directions of the image.

The first thing to do is to compute a histogram of the orientations of the tokens. I quantized the tokens angle with a step of 0.005 radians. Tokens that have a large contrast and a large length contribute more to the histogram than other tokens. An example of detected tokens and the corresponding histogram are shown below:

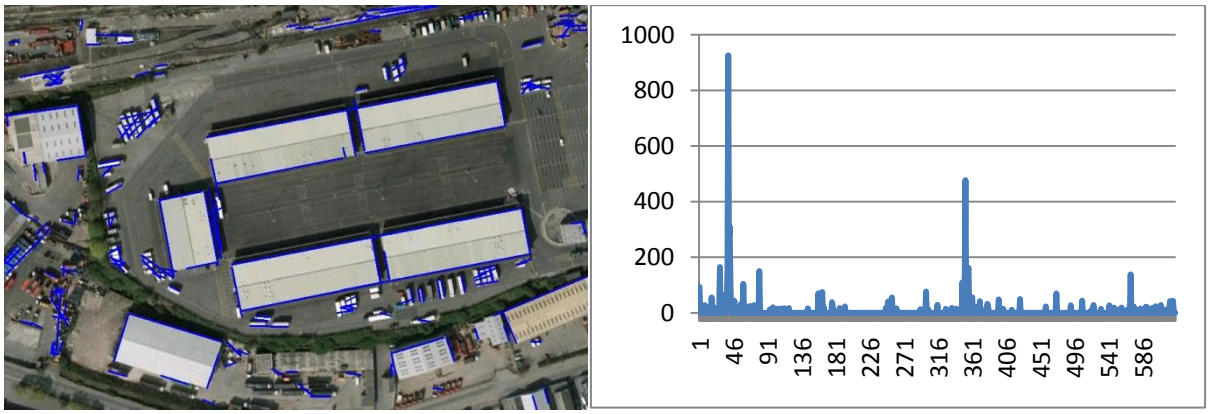


Figure 29 – Left: The original image (image 6) with the detected tokens. Right: The extracted orientations.

The two biggest peaks on the histogram are at 0.19 radians and 1.75 radians, that is to say almost $\pi / 2$ radians away. They correspond to the long and small edges of the five buildings in the middle. The `Rectify` function first detects the three main peaks by looking for the max, then deletes it, and looks for the next max and so on. Then, for each token, I compute the angle difference between the token and the closest optimal orientation. The idea is then to compare this angle difference with a threshold: if it is smaller than the threshold I rectify it, otherwise I discard it. The rectification in itself is shown on figure 30.

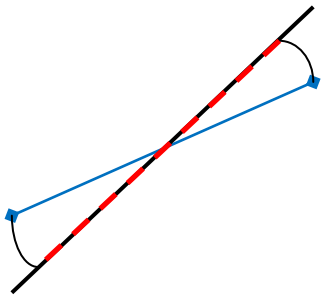


Figure 30 – The original token in blue is rectified to fit the optimal orientation in black. The token simply rotates around its mid-point; it keeps its original length. The result is the dotted red line.

However, the problem with using only one threshold is that tokens that have an orientation difference slightly bigger than this threshold are either discarded (if the threshold is small, see figure 31) or rectified but without fitting the building edge (if the threshold is big, see figure 31 too). To add flexibility to the rectification I add a second threshold. If the angle difference is smaller than the smallest threshold (which is 0.075 radians ≈ 4.3°) the token is rectified as shown figure 30. If the angle difference is between the first and the second threshold (0.30 radians ≈ 17°), the token is kept unchanged. If the angle difference is more than this threshold, the token is discarded.

The use of two thresholds overcomes the previous limitation and gives a good result by aligning the tokens and deleting unnecessary ones. More limitations are discussed in part 4.3.6.

Figure 31 – Top left: the original tokens.
Top right: Tokens rectified with one small threshold (0.075 rad), a lot of tokens are discarded because their orientation is slightly different from the main one, especially the buildings at the top.
Bottom left: Tokens rectified with a large threshold (0.30 rad), the tokens are kept but do not fit the buildings any more.
Bottom right: with two thresholds the main tokens are rectified, the tokens of the buildings at the top of the image are kept and a lot of small tokens with random orientations are removed (bottom left of the original image)

3.3.2 – Tokens sorting

Now that we have rectified the tokens and kept only the significant ones, the splitting of the image can be carried out. This is a recursive process (explained in 2.2.2) where we use a list of tokens to recursively split the image. We use the first token of the list to divide the image in two parts, then for each part, we select from the previous list of tokens a new group of tokens that "lie" in this part. The first token is used to split the part in two and each division undergoes the same process until there are no more tokens to split the image.

The first problem to solve is to know how to sort the tokens to have a good partition of the image. The point here is that tokens are used to split the whole area they lie on even if they are very small, which means that they cannot only delineate the building they are an edge of but also split other buildings, and we try to avoid that as much as possible. Below is an example of an image split with tokens that are sorted according to different criteria.

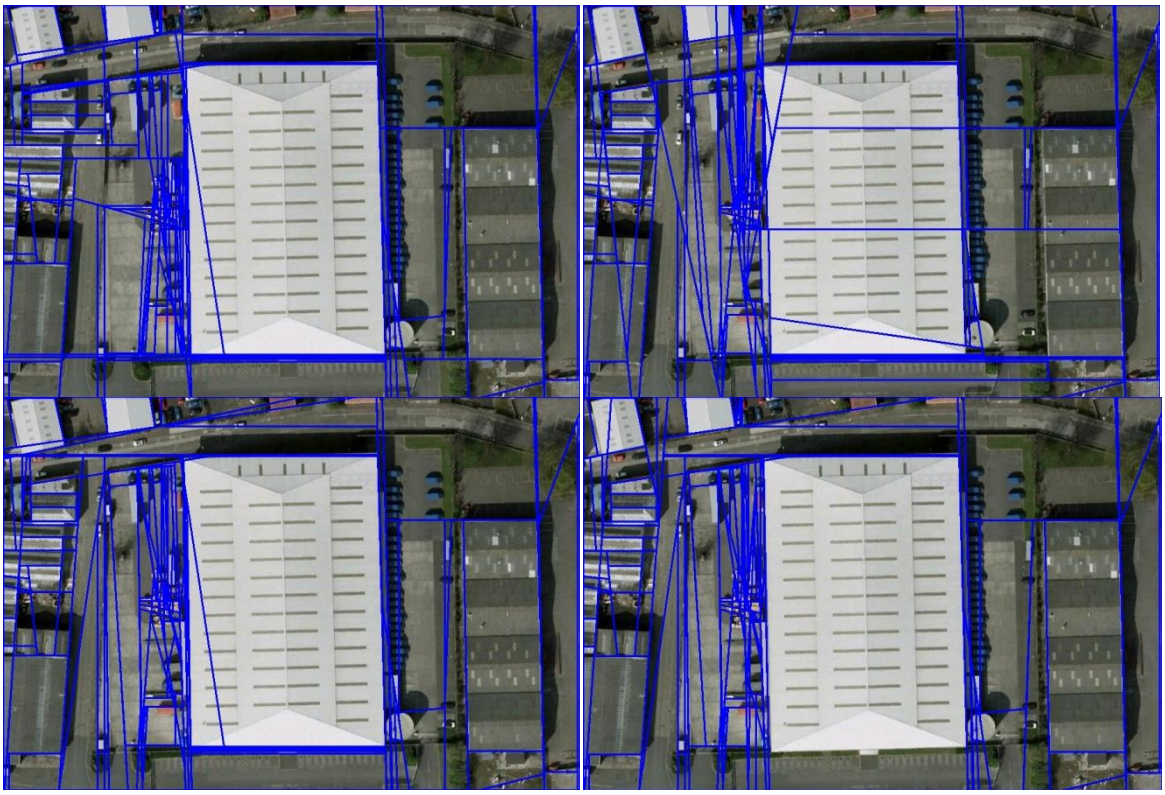


Figure 32 – Top left: no sorting
Top right: sorting according to the token's contrast
Bottom left: sorting according to the length
Bottom right: sorting according to the product length x contrast

Having tried different kind of sorting on different images I decided to sort the tokens by length because it gives better results. The problem of the choice of the sorting criterion is discussed in more details in part 4.3.5. Once we have sorted the tokens we can begin splitting the image into small polygons.

3.3.3 – Splitting the image

To split the image in meaningful areas I first create a binary tree of areas of the image. The theory is explained in part 2.2.2. The classes used to describe the binary tree are a NodeBSP (Node of Binary Split Tree) and EdgeBSP classes to describe the arcs.

A node in the tree holds a vector of points that stands for the polygon. The points are given in an anti-clockwise way, they can be as numerous as needed, which gives flexibility to the algorithm and allows us to detect a wide range of shapes. Both concave and convex polygons can be detected. The other attributes of this class are three EdgeBSP to the "right" son of this node, the "left" son of this node and to the "father" of this node (which could be used to create the adjacency graph in step 4). The other attributes are a number for this node and a list of EdgeBSP on adjacent nodes. The EdgeBSP class is made of two pointers: one on the origin node (this one) and one to the destination node.

The NodeBSP is different from the Node class used in the Graph creation to link tokens because it contains a set of points instead of a token and only three arcs instead of a list of arcs from one or the other endpoint. However the Edge class and EdgeBSP class are the same except that they do not link the same class. Here I could have used a superclass of Nodes and only one class of edges.

The first thing to do to recursively split the image is to create the root of the tree. It is a rectangle of the size of the original image. Once this node has been created we can call the recursive function:

```
void NodeBSP::createBSPTree(VToken* v, Image img)
```

with `v` being the original list of sorted tokens and `img` the original image. This function calls the function

```
this->findSons(Token)
```

which splits the current node in two nodes separated by the token taken as parameter.

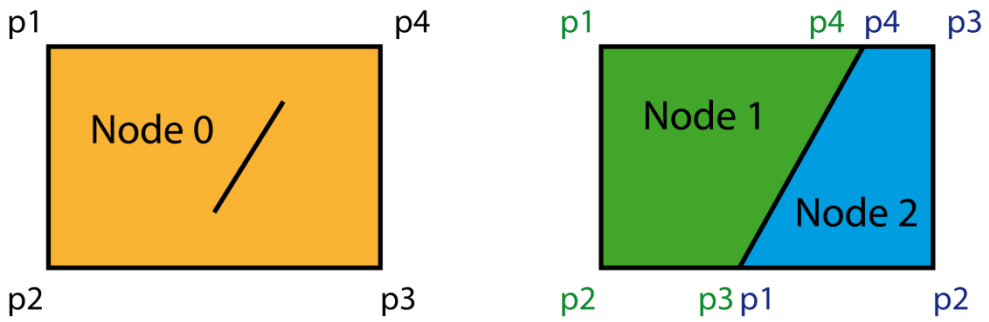


Figure 33 – Left: original node made of four points with the token that splits it. Right: the two nodes created from the previous situation.

The previous function makes the left and right pointers of this node point on these two new nodes. Then the token used to split the node is removed from the list and two new lists are created from the previous one. For each node, the corresponding list of tokens is made of tokens that have at least one endpoint inside the node or that cross the node (see below).

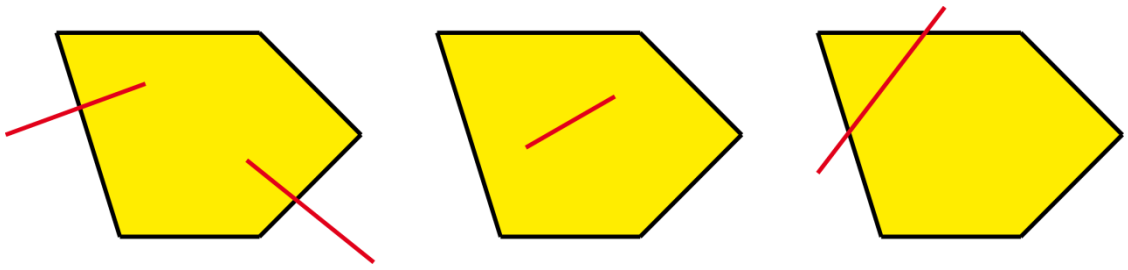


Figure 34 – Situations in which tokens (in red) are put in the node's list: at least one endpoint inside the polygon or a part of the token inside the polygon.

Each newly created node calls the function `createBSPTree` with its corresponding list of tokens. It ends when every token has been used to split the image. What we have at the end is a root node with a left and right son which also have right and left son and so on. Which one is the right or the left son does not matter. Figure 35 shows the different steps of the splitting of the original image into polygons and the corresponding binary tree.

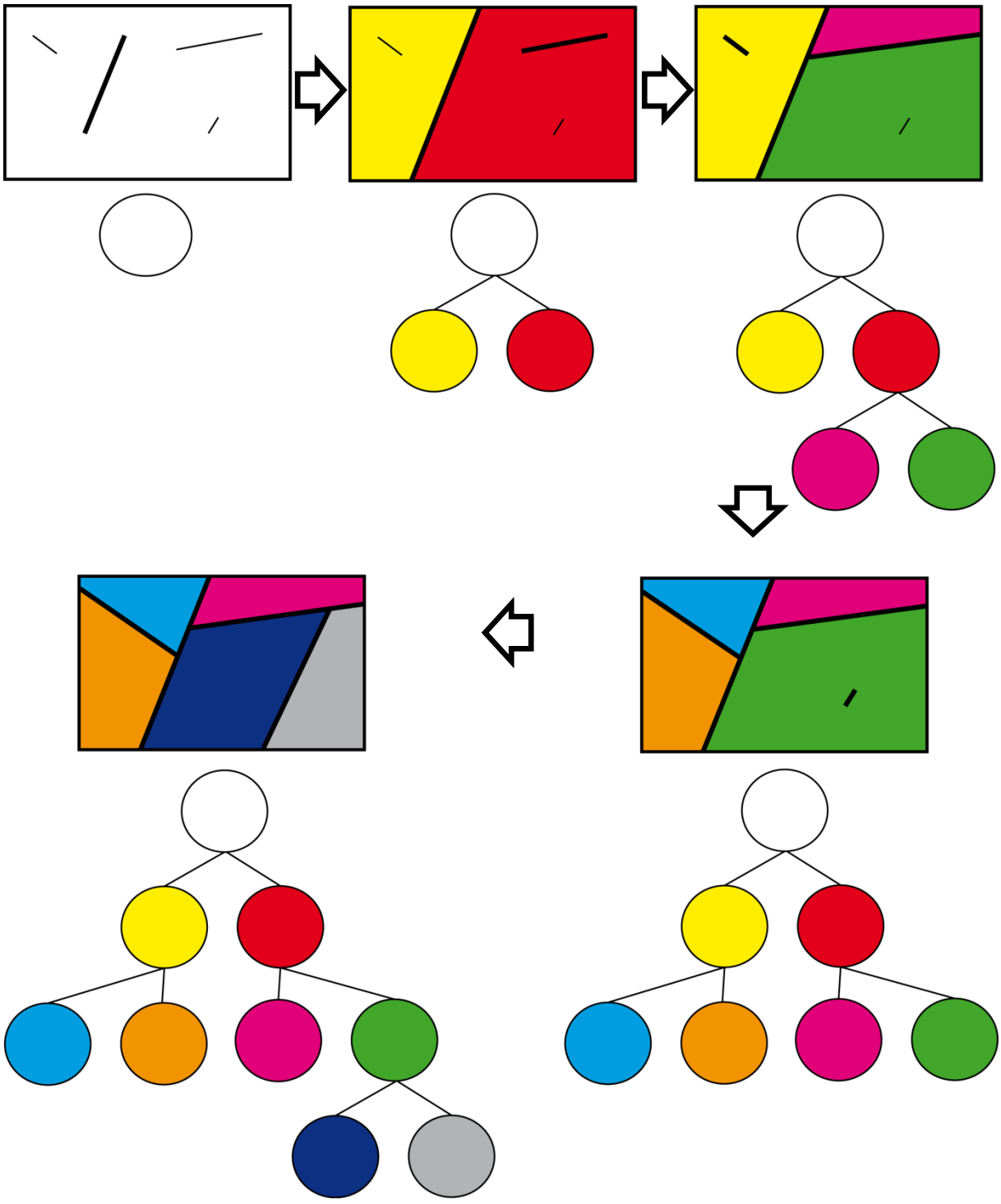


Figure 35 – This diagram shows the steps of the splitting of the image with the corresponding binary tree at each step. The original image is the white one at the top left of the diagram and the longest token is in bold. It is split in two polygons (yellow and red) which are then split in two polygons (pink and green) in the third image. This is done until there are no more polygons to split the image.

We have this binary tree but we still cannot access each node individually without traversing the tree. The next step is to turn the tree into a set of polygons that can be easily accessed and processed. Each polygon is called a BUS (Building Unit Shape) and the class used to store them is called BUSes. This class stores a vector of pointers on NodeBSP. The BUSes are the leaves of the binary tree (see diagram below, the leaves are in bold), they are nodes that could not be split again

and hence are homogeneous. To be able to process them individually, a function traverses the tree and if the node is a leaf, adds it to the BUSes object.

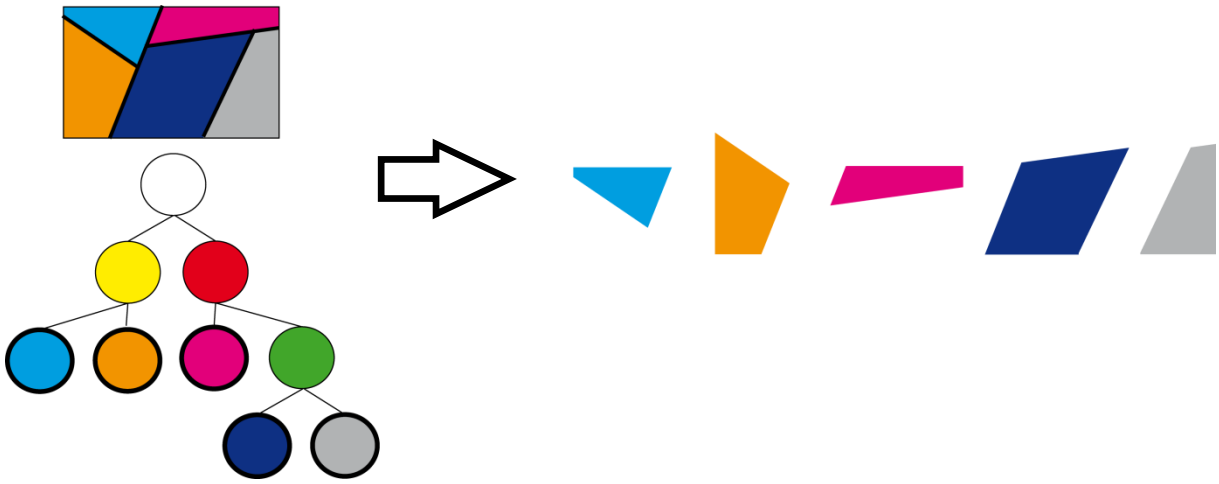


Figure 36 – Left: the split image and the corresponding binary tree. The final polygons are the leaves of the tree which outline is thicker than the other nodes. The corresponding extracted polygons are listed on the left.

We now have a structure that can be used in the next stages of the algorithm.

3.3.4 – Grouping the polygons

As we have seen previously, we have selected a lot of tokens and some of them do not stand for building edges. The previous step of the algorithm can be seen as an over-segmentation of the image in small areas (see 2.2.3) and the polygons need to be merged to create bigger areas that correspond to building roofs. The first thing to do for that purpose is to know the neighbouring polygons of each BUS. Then the texture of each polygon is compared to the one of the BUSes it is adjacent to and the polygons are merged if their textures are similar.

To know which polygons are adjacent to which one, we create an adjacency graph. It is implemented by using a list of edges in the NodeBSP class (called Adj) that stores edges to all the neighbouring nodes of the current node: `std::vector<EdgeBSP> Adj`
The function `createAdjGraph()` is called on the BUSes object to create this adjacency graph. It takes each BUS and checks if it shares an edge (with the function `ShareEdge` called on the current BUS) with the BUSes that have not been processed yet. If it does, an edge is added to the vector Adj of the current node, which origin is the current node and destination is the other BUS. Another edge is added to the vector Adj of the other BUS that goes from the other BUS to the current node. For the previous split image (figure 37) the adjacency graph would be :

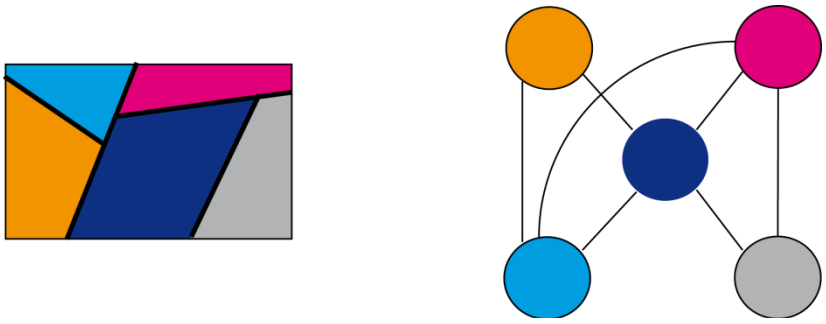


Figure 37 – Left: The detected BUSes in the original image – Right: the corresponding adjacency graph

After the graph has been created the grouping can begin. The function

```
BUSes BUSes::group(Image img, int diff_avg, int diff_dev)
```

is called on the original set of polygons. The parameters are the original image and the thresholds to decide if the polygons are similar or not. The function iterates through every BUS and checks if it has the same texture as its neighbours.

For that purpose it calls the function `IsSimilarTo` that computes the textures of both nodes. The first thing to do is to find which pixels belong to the polygon. To reduce the complexity of this computation I first compute a bounding box of the given polygon. Then for every pixel in this bounding box I check if it lies inside the polygon. For this I use the function `hasPoint(Point)` that for the given point computes the number of intersection between a semi-line having this point as an endpoint and the polygon edges (see figure below).

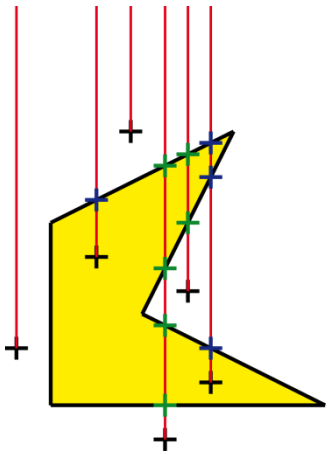


Figure 38 – For every point (black cross) the semi-line is in red (it works with any semi-line) If the point is inside the polygon the number of intersection is odd (blue intersection points). If the point lies outside the polygon the number of intersection points is even (green intersection points). The corners and boundaries are handled properly by my implementation.

Thus, for every pixel I know if they lie inside the polygon. If they do I use their colour to compute the average colour of the polygon and the standard deviation of the colour distribution.

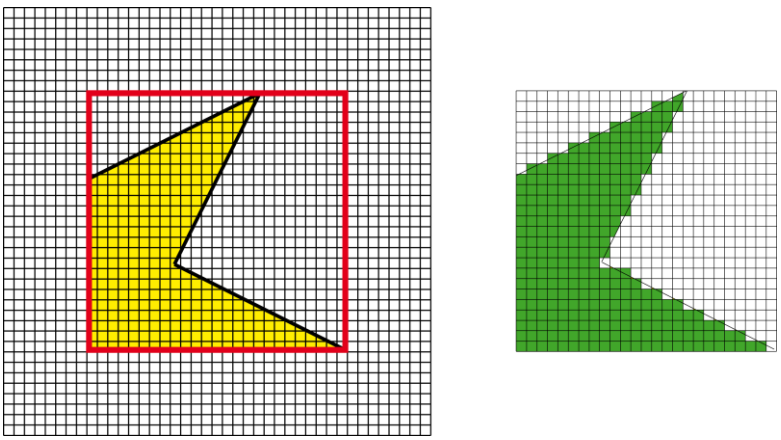


Figure 39 – For each polygon I first compute a bounding box (in red) then I checks if every pixel in that bounding box lies inside the polygon by the method showed figure 38. They are in green on the right.

The texture is stored in a class called `Texture` that holds the average pixel colour (three components R, G and B that create an object Colour) and the standard deviation of the pixel colour distribution (also a Colour object). These two parameters are a simple way to describe a texture. There are a lot of more complicated and powerful ways to describe it but this one is easy to compute and is accurate enough to give me an idea of the homogeneity of the polygons (sse part 4 for the limitations). Examples below show an idea of the difference of texture for different polygons.

Figure 40 – Top left: Average RGB colour Mu (93, 107, 109) and standard deviation Sigma (23, 22, 23): the polygon is quite homogeneous. Top right Mu (70, 79, 78) Sigma (36, 39, 40): the polygon is darker and more textured. Bottom left: Mu (171, 168, 156) Sigma (14, 13, 13). Bottom right: Mu (171, 168, 157) Sigma (23, 21, 20) These two polygons have similar textures: same average colour and similar standard deviation so they are going to be grouped

If the difference of texture (difference of average colour and of standard deviation) is less than a threshold, the two polygons are considered similar and can be grouped. The merging of two nodes is done by taking the two sets of points of each node, finding the shared edge and creating the resulting node from that. The next figure shows and explains the details of this process.

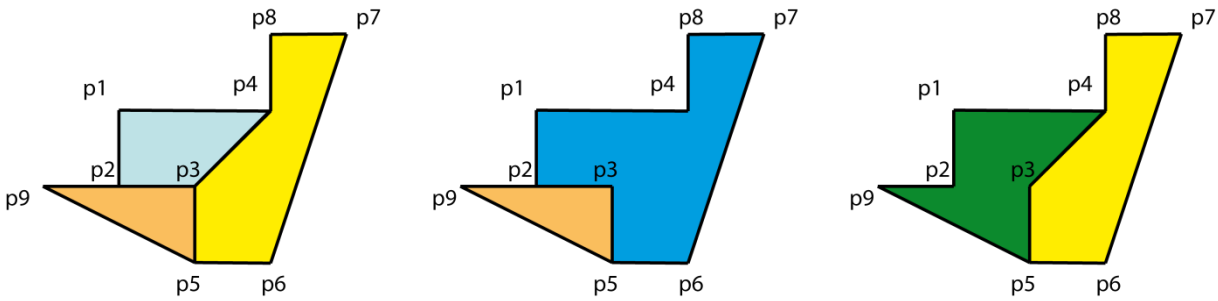


Figure 41 – Left: the original polygons. Middle: result of the merging of the blue and the yellow polygons. Right: Merging of the yellow and the orange one.

In figure 41, to merge the blue and the yellow polygons the first points of the blue polygons are added to a list until we find the first point of the shared edge (p3 p4), the list is: p1 p2 p3, then the points of the second polygon are added (always in the antic clockwise direction) p5 p6 p7 p8 p4 and the remaining points of the first polygon if there are any. The resulting polygon is p1 p2 p3 p5 p6 p7 p8 p4. The same thing is done for the yellow and the orange polygon which become p1 p2 p9 p5 p3 p4.

Polygons made of any numbers of points can be merged this way. The merging is iterative, once two polygons are merged, the new polygon is compared to its neighbours to see if they can be merged. For example in the previous situation, if all three polygons are similar, the resulting polygon would be as described in the next figure.

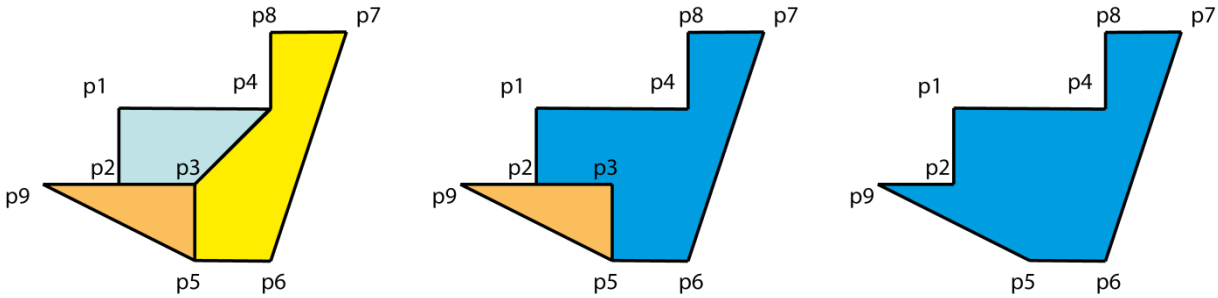


Figure 42 – The different iterations of the BUS merging step.

The grouping step can be iterated several times if necessary, until no polygons can be grouped anymore. This stage results in the grouping of the polygons in bigger ones that fit better the building roofs.

The next step is an intermediary process that simplifies the resulting BUSes before selecting the polygons that are more likely to be building roofs.

3.3.5 – Buildings simplification

As we have seen in the previous examples, the resulting polygons can be concave or convex and have some complicated shapes. This is good for the flexibility of the algorithm, however the merging of the polygons can create situations where some points are added even if they are not needed to have the global shape of the outline of the roof. The simplification step simply deletes these useless points to have a simplified outline. The next figure illustrates this situation.

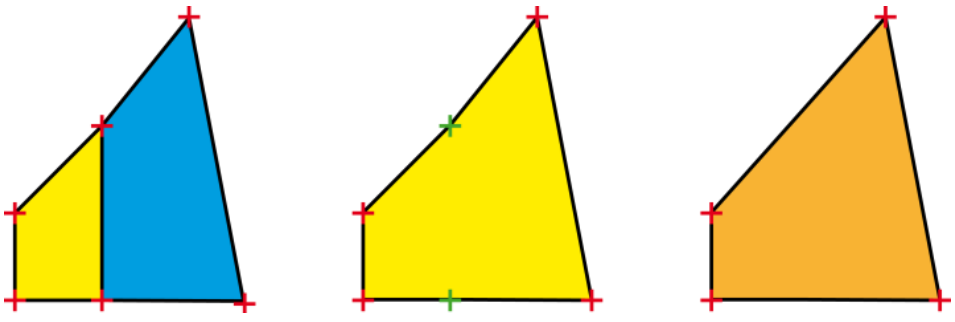


Figure 43 – Left: the two original BUSes to merge. Middle: the merged polygons with six points. Right: the simplified polygons where the green points (useless ones) in the previous image have been deleted.

3.4 – Buildings selection

We now have a partition of the image in polygons that can potentially be buildings. We have to find criteria that help us decide whether a polygon is actually a building or not. In order to do that I used some simple criteria: the homogeneity of the roof texture, the size of the building, the difference of colour between the building roof and its surroundings and the size of the shadow cast by the building.

3.4.1 – Removal of triangular and narrow polygons

The first selection step is to remove the polygons that cannot be buildings, that is to say polygons with only three points: buildings are rarely triangular, and buildings with at least one very small (see figure 16), because these polygons are likely to have been created in the splitting process without having any meaning for our research.

3.4.2 – Texture uniformity

The first criterion relies on the fact that building roofs have a uniform texture. The colour variation within the same roof is small and this is what is measured by the standard deviation of the pixels colour within a polygon. The first thing to do is to compute the texture of each polygon (see 3.3.4) and to get its standard deviation. If it is smaller than a given threshold, the BUS is likely to be a building as its roof is homogeneous. An example can be seen figure 40, where the images at the top show the difference of standard deviation between a roof and an area made of a road and a pavement.

3.4.3 – Building size

The next thing to compute is the size of each polygon. I perform the same steps as in 3.3.4 to compute the polygon's texture, except this time I only count the number of pixels that lie inside the polygon. Once I know the area of the polygon, I compare it to a given value. If it is bigger than this threshold, I keep it as it is likely to be a building. This step enables us to get rid of the small polygons that have a uniform texture but are too small to be buildings like trucks or containers for example.

3.4.4 – Colour difference between the building and its surroundings

Another property of the buildings in a satellite image is that building roofs usually stand out the ground (see figure below)

Figure 44 – Both in an industrial area and in a residential area, all the buildings can be easily detected by the human eye because of their difference of colour with their surroundings, be it parking lots, vegetation or roads and pavements.

This is due to the fact that it is very unlikely that a building roof have exactly the same colour as the ground around it, and even if it does, the shadow cast by the building creates a darker area around the roof outline that is used to evaluate the colour difference.

In order to assess this colour difference, we have to choose what we call the surroundings of the building. I use a small area around the block, of approximately the same shape, which edges lie "close" to the edges of the buildings. For this I create a box around the building. This is done by going through each of its corners and finding what kind of angle the edges of the construction create. According to the edges orientation, a new point is created that corresponds to the corner of the new box. The criteria to determine where the new corner is located are: which endpoints of the tokens are shared and the orientation of the tokens (see next figures for more explanations).

Figure 45 – The corner (red cross) created by the two black tokens gives a new point (in blue) found by subtracting the distance d to the corner abscissa and ordinate on the left. On the right, the distance d is subtracted to the abscissa and added to the ordinate because the orientation of Token 1 is different.

I go through every possible situation in terms of corner orientation to be able to create a new point for each type of corner of my polygons.

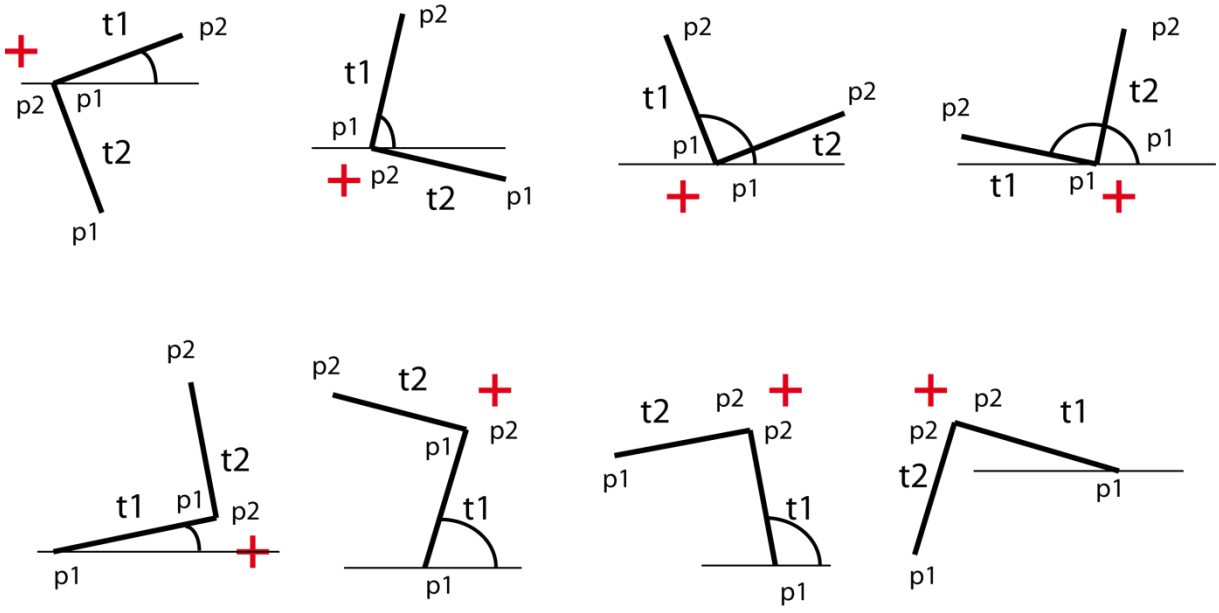


Figure 46 – Possible situations depending on the shared edge points (p1 or p2 of each token) and the first token (t1) orientation. The new corner point is in red.

By creating a new point for every corner of the polygon I create a box around the block (figure 47). Then I compute the texture of the box with the same technique as in 3.4.3 using only the points that lie inside the surrounding box and outside the polygon and I use their colour to get the box texture.

I then compute the texture of the original polygon and compare them. If the difference is bigger than a threshold I keep the polygon because it is likely to stand for a roof building.

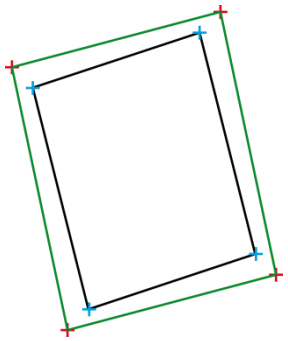


Figure 47 – The original polygon is in black with blue corner points. The new polygon is in green with the new points in red.

3.4.5 – Size of cast shadow

We now have discarded all the polygons that were too small : trucks and containers, all the polygons that did not have a uniform texture, all the polygons that were not different enough from their surroundings, but there can still be some parking lots, or vegetation areas that are big, uniform and different from what surrounds them. To remove them we have to use the fundamental difference between these areas and the buildings: buildings are high and cast shadows. This step is not a shadow detection step, it is just a basic step that gives an idea of the "amount of shade" around the polygon.

The idea is the same as for the previous criterion: I create a box around the polygon and count the number of dark pixels inside this box (and outside the polygon). The threshold for the "dark" pixel is set to a luminance of 35. If the ratio:

```
number of dark pixels in the box / total number of pixels in the box
```

is bigger than a threshold, there is probably a shadow cast by this BUS and the polygon is considered as a building roof.

3.4.6 – Conclusion

All these selection criteria are successively applied to the original set of polygons. Before this step we had a large number of buildings that partitioned the original image, after this selection we only have a few polygons that fit the buildings.

We now have the final outlines of the constructions. Most of the polygons are at the right location while a small number of them do not represent actual buildings and while some buildings are missing. These problems come from the selection criteria that are not adapted to every picture and the difficulty to find the right thresholds depending on the image and its content. They also come from previous parts of the algorithm that have created errors or approximations that could not be rectified. The results are detailed in the fourth section.

The final step of the algorithm is to find the height of each selected building. This is described next.

3.5 – Height inference

We now have the buildings outlines and need to know their heights to build the 3D model. As I explained in 2.4.2 we want to detect the shadow corner corresponding to every polygon corner. This is made of several parts: first processing the image to enhance the shadows, then create a filter (mask) to detect the corners, rotate this mask to make it fit the detected corner, next convolve the mask with a small area around the polygon's corner and finally find the location of the best response of the filter.

3.5.1 – Shadows enhancement

This stage is designed to detect the shadow corner more easily. The idea here is to have a bigger difference of luminance between the shade area and the area it is projected on, than in the grayscale version of the original image. This is because the shadow is usually projected on an asphalt road or on some vegetation which can be quite dark.

I first tried to detect the shadows' corners on the original grayscale image, but the corners were not accurately detected. One solution is to pre-process the image to have shadows that stands out more on the ground, and have a bigger luminance difference between the shadow and the ground to have a better response to the filter. The first idea is then to simply increase the contrast of the image by equalizing the image's histogram. The result is shown in the figure below.

Figure 48 – Left: the original grayscale image with the detected shadow corners in white (explained in the next parts). Right: the image after the histogram equalization and the newly detected shadow corners.

The difference in the detected corners is not striking because the contrast enhancement focuses on making details more visible and on increasing the colour difference in places that are not always important for the shadow detection like the inside of the roof or the road. What I want to achieve is having a big colour difference at the limit of the shadow. The next solution is then to use a binary threshold on the image and to keep the shadows in black and everything else in white. This way there would be no doubt about where the shadow limit is. I tried with several thresholding methods and different thresholds as shown on next figure. The result is a binary image where shadows are dark and their limits with the ground are very clear.

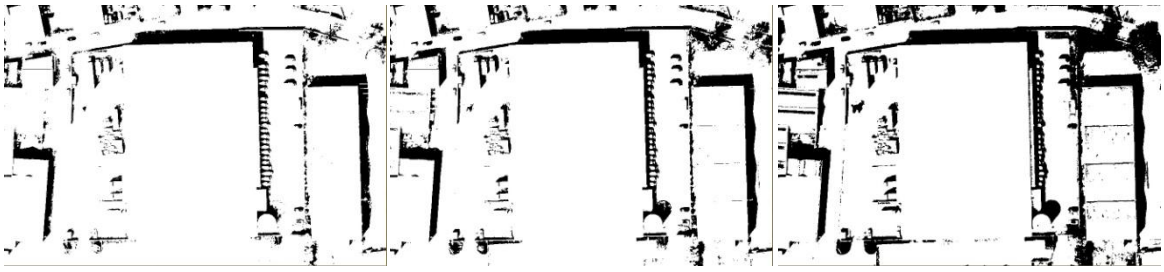


Figure 49 – The same image binarized with different thresholds: 35, 45 and 65. I then decided to keep a threshold of 65 because it keeps more details.

The next stage is to create the filter to detect these shadow corners.

3.5.2 – Filter creation to detect corners

This method is based on the one described in 2.1.3 that can be used to detect the buildings outlines, it has been adapted to detect shadow corners. In order to do that we have to detect shapes with a straight angle in the shadow. The original shape of such a filter to convolve with the image is described below.

255	255	255	255	255
255	255	255	255	255
255	255	0	0	0
255	255	0	0	0
255	255	0	0	0

-9	-9	-9	-9	-9
-9	-9	-9	-9	-9
-9	-9	16	16	16
-9	-9	16	16	16
-9	-9	16	16	16

Figure 50 – Left: a shadow corner (with axis-aligned edges) on a binary image and the corresponding pixel values. Right: the corresponding filter.

The convolution of the previous filter with the image on the left would give a negative result with a large absolute value:

$$-9 \times 16 \times 255 + 16 \times 9 \times 0 = -36720$$

In a uniform area with pixel's luminance *a*, the convolution gives:

$$-9 \times 16 \times a + 16 \times 9 \times a = 0$$

I tried several filter sizes: 3 x 3 (which is too sensitive to noise), 5 x 5, 7 x 7 and even 11 x 11 and decided to keep a mask of size 5 x 5.

3.5.3 – Mask rotation

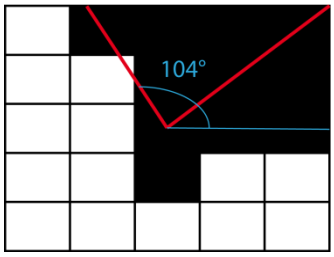
The previous mask can only be used to detect one type out of four axis-aligned straights corners. The goal of this step is be able to detect any kind of straight angle.

The information needed to do that is the angle made by the first edge and the horizontal axis. Once I know that I compute the rotation matrix which center is the center of the mask and which angle is the edge orientation that we have just found. Then I rotate the previous mask using the rotation matrix to have the new one. Everything is done using existing OpenCV functions:

```
Rot = cv::getRotationMatrix2D(center, angle, 1.0);
```

```
cv::warpAffine(M,Dst,Rot,M.size(),cv::INTER_LINEAR,cv::BORDER_CONSTANT);
```

This gives me a new mask to convolve with the image.
The example below shows the resulting mask for a given rotation.



-5	3	16	16	9
-9	-2	16	16	13
-9	-8	16	10	4
-7	-9	8	-9	-9
-5	-9	-9	-7	-5

Figure 51 – The rotation angle is 104°. Left: The corner shadow we are looking for. Right: The new mask.

Now that we have the final filter we need to convolve it with the image.

3.5.4 – Mask convolution

I convolve the mask with the image in a box around the polygon's corner. The size of the box depends on the roof size which can be seen as a way to take into account the link between size of the roof and height of the building: if a building's roof is big, it is more likely to be high and the shadow is then be cast further away from the roof's corner. The chosen relation is:
Size of the box: d = $\sqrt[3]{area}$ as it fits pretty well on the images I have chosen.

Once I have created this box I can convolve the mask with the image.

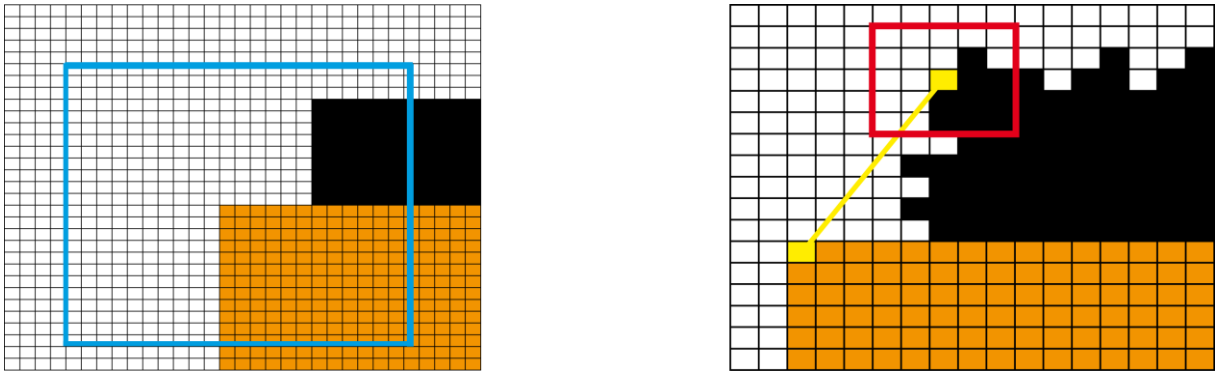


**Figure 52 – Left: The box in blue around the corner of the building in orange and its shadow in black.
Right: the convolution of the mask in red detects the shadow of the corner at the yellow pixel location.**

3.5.5 – Location of the best response

Now that I convolved the mask with the image, I just have to go through every pixel position in the resulting matrix/image and find the one with the lowest value, this one is the best response to the filter in the part of the image delineated by the box.

I do that for every corner of the polygon and keep the couple of points (block corner and shadow corner) with the best filter response. I then compute the distance between these two points to have the size of the shadow. I could have used a different measure like the average of distances or

keep only the two best ones, but the differences between the results are big from one corner to another so I decided to keep only the best one which is supposed to be the most reliable one.

Once I have this distance, I know that it is proportional to the height of the building. So I multiply it by a factor to have the real height of the building. This factor has been found by trying different values and assessing the result on the final 3D model. The result of the height inference is discussed in part 4.2.4. Once we have this last information, we know the complete shape and dimensions of the buildings to rebuild in the 3D model and can focus on displaying it.

3.6 – 3D model creation

This step is quite short as its point is to write the VRML file that is needed to display the 3D model.

This is done with the function `create3Dfile()`. This function first writes the header of the VRML file in a file. Then it adds a black plane that represents the ground. Next, for each detected building it adds this text with the appropriate numerical values:

```
Shape {
            appearance Appearance {
                  material Material {
                  diffuseColor 1 1 1
                  }
            }
            geometry IndexedFaceSet {
                  solid TRUE
                  coord Coordinate {
                        point [92 76 0, 125 109 0, 178 60 0, 137 20 0,
86 70 0, 92 76 26, 125 109 26, 178 60 26, 137 20 26, 86 70 26 ]
                  }
                  coordIndex [4 3 2 1 0 -1 1 6 5 0 -1 2 7 6 1 -1 3 8 7
2 -1 4 9 8 3 -1 0 5 9 4 -1 5 6 7 8 9 ]
            }
}
```

The `Shape` keyword means that a shape is going to be added in the model. The `appearance` keyword specify what the shape is made of, here I choose a simple grey and uniform material with a diffuse colour (not shiny). Then, to describe the geometry in itself I use an `IndexedFaceSet` which is a way of describing geometry by specifying its vertices and its faces. The word `Coordinate` is used to input the points that create the building: the abscissa and ordinate of these points are the pixel coordinates of the detected buildings found at the end of step 3.4. Their z-coordinate are 0 for the points that are one the ground and the inferred height of the building for the points that outline its roof. Once the points have been given, the `coordIndex` keyword inputs the points that make a face. The order of the points specifies the direction of the face normal and the number -1 means that we change face.

For example the first face created by the code above is made of five points:

```
92 76 0, 125 109 0, 178 60 0, 137 20 0, 86 70 0
```

Their z-coordinate is 0 so this is the ground floor of a building. This actually corresponds to the floor of the block at the bottom left part of picture 47 (next page).

This is done for every detected building and it results in a file that describes the detected constructions. It can be opened in any browser which has a VRML plug-in installed. I used Cortona plug-in for Firefox.

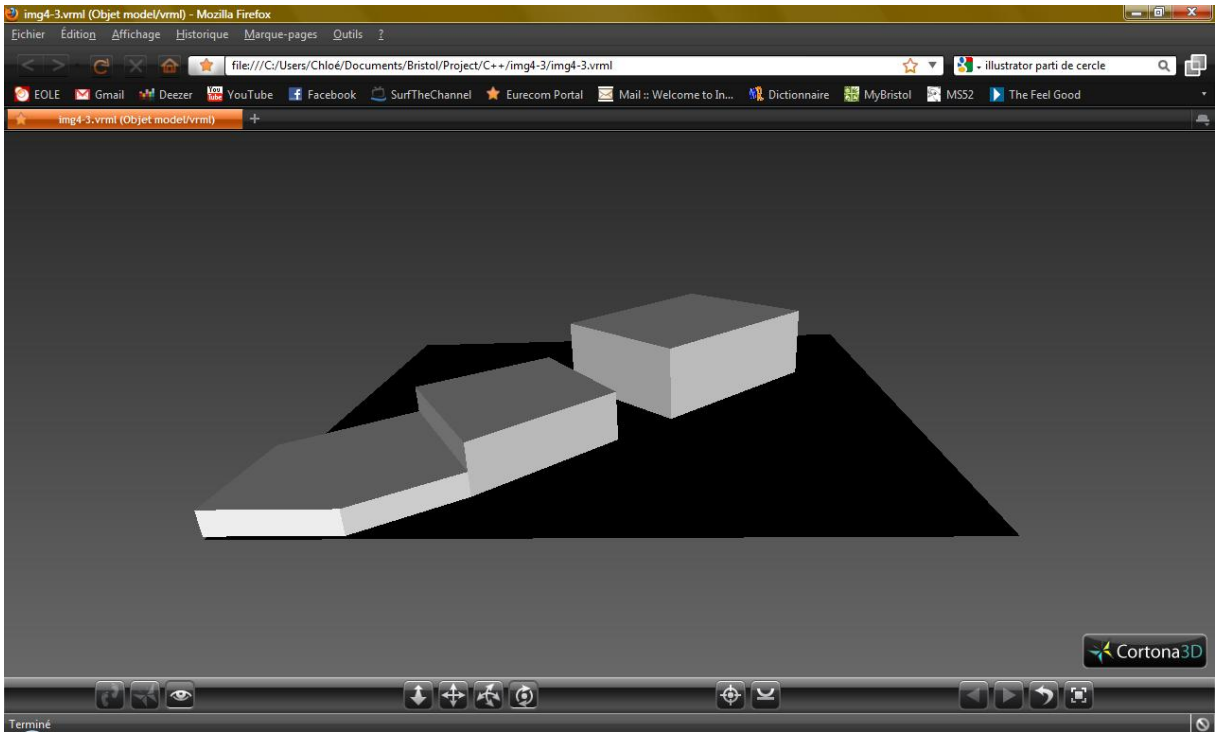


Figure 53 – The resulting 3D model seen in Firefox.

This step ends the part that deals with the design and the implementation of the chosen algorithm. Every step of the studied algorithms has been developed and improvements have been designed to overcome or get round of some of their limitations to result in usable 3D models. The problems that arose during the software development phase have led to the design of new solutions and to a better understanding of the strong points and the shortcomings of the method. The next part analyzes in detail the results obtained using the created software.

4 – Results description and analysis

This part describes the results obtained by running the algorithm on a set of different images. It shows the original image and the resulting 3D model in different cases and explains why it performs good or bad. It identifies the problems we encountered and suggests some ideas to improve the result. The first part focuses on the main limitations that arise whatever the image. Then I focus on the strong points of the software and on images that result in good 3D models before tackling situations where the overall quality of the model is lowered by its limitations.

4.1 – Main limitations of the program

The first limitation of this algorithm lies in the computational power needed to achieve a result. Indeed, if the number of tokens detected in the first step is very large, all the following steps: token linkage, polygons creation and polygon grouping are very computationally expensive. The complexity of the token linkage step for example grows exponentially in the number of tokens because of the recursive function that looks for the different possible paths.

The most expensive steps in terms of time / computational power are the token linkage as we have just seen, as well as the BUS grouping because of the texture computation step which requires going through each pixel in a given polygon, and finally the selection steps where the texture is involved as well.

To run the algorithm we have to reduce the number of detected tokens at the beginning of the algorithm, in the canny filter and the Hough transform step, which means modifying these parameters for each image (see next paragraph). The problem in reducing the number of tokens is that the accuracy of the algorithm is reduced as well. The more tokens are detected, the more building edges are detected and the more building hypotheses are created, which are then reduced by the selection process. By reducing the number of tokens, we are missing building edges at the beginning that cannot be retrieved in the next steps, this leads to a rougher building hypothesis creation and then to a looser building detection at the end. One way to overcome this problem would be to use a technique as described in (Nevatia, et al., 1989) where after detecting the rough edges, we detect parallel edges and U-structures and look for evidence of the presence of the missing edges (see figure below)

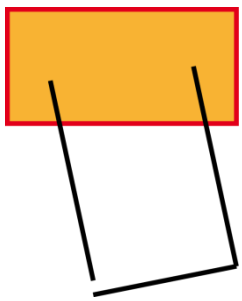


Figure 54 – Three detected tokens in black and the area where to look for evidence of the fourth token in orange.

This limitation forces us to find a trade-off between accuracy and computational power with the threshold used in the first steps of the algorithm.

The second limitation that partly results from the previous one is that we need to choose the parameters of the algorithm before running the program on any image. Unfortunately it is hard to predict the parameters of each step before running it or having an idea of the number of tokens detected and of polygons created. Therefore, we have to try with different thresholds, mainly for the

canny filter and the Hough algorithm, to decide which ones give the best trade-off. I reduce the number of parameters to adjust for each image to nine parameters:

- threshold for the Canny filter (higher threshold)
- threshold for the Hough algorithm (minimum length of the detected tokens)
- use of the `Rectify` function (true or false, this is explained in the next part)
- difference of average colour in the grouping of polygons
- difference of standard deviation in the grouping of polygons
- maximal standard deviation of the texture of a polygon to be considered a building
- minimum size to be selected as a building
- minimum difference of average colour between a polygon and it surrounding
- size of the shade area cast by a building

These parameters are the main ones that need to be changed according to the content and the scale of the image to have a good result and a computational complexity that is manageable (in the order of minutes). The appendix shows the different parameters used for the images I tried the algorithm on.

The next part focuses on the strengths of the algorithm and in the cases where it performs well. Then I tackle the situations where the algorithm give limited results and explain its limitations and what could be done to improve the resulting model.

4.2 – Strengths of the algorithm

I ran the algorithm on a set of images from Google Earth. These are images of mainly industrial and residential areas. The results can be seen in the appendix. In this part I focus on the situations where the algorithm performs well and describe its strengths.

4.2.1 – Detection and token grouping

The token detection and linking steps give an overall good result. The detection is accurate, it detects the most contrasted edges (using the right parameters), be they building edges or other edges. Just from the detected tokens we can already guess the outline of the buildings.

Figure 55 – The edges detected and linked from image 10.

The linking step is quite successful as it turns the sparse edges in longer ones while getting rid of small edges that generally do not correspond to construction edges.

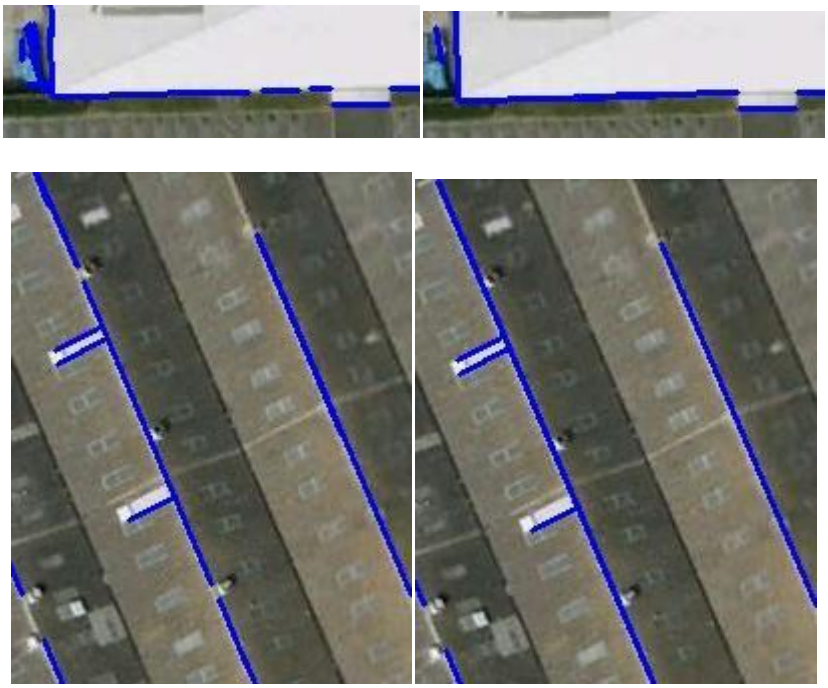


Figure 56 – Top: detail from image 5. Bottom: detail from image 9. The tokens are properly detected and linked.

This step is an important one as the following processing relies on it and we can see that it performs well and gives tokens that correspond to edges, are long enough and unique for the following steps.

4.2.2 – The polygon grouping

Generally speaking, the BUSes grouping process gives a good result as its aim of merging similar polygons is achieved. Even if on the images the grouping step is not striking (see next figure), it is an essential part that enables us to have more reliable results and that works well as long as the polygons are homogeneous enough. The figure below shows a good example of grouping the extracted polygons. The grouping step is iterated twice to give a better result.

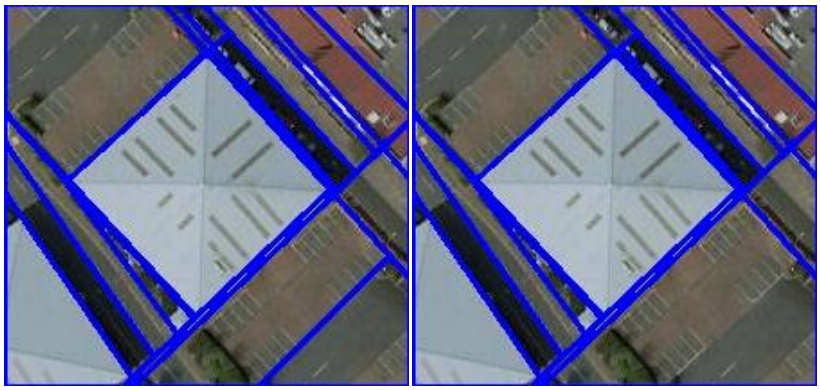


Figure 57 – Grouping of polygons on image 4-2, two polygons are grouped: the one at the bottom right part of the image with the parking lot and the small one at the top in the middle of the image.

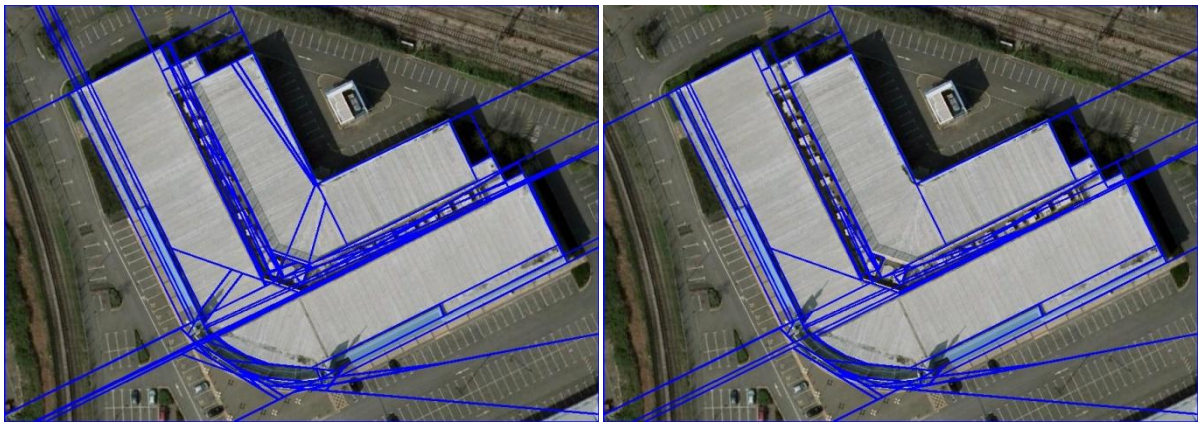


Figure 58 – The original BUSes on the left in image 10 and the grouped ones on the right after two iterations.

Figure 58 shows the difference between the original BUSes and the grouped ones. The result is quite visible in this case. The polygons at the top left of the image that are parking lots are all grouped together and the left part of the roof are less fragmented.

4.2.3 – Selection criteria

The selection criteria are designed for images with these properties:

- buildings with uniform roofs (no pitched roofs or very textured ones)
- buildings that are separated from each other (adjacent buildings are sometimes dropped)
- buildings that stand out the ground (not hidden by the vegetation)

The buildings that fit these criteria are usually very well selected by the algorithm. This applies mainly to industrial areas where the buildings are separated by parking lots or roads, where there is not too much vegetation and where the building roofs are flat. This is illustrated below.

(a) (b) (c) (d)

Figure 59 – (a) Original grouped BUSes. (b) BUSes that are big enough and uniform. (c) BUSes that stand out the ground. (d) BUSes that cast a shadow.

We can easily see from the previous figure that the size selection criterion efficiently removes all the small polygons that we can see at the bottom in the first image. The uniformity criterion removes polygons that are made of roads and trees and shadow as well as the building in the middle where the fourth edge is not in the right location. We can already see that a lot of false buildings are removed after the first selection. The second selection checks that buildings stand out their surroundings and this works well for the five buildings that are properly detected, as well as for some polygons at the bottom which are also different from their surroundings without being buildings. The

last stage that checks if the block casts a shadow efficiently removes the building at the bottom that seems to be a wasteland and does not cast a shadow.

The two last steps: checking if the buildings stand out and measuring the size of the shade area around the buildings are very efficient (figure 59 right and figure 60)

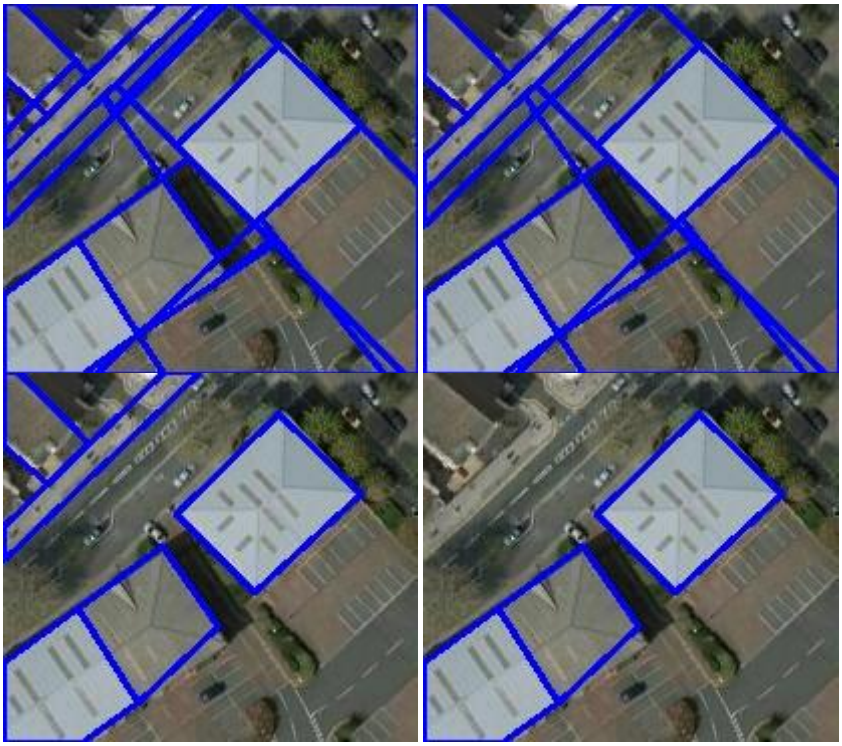


Figure 60 – Top left: Original BUSes. Top right: selection according to the size and uniformity. Bottom Left: selection according to the standing out criterion. Bottom right: selection according to the size of the shade area.

The standing out criterion removes a lot of false buildings in the previous image and the shade area that measure the amount of dark pixel around the polygon deletes the remaining ones on the left. These two last steps are well adapted to this kind of images and result in an efficient building selection.

4.2.4 – Height inference

The height inference is a difficult part because of the lack of information needed to go from two dimensions to three dimensions. The method I used performs well in cases where:

- There is a shadow
- The shadow is neatly outlined (ideally cast on a parking lot or a road, without vegetation)
- The building stands alone

In these cases, the height can be inferred and the results are quite good. For example in the image 7-1 (next page) the 3D model gives us a good idea of the original area.

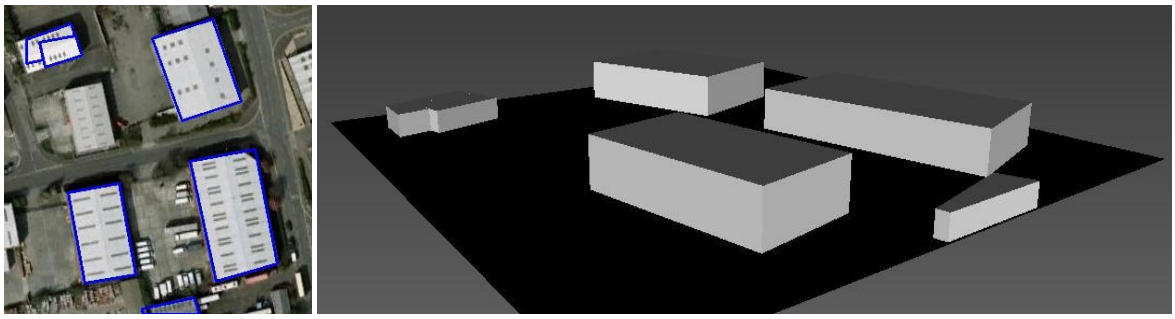


Figure 61 – Image 7-1. Left detected buildings in blue. Right: resulting 3D model.

The image 4-3 gives an example of an image where the height of a building is not well detected because the corner of the shadow of the building at the bottom left of the image cannot be found. This is due to the fact that this building is adjacent to the other one and it leads to a bad height.

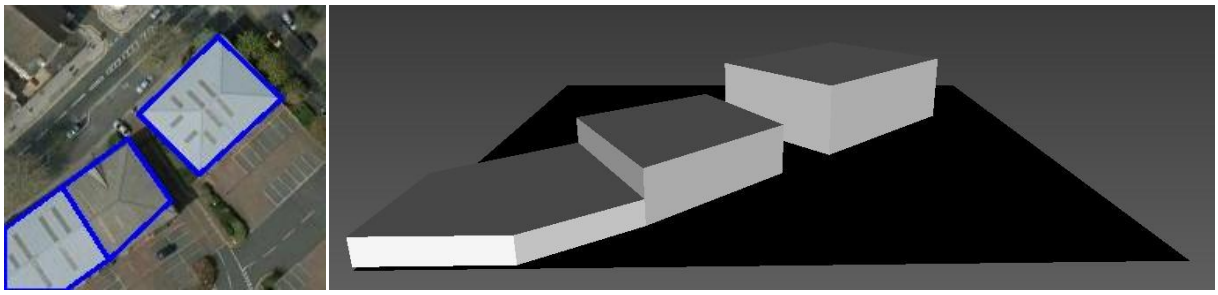


Figure 62 - Image 4-3. Left: detected buildings in blue. Right: resulting 3D model.

The same thing happens in the image 21:

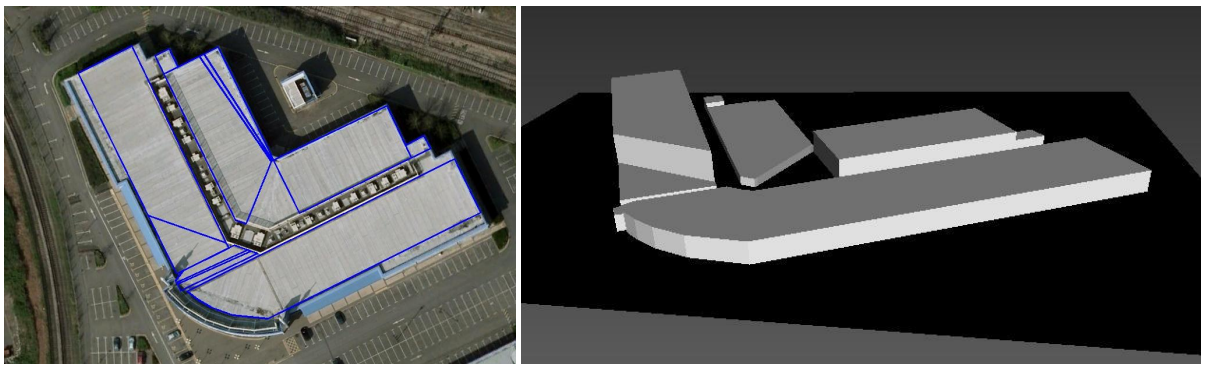


Figure 63 - Image 21. Left: detected buildings in blue. Right: resulting 3D model.

The overall shape is detected but the parts of the building are fragmented. Each polygon is processed as an individual building but some of them do not cast shadow because they are adjacent to other ones. The height inference is tailored to images where buildings are smaller than on the image 21 and are detected as a single polygon as in image 4-3 or 7-1. To be able to detect only one big building, the grouping step would have to be more efficient, maybe by using a different texture description. This is discussed in part 4.3. Overall the height inference give us a good idea of the original landscape, be it one building or a set of scattered buildings. The result of this step is satisfying given the 2D information that we have at the beginning.

4.2.5 – Wide range of detected shapes

Even if the algorithm is tailored to detect mainly convex polygons of approximately the same sizes, we have seen in the images used to test it that it performs well with convex polygons of different sizes, with concave polygons and with buildings with rounded parts. The images below illustrate these different cases.

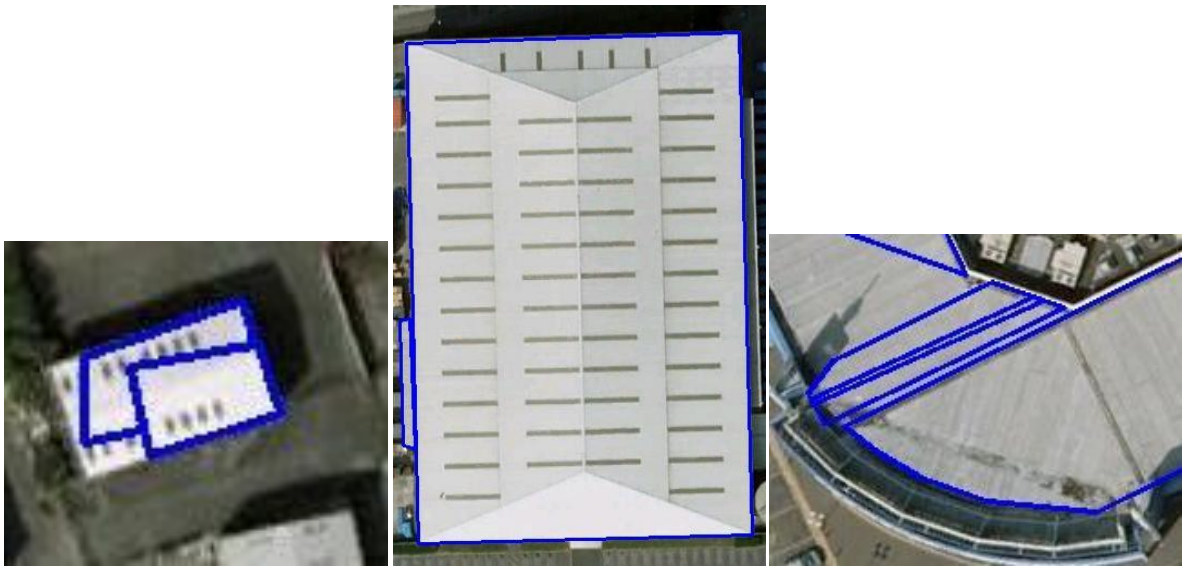


Figure 64 – Left: Concave polygon on image 7-1. Middle: Different sizes of polygons on image 5. Left: Rounded building part on image 10.

Concave polygons are handled properly (even if on the image it is not a real concave polygon, it shows that it is possible) because of the way BUSes are described: as a set of points given in an anti-clockwise direction. It makes it possible to have different levels of details for the polygons (the *Simplify* function makes it possible to have a concise description of their outlines) and to describe their shapes accurately. It gives a lot of flexibility to the algorithm. The second image shows that it is possible to detect buildings of very different sizes in the same image which is valuable like on image 5. The last one shows that it is possible to detect rounded building parts by approximating them by straight edges and that the resulting 3D model is quite accurate.

As a conclusion, the algorithm performs well on images with a small number of separated buildings with flat roofs, typically industrial areas. In these cases, the reliability of the algorithm is good and the 3D model is quite true to the original scene. On the set of eleven images that verifies these properties I find an average of 71% of buildings detected properly and 28% of missed buildings (see appendix for the chart). This rate of missed buildings changes a lot according to the pictures from 0% to 100% whereas the rate of detected buildings has a lower deviation. Moreover, the algorithm has indubitable strengths like its flexibility to detect various shapes and sizes of buildings, its selection criteria that are well-adapted to industrial areas and the height inference that gives an overall good result.

4.3 – Limitations of the algorithm

In this part I focus on the problems that arise while running the algorithm on different kinds of images. I explain for each of them why the result is not what we would expect and suggest improvements to the existing algorithm to overcome these limitations. We have already briefly mentioned some problems in the previous parts. This section focuses in detail on the problems of the scale of the image, of the limitations created by the shades and by the low contrast of some images, of the need to have separated buildings, of the problems of the roofs and finally of the shortcomings inherent in the functions Sort and Rectify.

4.3.1 – Scale

In this part I deal with the problem of the scale and the size of the image. They can be an obstacle and have to be chosen carefully for several reasons: the computational power first, the accuracy of the detection and the need to adjust the parameters.

The scale of the image means the level of zoom on the image or the altitude the image has been taken from. The first important thing is that the smaller is the scale (the smaller the buildings in the image), the larger the number of tokens to describe it is needed, because there are more constructions in the image. This increases the number of tokens to process which leads to a really heavy computational load and is not manageable.

Another reason to have a large scale is that if there are a lot of small buildings in one image it is harder to detect every edge of each building. The rate of missed edges is bigger because the edges are smaller and the accuracy of the result decreases. The bigger the buildings in the image the more likely it is to detect at least part of each edge, and this is important because the algorithm is not able to rebuild a building without detecting all of its edges at the beginning. This is actually a different limitation which is the cause of a lot of missed buildings. If one edge is missed, there is no token to split the image properly in the splitting step, therefore the resulting BUS does not correspond to the building, it generally includes the building and part of something else with a different texture. The resulting BUS is not homogeneous and is selected in the following steps. An illustration of this situation is shown below.

Figure 65 – Left: The original image with the detected tokens in blue. Middle: detail of the previous image with the detected tokens and the real building in orange. Right: The BUS created from the previous tokens and the building in orange.

We can see in the previous image that the BUS created from the extracted tokens does not fit the real building in orange because the edge at the top of the building has not been detected. The result of the extracted edge could be improved by using the technique described in 4.1 and illustrated in figure 54 where we specifically look for the missing edges of a set of typically three tokens. But the easiest way to avoid this situation is to have a scale which is adapted to the algorithm, where there are a few buildings in the image.

However, another problem arises if there is only one building in the image or generally speaking if the buildings in the image are too large. In this case the buildings are split in several parts by the splitting process and are not aggregated enough by the grouping step because of discrepancies in the polygons textures. Each polygon is then considered as a single building and the selection step removes some of these polygons because they do not stand out their surroundings or they do not cast shadow. This is what happens with the image 10 below.

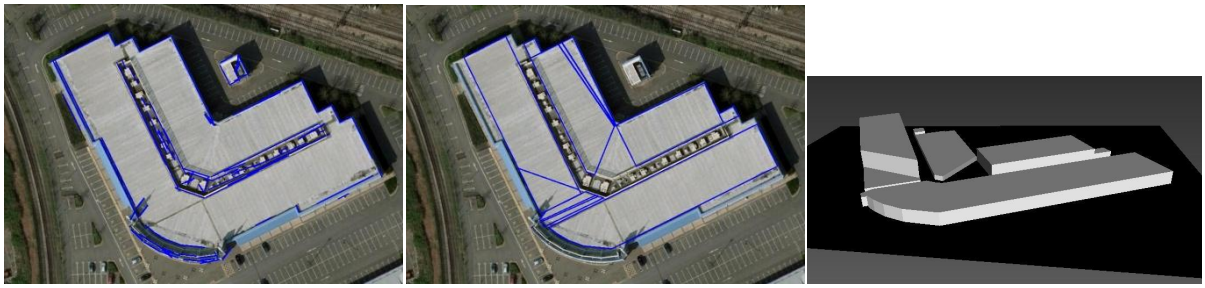


Figure 66 – Left: the linked tokens of the original building. Middle: the resulting BUSes. Right: the 3D model

On the previous image, the building is fragmented and the height inference for each part is very difficult to do, that is why even if we can guess the shape of the building and recognize it on the original picture the 3D model is still very rough. This image scale is too big to give a good result.

The figure below shows images of different scales used to run the algorithm.

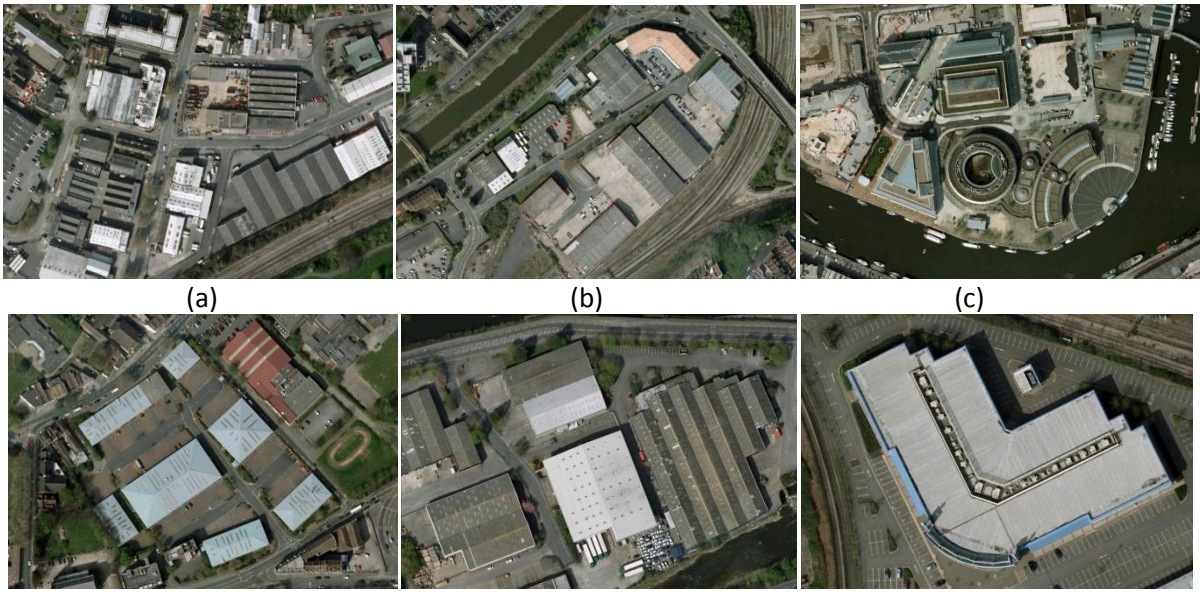


Figure 67 – Different images with different scales. (a) image 3 : altitude 339 m (b) image 19 : altitude 437 m (c) image 8 : 386 m (d) image 4 : 301 m (e) image 21 : 250 m (f) image 10 : 162 m

These images are pictures of industrial areas, with buildings of various shapes and mainly uniform roofs but taken from different altitudes. The scales of images (a), (b) and (c) are too small, they give very rough results and a lot of buildings are missed even when the parameters are properly adjusted (see appendix for the final 3D model). The scales of images (d) and (e) are good, they both give good 3D models. The scale of image (f) is too big.

The figure below shows a comparison of the result of the same image at different scales.

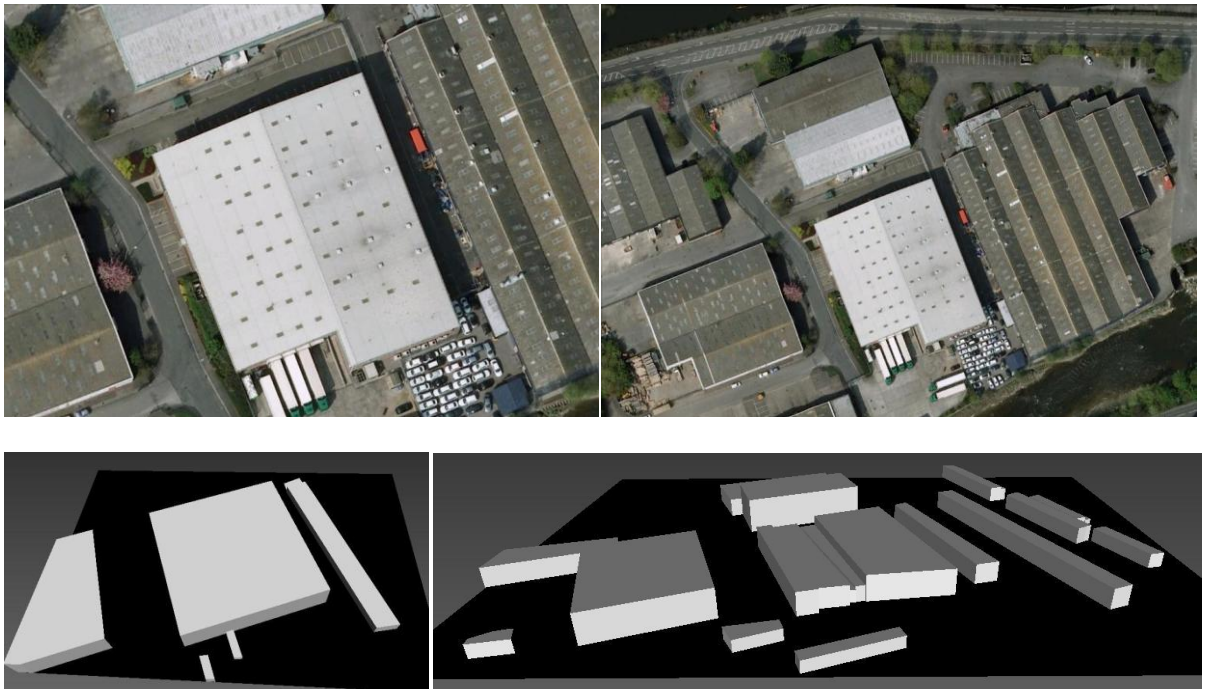


Figure 68 – Left: image 9 and the resulting model at the bottom. Right: same image but smaller scale and the 3D model.

In the left image with the bigger scale, the building at the top has been missed, some trucks are considered as buildings and only a small part of the warehouse on the right is detected. On the same image with a larger scale, the buildings at the top are detected but there are more false positives and the building in the middle is fragmented. Each scale gives a different result, the bigger scale gives a more accurate result on the detected buildings whereas the smaller scale gives a better idea of the whole area but is less accurate.

Once the scale has been chosen it is better not to change because it requires changing the parameters of the algorithm: parameters to link the tokens because the tokens do not have the same size from one image to the other, and parameters to select the buildings like their areas or the size of their shadows. A pre-processing step would be useful to automatically find these parameters, before any other stage. This could use a description of the texture of the whole image: a coarse texture would generally imply that the buildings are big. We could also use the size of the extracted tokens after the Hough algorithm to have an idea of the parameters to use next. This pre-processing step would be useful to improve all the limitations described in the next parts where parameter fit is important.

The size of the image is important too, as different image sizes at the same scale can give very different results. It is interesting to have big images because they give more clues about the surroundings of the buildings. In small images a lot of buildings are incomplete, some shade areas are missing, some shadow corners are hidden and this lowers the accuracy of the result. Moreover a small image reduces the number of tokens orientation and distorts the rectification of the orientation (see part on the *Rectify* function). On the other side, a smaller image is less expensive to process and can sometimes remove parts of the images that are useless and disturb the detection process. However if the application had to run on a whole city area, no area can be missed by cutting an image and they would probably have to be big anyway, to reduce the number of junctions between images (and 3D models).

The images below show different results on images of different sizes coming from the same original image (same scale).

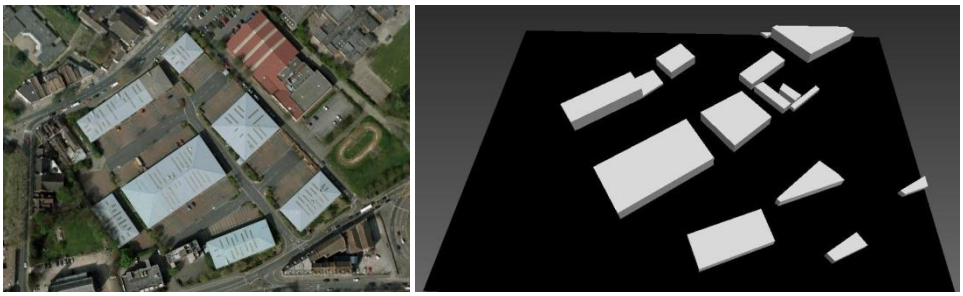


Figure 69 – Original image (image 4: 879 x 620) and the resulting 3D model

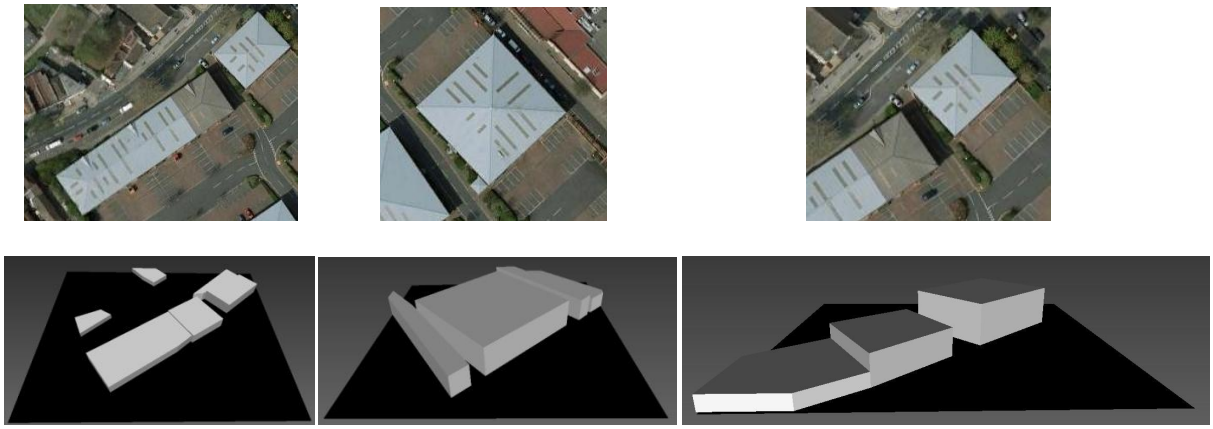


Figure 70 – Parts of the previous image and their models: image 4-1 (322 x 257), image 4-2 (225 x 212), image 4-3 (207 x 184)

The different image sizes used to create the 3D models required different parameters to achieve a satisfying result. On the left, a part of the main building is missing at the bottom left of the image while a false positive is detected. In the middle, the square building is detected as well as two other buildings that are not in the picture. The third picture on the right gives a good result except that the height of one building is fake because it is incomplete and adjacent to another block. As a result, large images (900 x 600) tend to give better results because they are more complete and have more information that can be used by the algorithm.

Finally, to reconstruct a large area of a city by using several pictures it is better to choose large images with a scale that gives good results like the one on image 4 and 21 (altitude between 250 and 300m), where there is between five and ten big buildings, and not to change scale for the whole process. This gives us a manageable computational complexity but still a good accuracy to detect the buildings which are not too small or too fragmented.

4.3.2 – Shadows

Shadows are used in the algorithm to gain information about the height of the buildings and about their likelihood to be real buildings. However, shadows also create some problems in the detection and selection processes, for example shadow edges can be detected as building edges, some parts of the urban scenes cast shadows without being buildings and some other parts are surrounded by shadows and therefore are selected as buildings.

The first step that suffers from the presence of shadows is the edge detection step with the canny algorithm. In cases where the shadow is much contrasted on the ground the edge of the shadow is detected as a building edge. This adds tokens that create BUSes that need to be removed in the following steps.

Figure 71 – Tokens are detected at the edge of the shadow in image 11 and 4-2. These tokens are not removed and create false buildings.

The previous pictures show that shadows can create buildings where there are no real ones. They first create edges that generate polygons, these polygons are kept by the selection process because they are surrounded by a shade area like real constructions. It is very difficult to make the difference between these polygons and actual buildings as they have the same properties: they are big enough to be buildings, they have uniform texture (parking lots or streets), they stand out their surroundings and they cast a shadow. An idea to avoid this situation would be to be able to compare the building's shape and the shade's shape to see if the shade belongs to the buildings next to it or not. This means that the algorithm needs a robust shade detection processing whereas we only detect the size of the shade and its corners. For example in the previous picture it would extract the shade at the bottom left of the image and compare its dimensions with the detected building to see that it does not fit (see below).

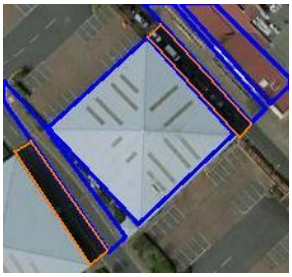


Figure 72 – The shades to detect in orange. They would be compared with the detected polygons.

The second problem with the shades is that some elements of the urban landscape cast shadows even if they are not buildings. If they are uniform and big enough they are kept by the selection step.

Figure 73 – In this detail of the image 8, the road casts a shadow on the railroad below.

The previously detected polygons could be removed by the selection step if we used a lower threshold for the uniformity of the texture but it is likely to discard real buildings. We could also use the shape of the polygon which is very long and narrow to detect that it cannot be a block roof given the scale of the image, but it is very specific to this particular case.

The last problem that arises while running the algorithm on different images is that shadows from other buildings can distort the result of the selection criterion that decided whether a polygon stands out its surrounding. Some pieces of the urban landscape like parking lots are surrounded by shadows from buildings and are taken for real buildings. This is an intrinsic limitation of this criterion that only counts the number of dark pixels and does not take their location into account. The software would need deeper processing of the shadow to avoid this situation. Adding a module that would compare the shape of the shade with the shape of the building like in figure 72 would solve or at least reduce this problem.

Figure 74 – Details from image 5 and 21 where some polygons are selected because of a shade area that is cast by another building.

Shadows are a valuable source of information and therefore cannot be removed or ignored while running the algorithm, but they also limit its the result by inserting new edges that create artificial building hypotheses and by making the selection process more difficult. However, an improved shadow processing as explained before seems to be able to lessen this impact and improve the global result of this algorithm.

4.3.3 – Contrast

The algorithm needs a good contrast to work properly. This is important for the early edge detection step as well as for the selection steps.

The edge detection stage relies on a highly contrasted image to be able to detect all the edges of one building. As we have seen earlier (4.1 and 4.3.1) it is very important to detect all these edges because if only one is missing the building cannot be reconstructed.

Figure 75 – Image 7-1. Left: the left edge of the building outlined in red is not detected. Middle: the resulting BUS. Right: the final buildings.

Figure 76 – Detail from image 4: the detected tokens and the original BUSes.

In some cases the contrast of one of the edges is not good enough to be detected because of some vegetation (figure 76) or because the roofs outline is not marked enough and the roof has the same texture as the ground (figure 75). The first idea to overcome this limitation is to increase the accuracy of the edge detection by increasing the threshold of the canny filter, but this would add a huge number of tokens that would not correspond to any buildings and the computational load would be too heavy. We could also work on an image where the contrast has been enhanced but this would also increase the contrast of the shadow on the ground and of details we are not interested in.

Figure 77 – The previous grayscale images with equalized histograms. The hidden edges do not seem to be more visible.

However the solution described in 4.1 seems to fit best this problem and increases our chances to detect the last edge properly.

Another problem due to a low contrast is the “Stand out” criterion used to select buildings. If some buildings are properly detected but do not have a colour difference big enough with their surroundings they are removed from the BUS selection. This happens when the contrast is not big enough in the building area or when there is some vegetation or shadows like in the image below.

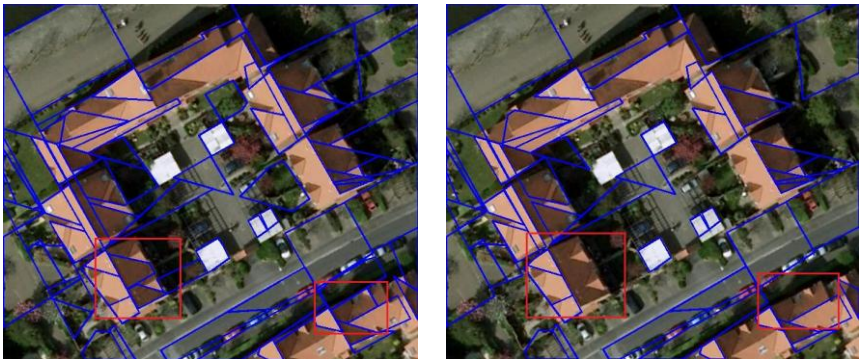


Figure 78 – Left: BUSes selected according to their size and their texture homogeneity. Right: the polygons in the area outlined in red do not verify the stand out criterion and are discarded.

The figure on the previous page shows parts of roofs that are in the shade and therefore do not stand out their surroundings and are not considered as buildings. This is also part of another problem that is discussed in the next section about roofs and especially pitched roofs.

4.3.4 – Roofs

We have seen previously that only uniform roofs are selected among all the building hypotheses and that the algorithm is well adapted to buildings with flat roofs. These two limitations are due to the selection criterion and to the architecture of the algorithm in itself.

One of the criteria to select the roofs is the polygon homogeneity. It is measured by the standard deviation of its texture. This leads to the removal of some BUSes that corresponds to buildings but do not have homogeneous roofs (see figure below)

Figure 79 – Left: detected BUSes on image 5. Right: selected buses according to their uniformity.

On the pictures above, some polygons fit the buildings exactly but are not selected because their roof texture is heterogeneous. The first idea to overcome this problem is to increase the threshold of the selection but this also increases the number of false positive. The images 19 and 19-1 also suffer from this limitation as the roof of the buildings are very textured and made of several parts. The building in the middle is detected as several buildings and the result is not what is expected.

Figure 80 – Left: The BUSes detected on images 19-1. Right: The final buildings.

The previous image is a picture of part of a city where the roofs are very textured and made of several elements and only one part of the roof of the building is detected.

The right approach in the previous cases would probably be to use another texture description. This could be using a Run-length primitive (Burghardt, 2010) that gives the length of texture primitives (like grey levels) in different directions, or this could by using Law's texture energy measure (Burghardt, 2010) that uses different vectors that evaluate the average grey level, the edges or the spots then convolve them together to get a mask to convolve on the image. The response to the mask gives description features. These different descriptions are more accurate than the one we use in the algorithm and would give better results for the grouping and the selection step.

Another problem to handle is the pitched roofs that are a feature of residential areas. Each roof section has the same texture except that usually one is in the shade and one is lit by the sun. If we used a texture description that recognizes that they are the same texture but lit differently (same hue but different brightness for example) it would be easier to select them. It could also be a way to group them during the grouping step. The results of the algorithm on images of residential areas are usually bad because of several factors:

- Different sections of pitched roof have very different brightness and create edges in various directions.
- Among these edges, some are missed whereas others are unnecessary: this creates a lot of building hypotheses with different shapes and colours and are hard to select.
- If the roofs sections are detected independently: they are not grouped and are treated independently. They are not kept by the selection process because they do not stand out or do not cast shadows.
- If the roof sections are detected as part of the same polygon, their texture is not uniform and they are discarded by the hypothesis selection stage.

Figure 81 – Image 11, 18-1 and 23. They show some residential areas with pitched roofs and the kind of tokens detection that can be achieved as well as the created BUSes and selected BUSes.

At the end the detected buildings are very rough or are not at the right location. The algorithm is not design to model a pitched roof and the best result we can achieve is to detect accurately their locations and outlines and model them with a flat roof.

The problem of the homogeneity and the shape of the roofs would need another new module to get a more accurate texture description than the one we used, and this would be used in the grouping part and in the selection steps to get a better final result. However, the algorithm is not designed to process this kind of roofs and other methods that fit better this problem are described in the literature (Fischer, et al., 1998) and should be implemented to have a satisfying result on this type of images.

4.3.5 – The Sort function

The Sort function is called after the tokens have been linked and their orientation rectified and just before splitting the image. This is an important step because it decides on the way the image is split and hence on the number of buildings that are split and need to be grouped. The aim of this step is to fit the real buildings as much as possible and to reduce the number of polygons to be grouped. The figure below illustrates different splitting situations.

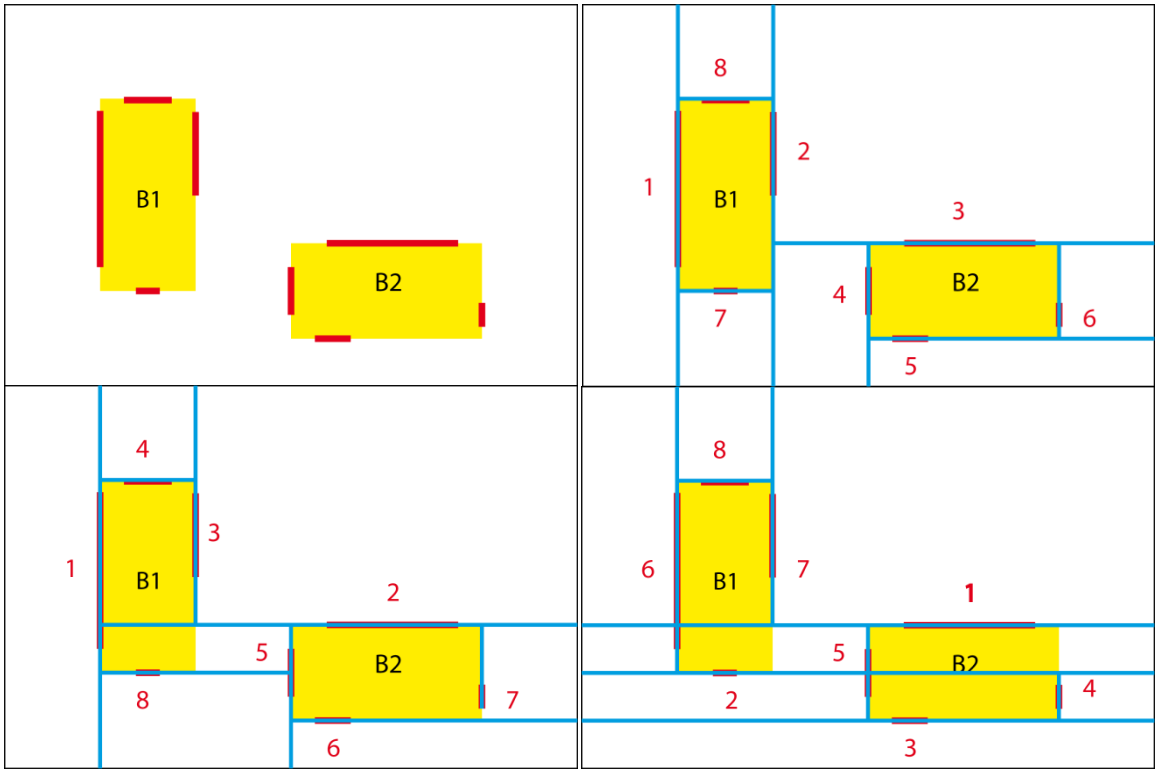


Figure 82 – Top left: the original image with two buildings in yellow and its detected edges in red. Top right: and ideal sorting of the tokens and the resulting ideal splitting of the image. Bottom left: sorting of the tokens according to their length. Bottom right: bad sorting of the tokens and the resulting splitting.

In the figure above, all the edges of the buildings have been at least partly detected, this is a simplified situation where there is no edge detected inside the building and where only one token is used for each edge building but it still illustrates pretty well my point. The diagram at the top right shows a situation where the splitting of the image is perfect because the tokens are sorted in a way that the binary tree creates polygons that do not split the real buildings at all. The polygons will not be changed by the grouping process and will directly undergo the selection step. In the bottom left

diagram, the tokens are sorted according to their length, this creates a situation where a part of the building B1 is separated from the recognized building. This polygon is not homogeneous and is discarded by the selection process. At the bottom right of the figure, the tokens have been sorted randomly and the same situation happens: both buildings are split and the blocks detected at the end are only parts of the real one.

This figure explains some situations in the real images where buildings are not detected or where they are split in the middle. It is important to choose carefully the criterion to sort the tokens so that they fit best the buildings. This would require a study in itself but I tried several criteria: sorting by contrast, by length or by the product length x ratio. I found that sorting by length is quite reliable because if the token is long it means that it has been created by linking several other tokens. This is a clue that there are several detected edges that are aligned (and verified the other criteria to link tokens) so that the evidence of the existence of a real edge is very high. The corresponding token has to be put in the beginning of the list of sorted tokens so that we can be sure that it splits the image along the entire real edge and not just a part as Token 3 at the bottom of the diagram for example. However, the results can be different from one image to another and it can be a good idea to adjust it depending on the image and the density of detected tokens.

An idea to overcome this problem would be not to use a binary splitting tree but simply to split the whole image according to every tokens, nevertheless this dramatically increases the number of polygons to be processed later and thus the computational power needed and this also over-segment the image in a lot of BUSes that need to be grouped. This solution shifts the load of the algorithm on the grouping step that is not always very successful because of the texture description as we have seen earlier, instead of really solving the splitting problem.

This part underlines the fact that even a well-designed step of the algorithm can generate some limitations and that this step also needs to be adapted to the images and to the rest of the algorithm.

4.3.6 – The Rectify function

This function is the one that rectifies the orientations of the tokens after they are linked and before they are sorted to split the image. The goal of this function is to remove tokens with an orientation that do not fit the main ones because they are likely to be useless for the rest of the algorithm. This is very useful while running the algorithm on most images; this reduces the segmentation of the image and the number of polygons to group. The figure below shows the difference of result with and without the Rectify on image 7-1.

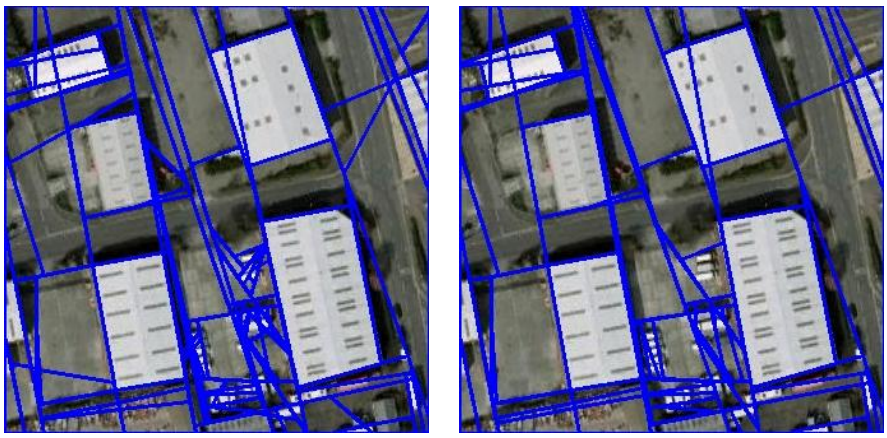


Figure 83 – Left: with the Rectify function. Right: without the Rectify function.

We can easily see that the *Rectify* function prevents the creation of some small polygons at the bottom of the images. These polygons are created because of the trucks there that generate a lot of tokens with very different orientations. They do not improve the detection of the buildings, most of them are very small and triangular and would be deleted in the next steps anyway but the *Rectify* function prevents their creation and reduces the load on the following parts of the algorithm.

Nonetheless in some cases and especially on small images the main directions do not include the directions of some small tokens that are very useful to detect the edges of the buildings and thus are deleted. This results in an image splitting that do not fit the buildings at all. An example is given below.

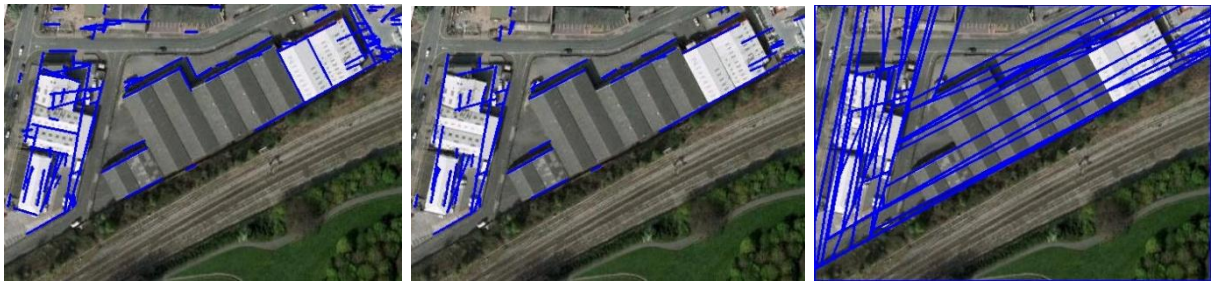


Figure 84 – Image 2. Left: the linked tokens. Middle the rectified tokens. Right: The created BUSes

In the previous image the lack of contrast makes it difficult to detect the edges of the building close to the railroad; however some vertical edges are detected for the buildings on the right side of the image and on the shadow, which could be used to create BUSes. Even so, the rectify function removes these tokens and the resulting BUSes do not fit the buildings at all. The figure below shows the result without using the Rectify function.

Figure 85 – Left: with the Rectify function. Right: without the Rectify function

Even if the big building has not been detected at the bottom, the BUSes on the right and on the left have been and are reconstructed in the final 3D model. For the first image no building are reconstructed at the end.

Another situation where the function Rectify creates a limitation is where a part of the building is rounded. The algorithm is tailored to detect buildings of various shapes with straight edges, not especially with rounded parts. However the algorithm can rebuild rounded parts by approximating them with small straights segments but this means that a lot of different directions are involved and have to be kept. The picture below shows the result of the detection and building of the 3D model from image 10, where a part of the building is rounded, with and without using the Rectify function.

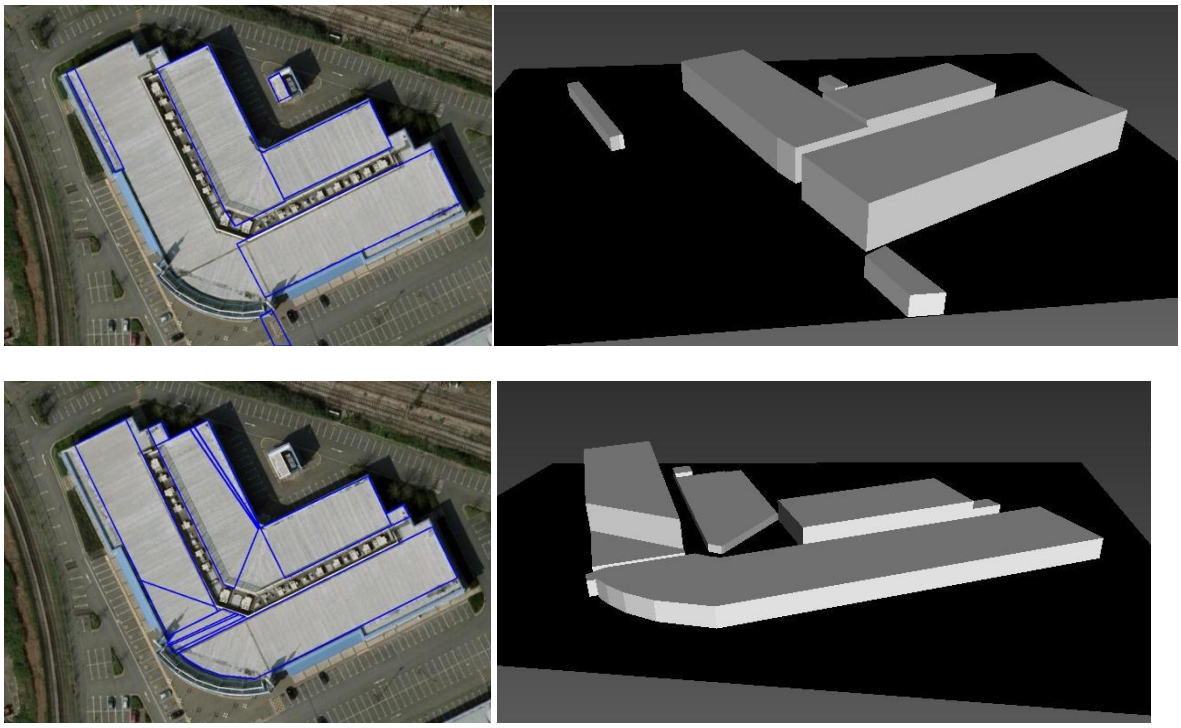


Figure 86 – Top: Final BUSes and final 3D model using the Rectify function. Bottom: Final BUSes and final 3D model without the Rectify function.

The effect of the *Rectify* function on the previous image is striking; it deletes all the tokens that make the rounded part of the building and thus removes a whole part of it. Without the *Rectify* function the rounded shape is rendered perfectly.

The first thing I do to solve this problem is to add a Boolean parameter that enables the user to choose whether or not to use the Rectify function. Still, this solution means that the user has to wait and see the linked tokens or the BUSes created with the function before deciding to use it or not, there is no automatic way to decide this. We could increase the number of directions that are considered as main directions but this would eliminate the purpose of the function. We could also think about automatically accepting directions that are perpendicular to the main ones for example, this would probably solve the problem in image 2, but not the problem of the rounded parts. Deciding whether or not to use this function needs the user input but manages to solve all this problems.

4.3.7 – Summary

This part presented different kinds of limitations of the algorithm, first limitations due to the type of images we choose and especially about their scale and size. Then we tackled limitations due to the content of the images we used: the problems created by shadows cast by the buildings, by a low contrast of the images and by the kind of roofs we deal with. The last kind of limitations is the one brought by the algorithm in itself and specifically by the Rectify and Sort functions. Some new modules and techniques are needed to solve some of these problems like a new way to describe the roof texture for instance or to process the pitched roof in general. Other problems need to spend more time adapting the parameters or the techniques we used to each image, for the sorting process for example.

Conclusion and future work

The first thing to be learnt from the implementation of this algorithm is that the technique to use to detect buildings has to be tailored to the type of building we want to reconstruct. The algorithm I chose to implement performs well on images of industrial areas where buildings are scattered, with flat roofs and where the scale is appropriate. Its strengths are its flexibility, its efficiency in the selecting process of the building hypothesis and its height inference. However improvements could be done to overcome limitations due to the contrast of the image or shadows and roofs.

The implementation of the Boldt algorithm (Boldt M., 1989)and the technique described by Sohn & Dowman (Sohn, et al., 2001) give similar results to the one described in their articles. The building hypothesis creation described previously in 2.2 was applied to the same kind of buildings my program performs well on. The selection process I designed for this application works well on industrial areas and the height inference performs well given its low complexity and the input we used. The algorithm is limited when it comes to residential areas mainly because of the pitched roofs of the houses and of other limitations like shadows and low contrast that are enhanced in these areas. Some improvements can be added to the current algorithm to overcome some of its shortcomings, nevertheless to build a 3D model of another kind of areas, other algorithms would be more appropriate than this one.

Another important fact is that even while using the algorithm on identical areas, the parameters have to be adjusted to the image and this is an obstacle to the automation of the algorithm to work on multiple images. A solution would be to implement a pre-processing step that determines the parameters before running the algorithm. This is closely related to the fact that this kind of processing is computationally expensive and needs the parameters to be adjusted to have a complexity that is manageable.

Further work to improve or add functionalities to the program includes:

- Implementing the improvements described in part 4.3 :
  - Processing shadows to remove fake buildings
  - Add a module that looks for missing edges from detected structures in extracted tokens
  - Choose and implement another texture description method for the roofs
  - Use an appropriate method to process pitched roofs
  - Adjust the token sorting criterion
- Use a pre-processing step to determine the algorithm parameters before running the program.
- Add a framework to use the algorithm on multiple and adjacent images to build a 3D model of a large landscape from small images with a big scale.
- Texture the roof of the resulting model by extracting textures for the 2D image.

These improvements are based on the conclusions drawn from the implementation of the chosen algorithm. It enabled us to know its strengths and limitations for practical purposes and to design improvements to the chosen method. It fulfilled its aim to give us a good idea of what is achievable in terms of reconstruction of a 3D urban scene from a 2D satellite image.

References

Boldt M., Weiss R., Riseman E. 1989. Token-based Extraction of Straight Lines. *IEEE Transactions on Systems, Man and Cybernetics.* November/December 1989. Vol. 19(6).

Bruce, Irvin R. and McKeown, JR David M. 1989. Methods for Exploiting the Relationship Between Buildings and Their Shadows in Aerial Images. *IEEE Transactions on Systems, Man and Cybernetics.* November/December 1989. Vol. 19, 6, pp. 1564-1575.

Burghardt, Tilo. 2010. Image Processing & Computer Vision : Segmentation Basics. *University of Bristol Computer Science : Resources.* [Online] 2010. [Cited: 09 09 2010.] http://www.cs.bris.ac.uk/Teaching/Resources/COMS30121/slidesIP/2010/COMS30121_IPCV_L5_1pp.pdf.

Burghardt,Tilo. 2010. Image Processing and Computer Vision - Texture Description. *Dept of Computer Science, Univeristy of Bristol.* [Online] 2010. [Cited: 15 09 2010.] http://www.cs.bris.ac.uk/Teaching/Resources/COMS30121/slidesIP/2010/COMS30121_IPCV_L7_1pp.pdf.

Burns, Brian J., Hanson, Allen R. and Riseman, Edward M. 1986. Extracting Straight Lines. *IEEE Transactions on Pattern Analysis and Machine Intelligence.* July 1986. Vols. PAMI-8, 4, pp. 425-455.

Canny, John. 1986. A Computational Approach to Edge Detection. *IEEE Transactions on Pattern and Machine Intelligence.* November 1986. Vols. PAMI-8, 6, pp. 679-698.

Cortona3D. 2009. Cortona3D Viewer. *Interactive 3D Technical Communications.* [Online] 2009. [Cited: 4 May 2010.] http://www.cortona3d.com/Products/Cortona-3D-Viewer.aspx.

Duda, Richard O. and Hart, Peter E. 1972. Use of the Hough Transform to detect lines and curves in pictures. *Communications of the ACM.* January 1972. Vol. 15, 1, pp. 11-15.

Fischer, André, Kolbe, Thomas H., Lang, Felicitas, Cremers, Armin B., Forstner, Wolfgang, Pümer, Lutz, Steinhage, Volker. 1998. Extracting Buildings from Aerial Images Using Hierarchical Aggregation in 2d and 3D. November 1998. Vol. 72, 2, pp. 185-203.

Garnica, Carsten, Boochs, Franck and Twardochlib, Marek. 2000. A New Approach to Edge-Preserving Smoothing for Edge Extraction and Image Segmentation. *International Society for Photogrammetry and Remote Sensing.* Amsterdam, The Netherlands : s.n., 16-23 July 2000. Vol. XXXIII, Part B3.

George Vosselman, Ildiko Suveg. 2004. Reconstruction of 3D building models from aerial images and maps. *ISPRS Journal of Photogrammetry & Remote Sensing.* 2004. Vol. 58, pp. 202-224.

Jaynes, Christopher O., Stolle, Franck and Collins, Robert T. 1994. Task driven perceptual organization for extraction of rooftop polygons. *IEEE Workshop on Application of Computer Vision.* 1994. pp. 152-159.

McGlone, J. Chris and Shufelt, Jefferay A. 1994. Projective and Object Space Geometry for Monocular Building Extraction. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition.* 1994. pp. 54-61.

Nevatia, Ramakant and Lin, Chungan. 1998. Building Detection and Description from a Single Intensity Image. *Computer Vision and Image Understanding.* s.l. : Elsevier Science Inc., November 1998. Vol. 72 Issue 2, pp. 101-121.

Nevatia, Ramakant and Mohan, Rakesh. 1989. Using Perceptual Information to Extract 3D Structures. *IEEE Transactions on Pattern Analysis and Machine Intelligence.* November 1989. Vol. 11, 11, pp. 1121-1139.

Nevatia, Ramakant. 1982. *Machine Perception.* Englewood Cliffs, New Jersey : Prentice-Hall Inc., 1982.

OpenCVWiki. 2010. Welcome - OpenCV Wiki. *OpenCV Wiki.* [Online] 10 06 2010. [Cited: 01 09 2010.] http://opencv.willowgarage.com/wiki/.

OpenVRML. 2009. OpenVRML. *OpenVRML.* [Online] 2009. [Cited: 4 May 2010.] http://www.openvrml.org/.

Petrou, M. and Bosdogianni, P. 2004. *Image Processing the Fundamentals.* 2004.

S. U. Zheltov, A. V. Sibiryakov, A. E. Bibitchev. 2001. Building Extraction at the State Research Institute of Aviation Systems (GosNIIAS). *Automatic Extraction of Man-Made Objects from Aerial and Space Images.* 2001. Vol. III.

Shiyong, C., Qin, Y., Zhengjun, L., Min, L. 2009. Robust Building Outline Representation And Extraction. *International Archives of Photogrammetry Remote Sensing and Spatial Information Science.* September 2009. Vols. VOLUME XXXVIII-7/C4.

Simple Linear Regression - 2010. - Wikipedia, the free encyclopedia. *Wikipedia.* [Online] 3 June 2010. [Cited: 2 September 2010.] http://en.wikipedia.org/wiki/Simple_linear_regression.

Sohn, G. and Dowman, I.J. 2001. Extraction of buildings from high resolution satellite data. *Automatic Extraction of Man-Made Objects from Aerial and Space Images.* 2001. Vol. III.

Venkateswar, V. and Chellappa, Rama. 1992. Extraction of Straight Lines in Aerial Images. *IEEE Transactions on Pattern Analysis and Machine Intelligence.* November 1992. Vol. 14, 11, pp. 1111-1114.

VRML. 2010. VRML. *Wikipedia.* [Online] 27 April 2010. [Cited: 4 May 2010.] http://en.wikipedia.org/wiki/VRML.

Woo, Dong-Min, et al. 2008. Building Detection and Reconstruction From Aerial Images. *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences.* Beijing : s.n., 2008. Vol. XXXVII Part B3b.

Xiaojing Huang, Leong Keong Kwoh. 2008. Monoplotting - A Semi-Automated Approach For 3D Reconstruction From Single Satellite Image. *The International Archives of the Photogrammetry, Remote Sensing, and Spatial Information Sciences.* Beijing : s.n., 2008. Vol. XXXVII, Part B3b, pp. 735-741.

Appendix 1 - Parameters of the algorithm for each image

image	canny	lines	diff_avg	diff_dev	Rectify	thresh_area	sigmaMax	DiffColour	shadeArea
img2	400	10	17	14	true	600	26	0	0.02
img3	500	10	17	14	true	600	35	-10	0.03
img4	400	10	17	14	true	600	26	0	0.02
img4-1	400	10	17	14	false	600	26	0	0.02
img4-2	400	10	17	14	true	600	26	0	0.02
img4-3	400	10	17	14	true	600	26	0	0.02
img5	400	10	17	14	true	600	26	0	0.02
img6	500	10	17	14	true	600	26	0	0.02
img7-1	500	10	50	50	false	600	35	0	0.18
img8	400	10	17	14	true	600	35	0	0.05
img9	500	15	17	14	true	1500	26	-10	0.02
img10	400	10	50	50	false	600	35	-10	0.05
img11	450	15	50	50	true	600	35	0	0.05
img18-1	350	10	50	50	false	600	35	10	0.04
img19	650	20	50	50	false	600	40	0	0.05
img19-1	500	20	50	50	false	600	40	0	0.05
img22	600	15	17	14	false	600	35	-10	0.21
img23	600	15	17	14	false	600	35	-10	0.22

Appendix 2 - Chart of the results for images of industrial areas

image	number of buildings or group of buildings	number of detected buildings	% of detected buildings	number of false positive	% of false positive
img4	16	10	62,50	1	6,25
img4-2	3	2	66,67	2	66,67
img4-3	3	3	100,00	0	0,00
img5	10	7	70,00	3	30,00
img6	10	6	60,00	1	10,00
img7-1	7	5	71,43	0	0,00
img9	4	2,5	62,50	0	0,00
img10	1	0,75	75,00	0	0,00
img21	7	4,5	64,29	7	100,00
img22	1	1	100,00	1	100,00
img25	13	6	46,15	0	0,00
		average	70,78	average	28,45
		standard deviation	15,51	standard deviation	38,83

Appendix 3 – Results for each image

Image 2

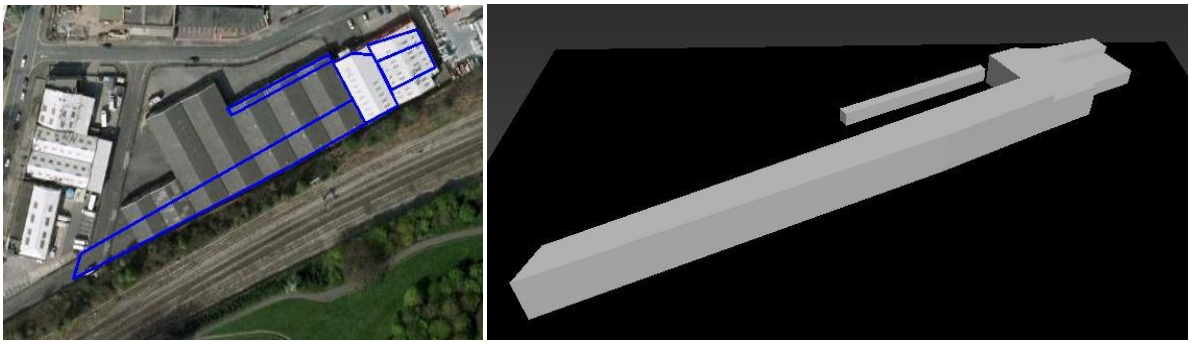


Image 3

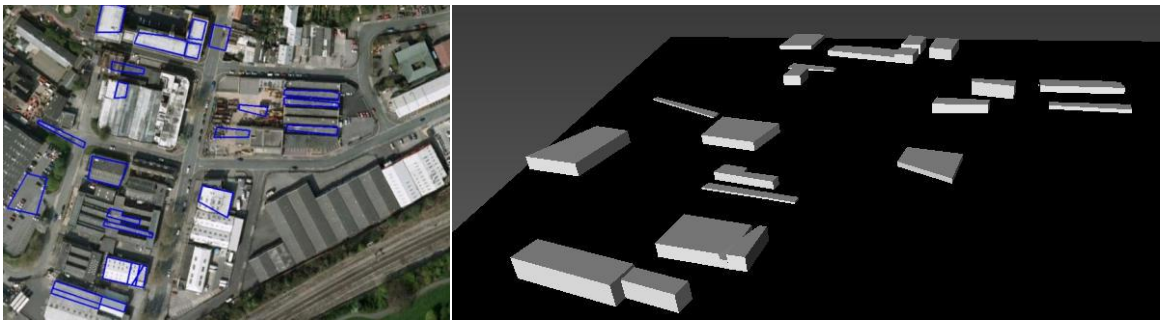


Image 4

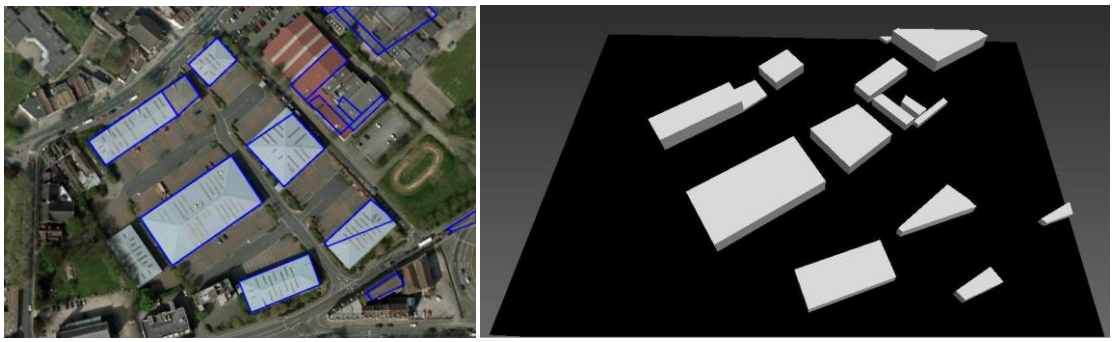


Image 4-1 (detail of image 4)

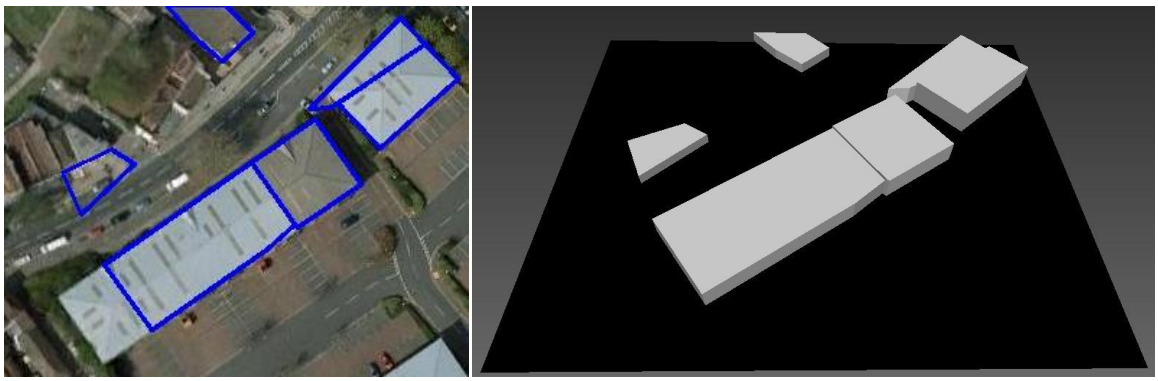


Image 4-2 (detail of image 4)

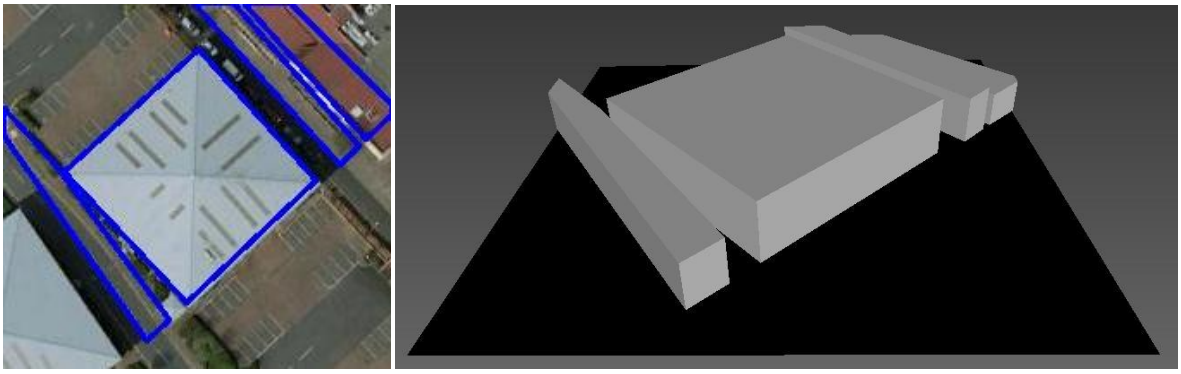


Image 4-3 (detail of image 4)

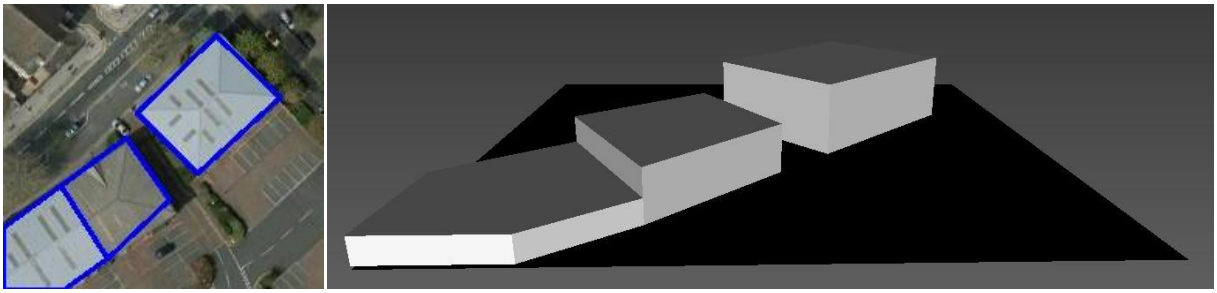
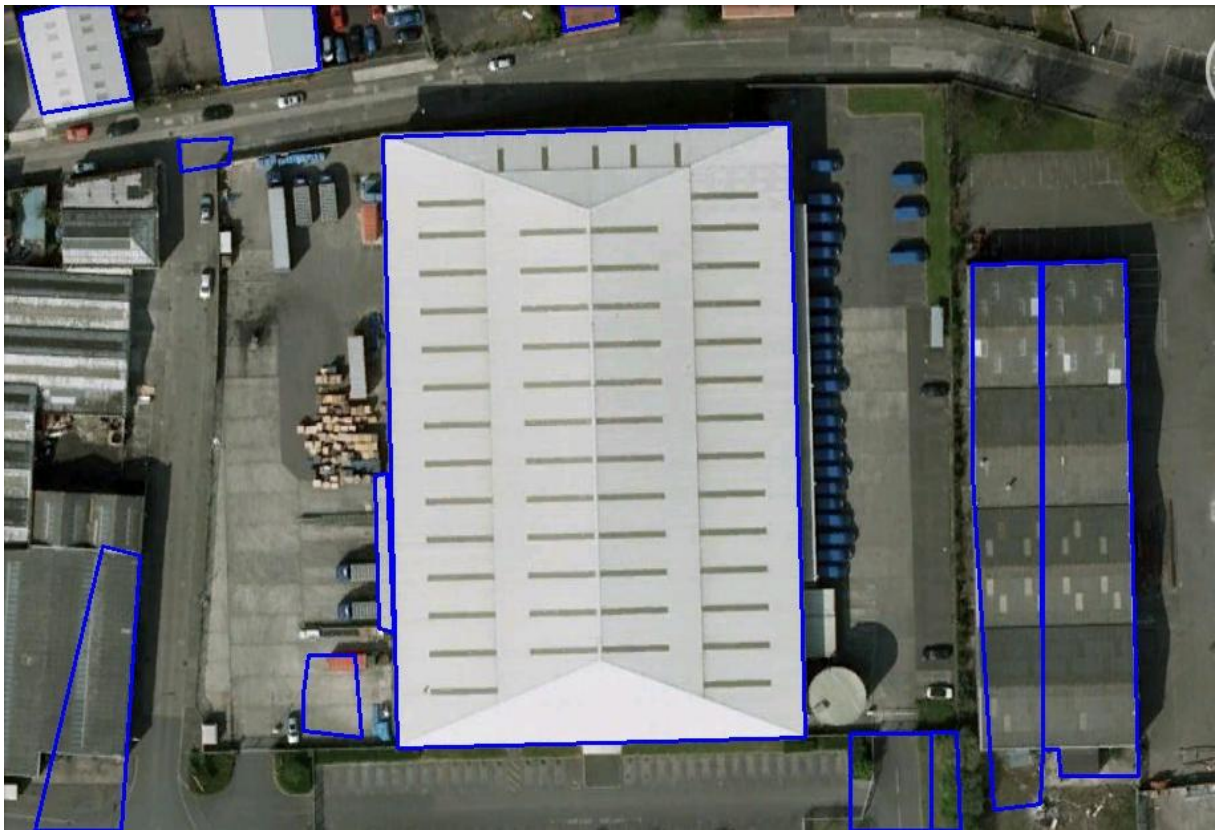


Image 5



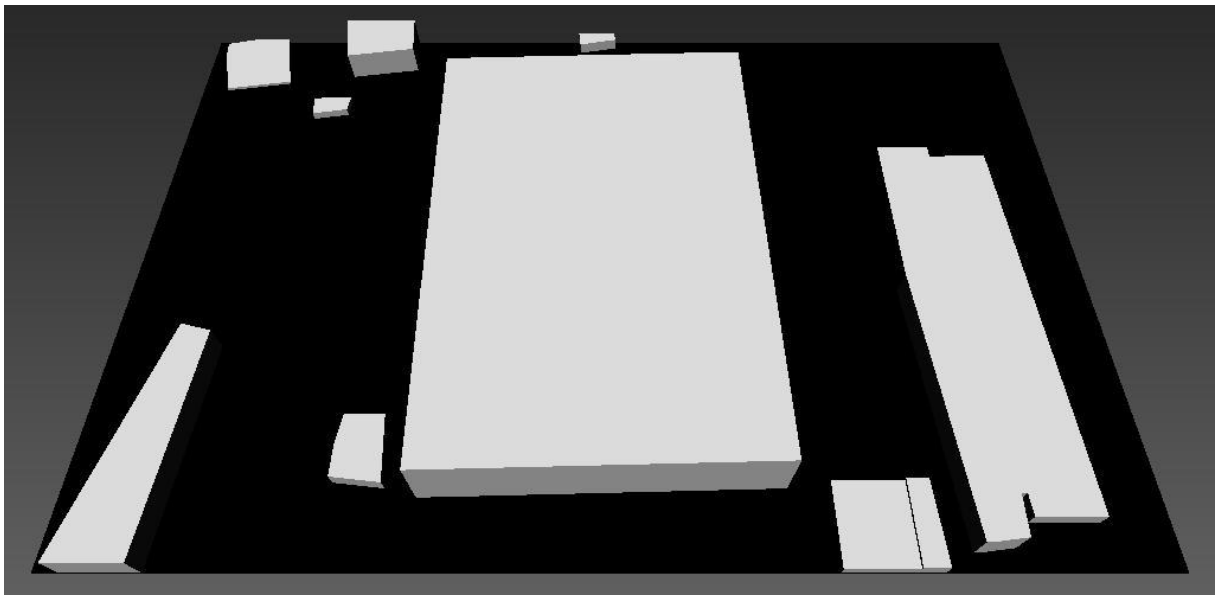


Image 6

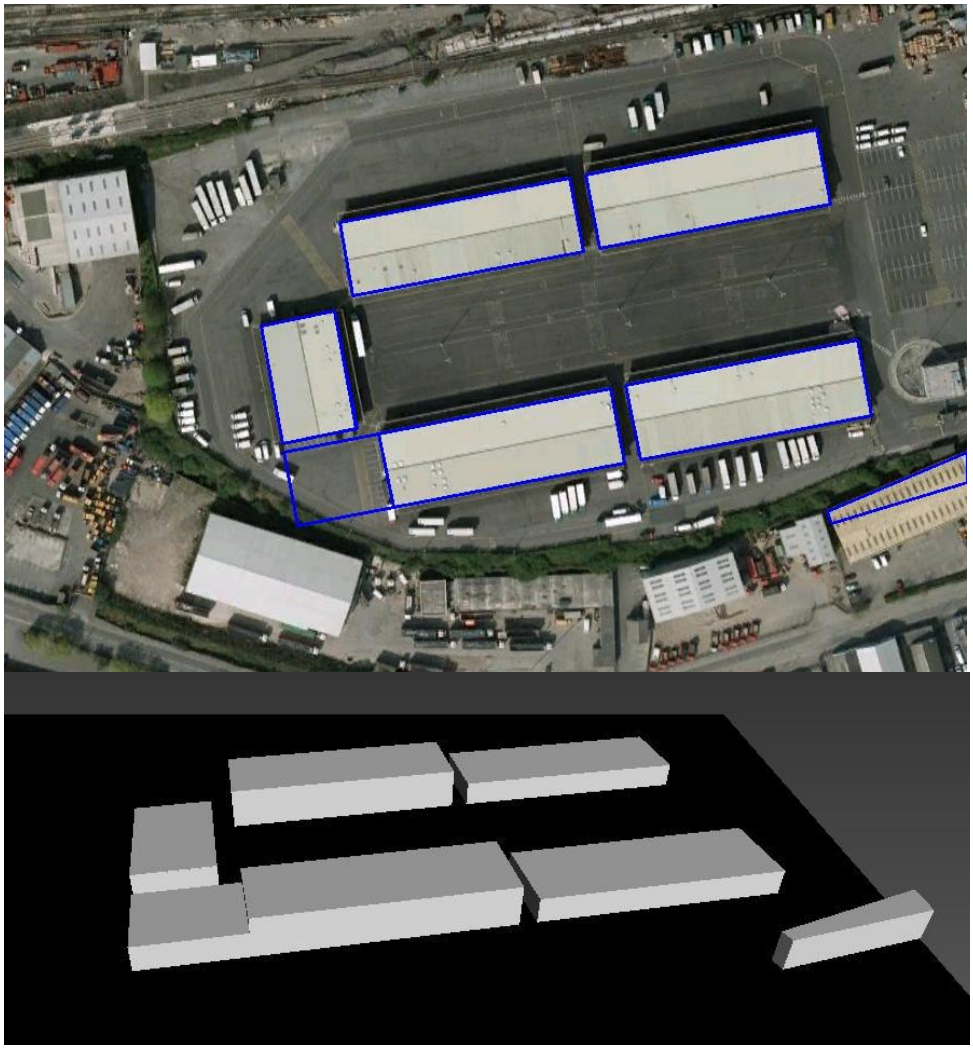


Image 7-1

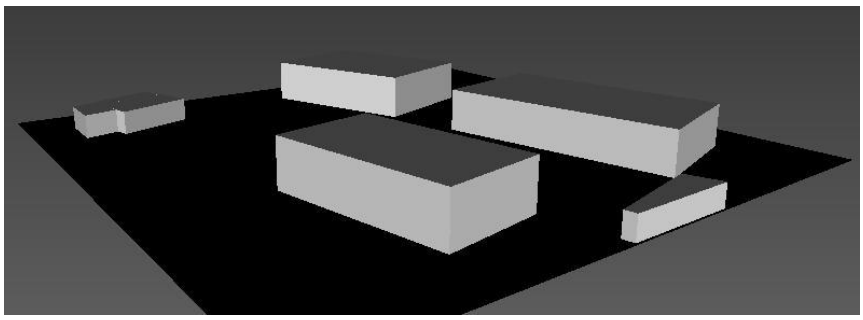
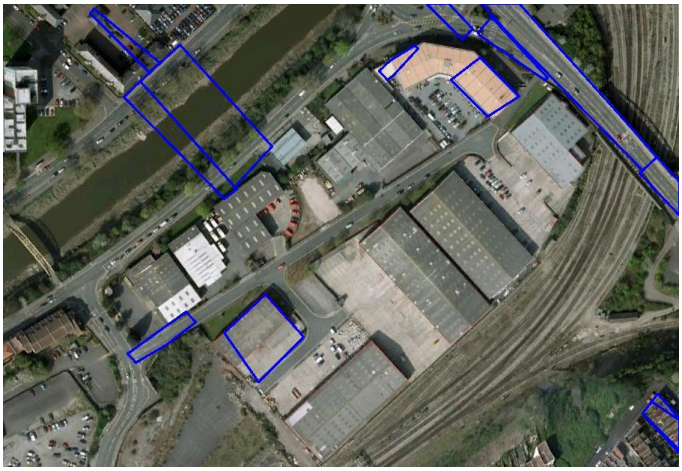


Image 8



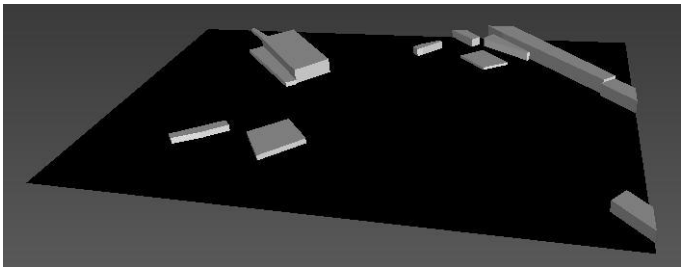
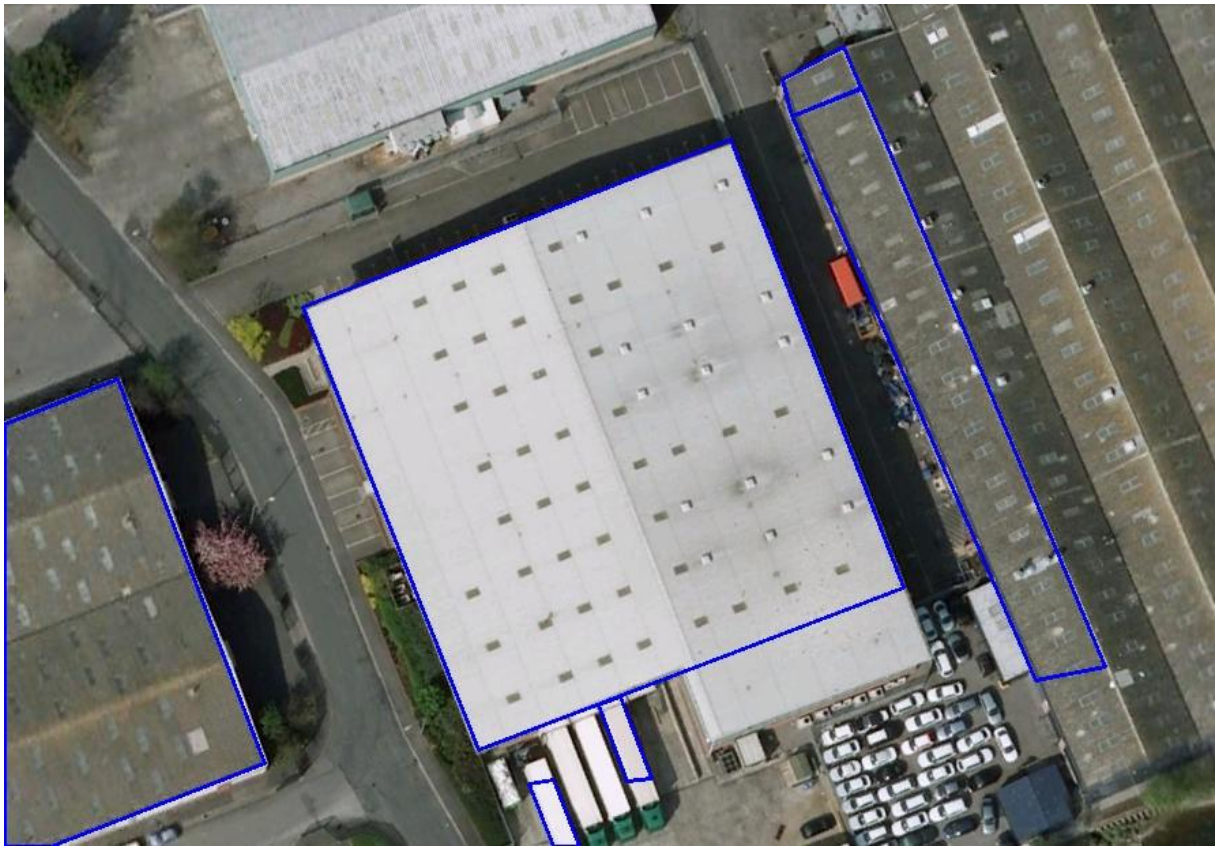


Image 9



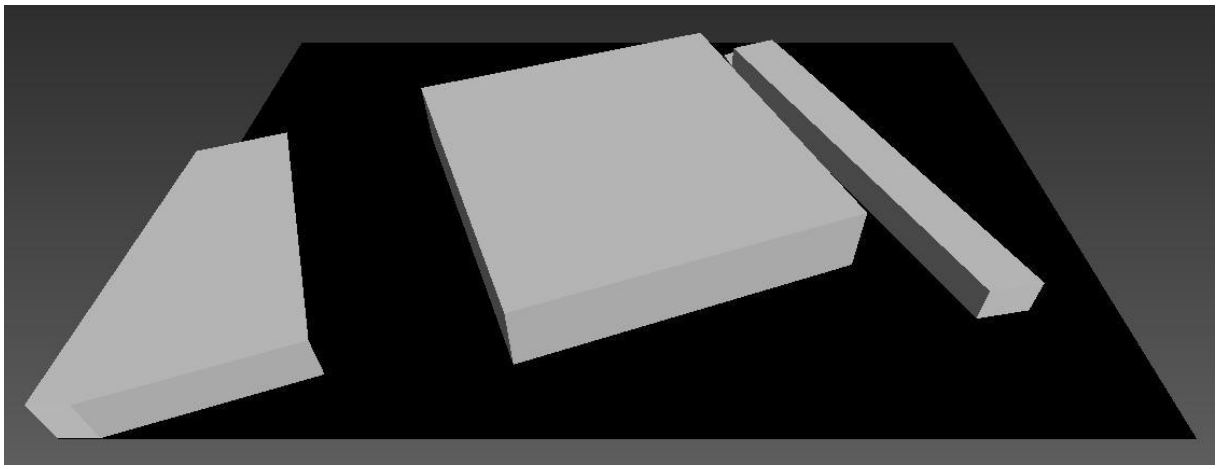
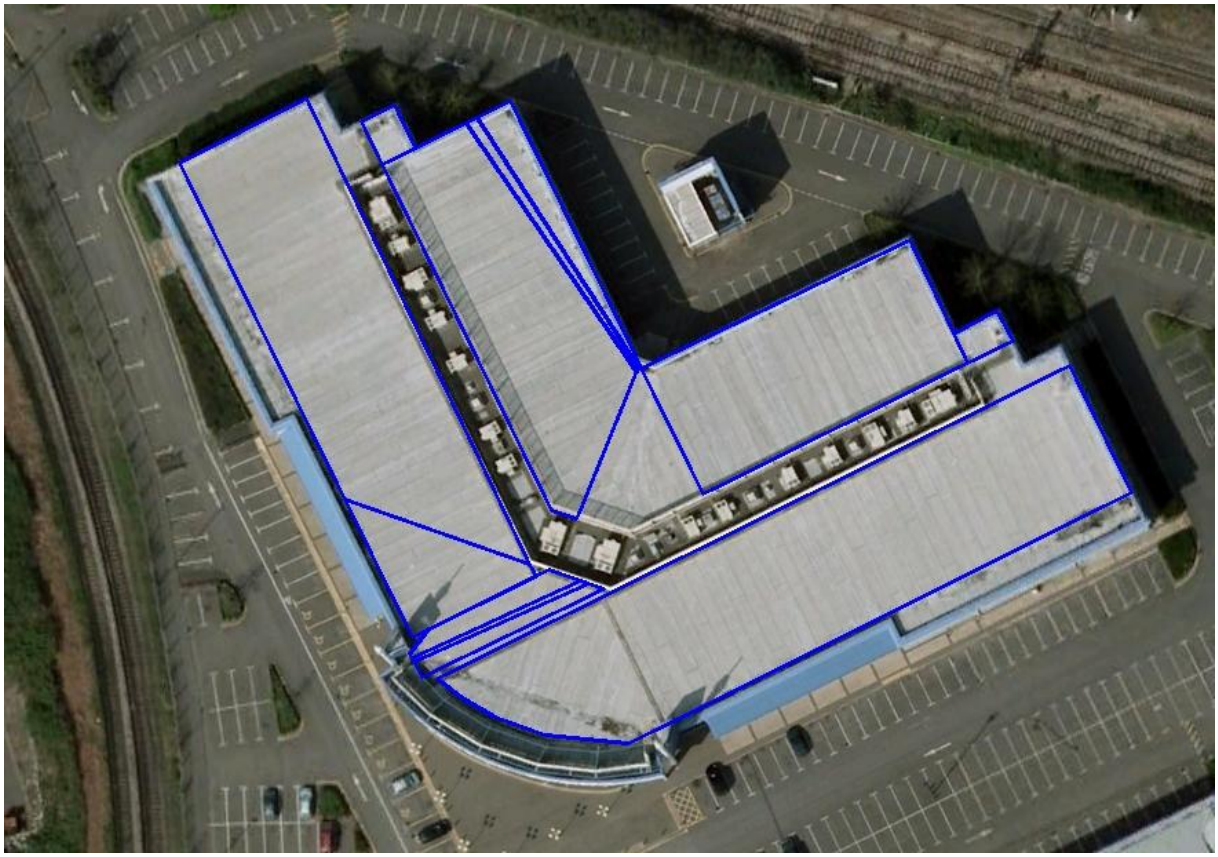


Image 10



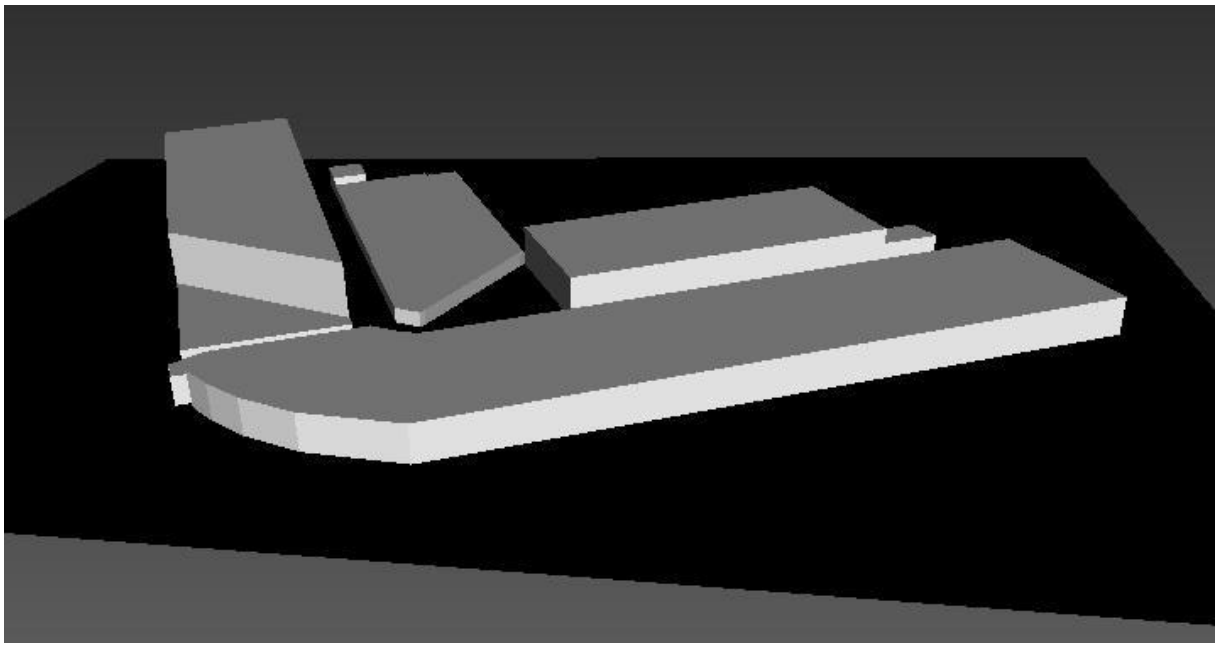


Image 11

Image 18-1

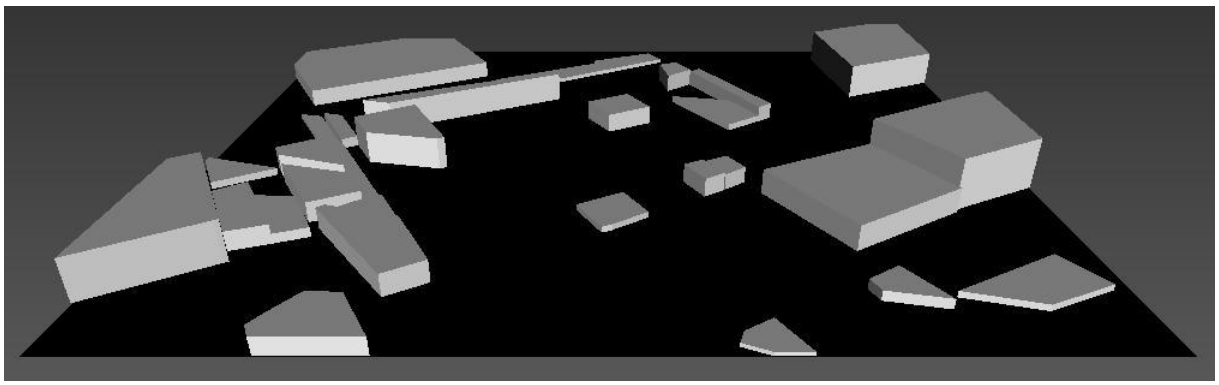


Image 19

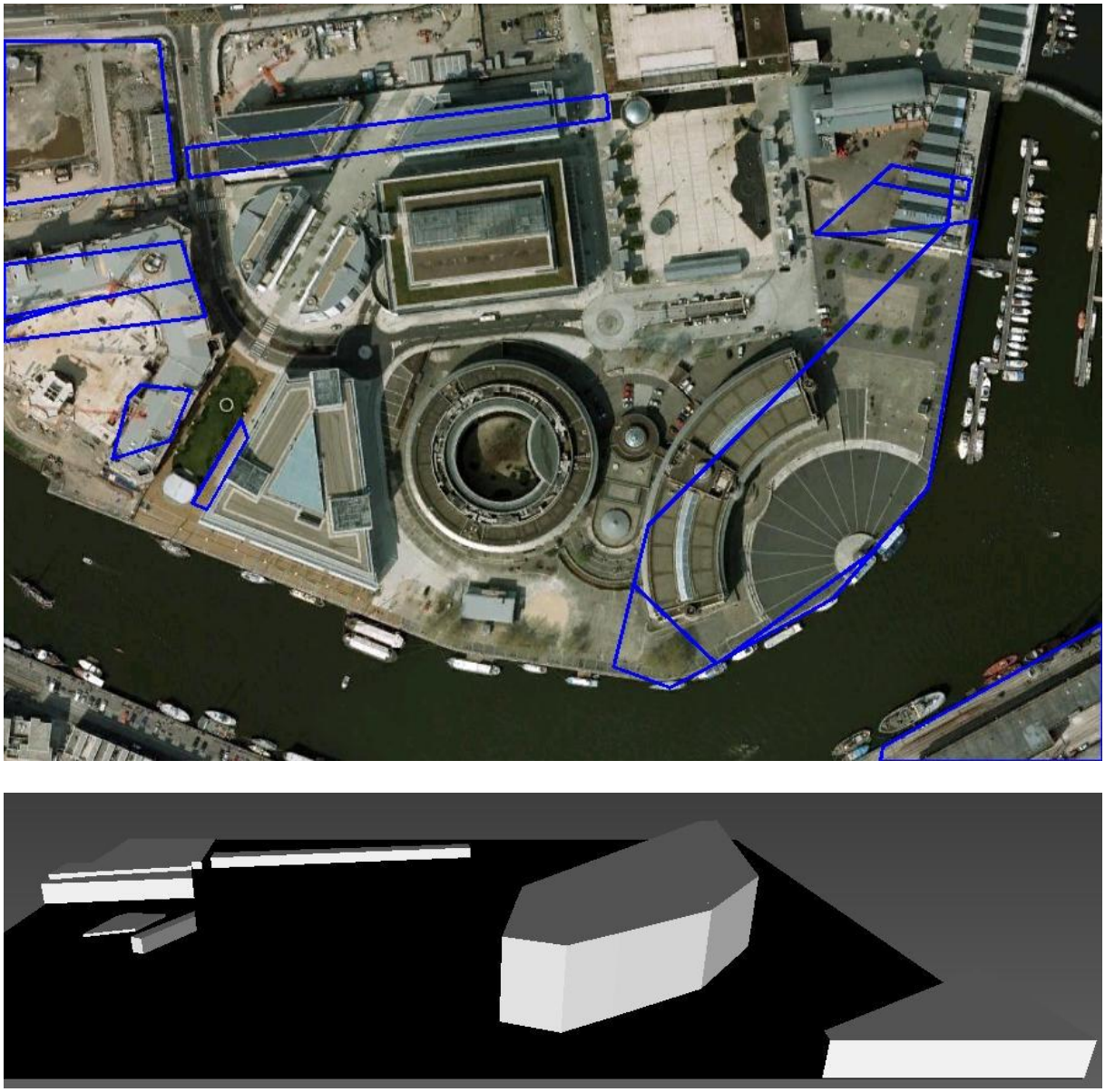


Image 19-1

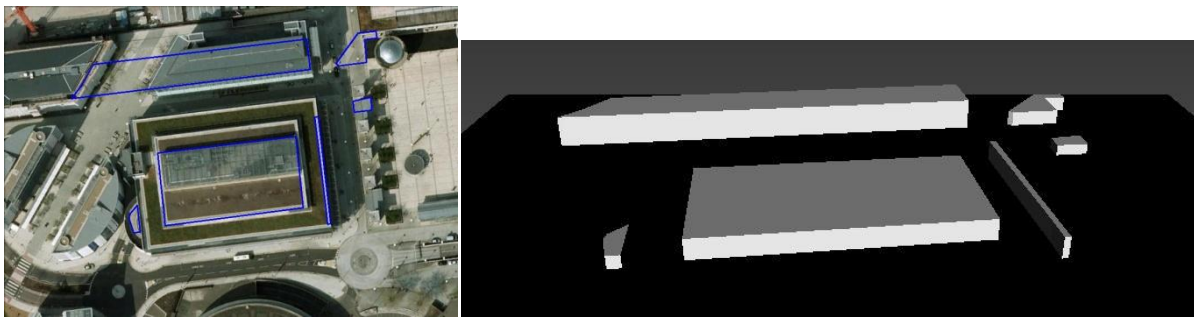


Image 21

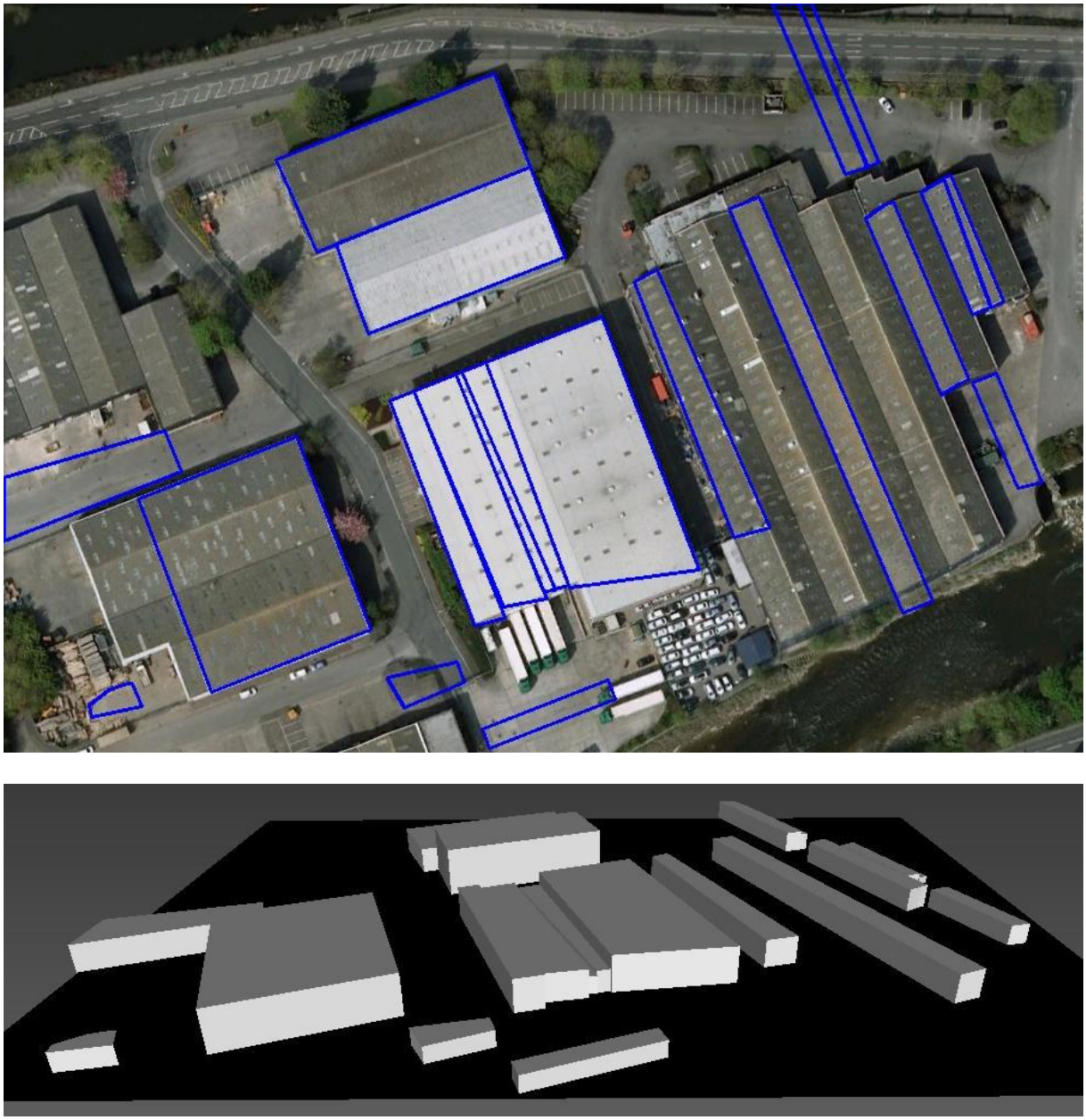


Image 22

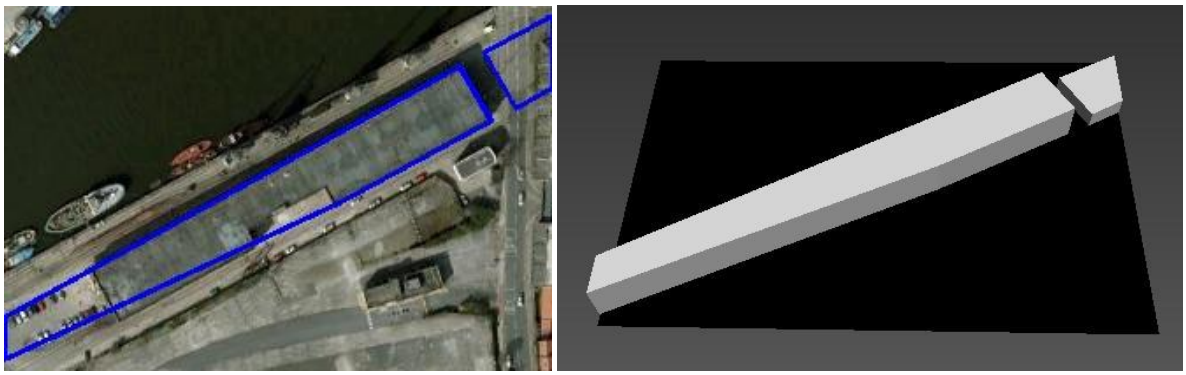


Image 23

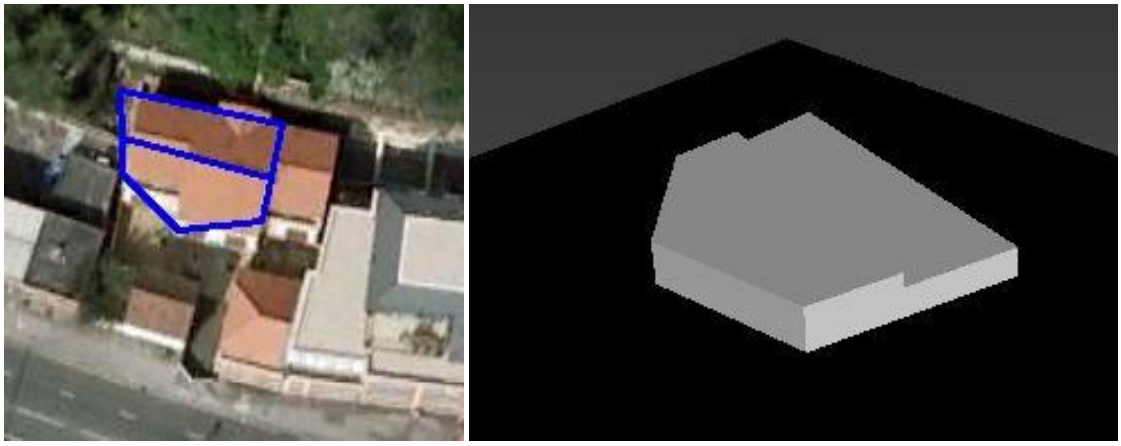
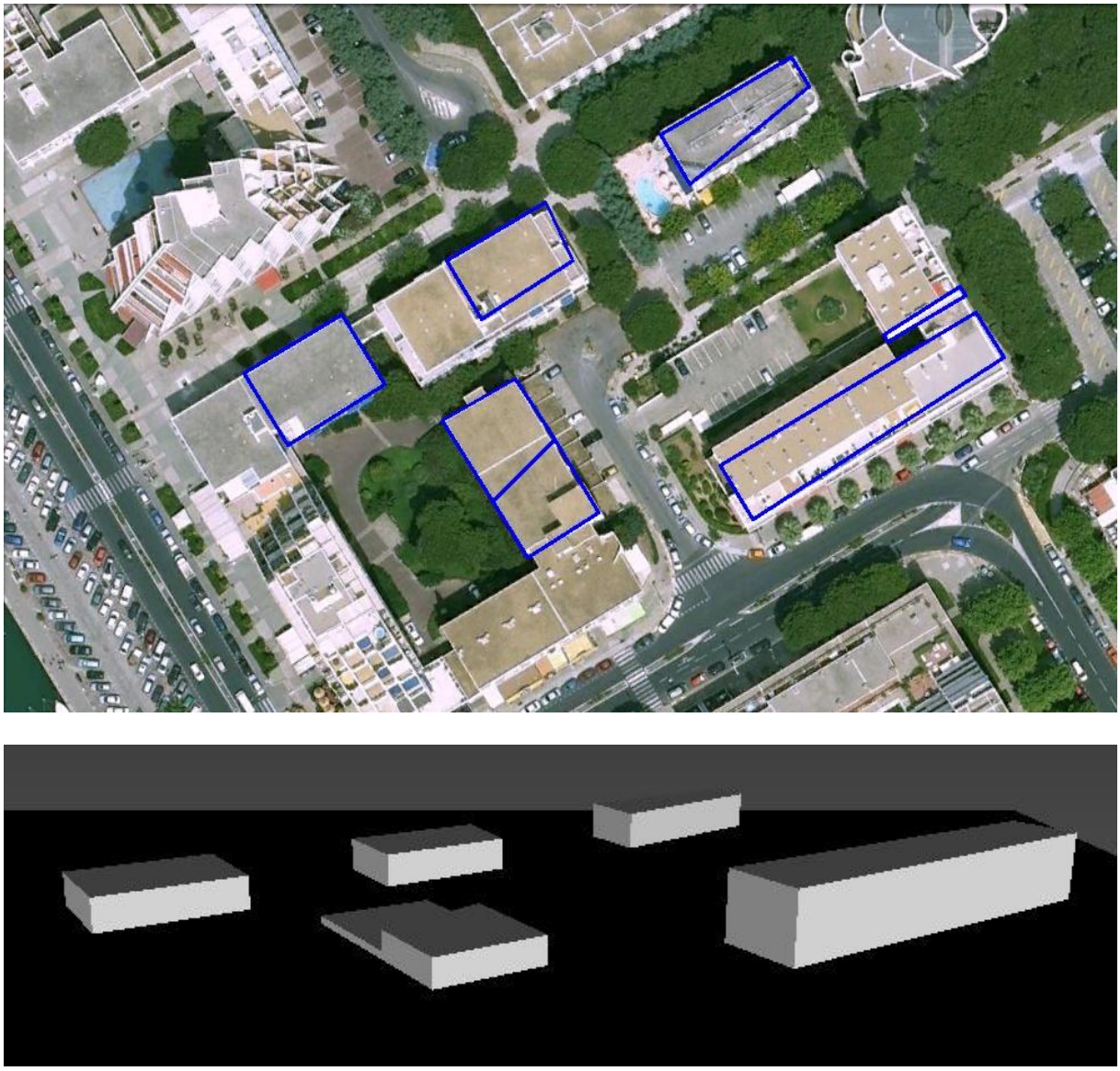


Image 25



Appendix 4 – Source code

Part 1

The code in the next pages is the code of the main, that shows the different steps of the algorithm.

```
#include "stdafx.h"

using namespace std;

vector<PathDir> Node::paths;

int _tmain(int argc, _TCHAR* argv[])
{
// parameters for the algorithm, to be adjusted for each image
int thresh_canny = 400;
int thresh_hough = 10;
bool rectify = false;
int diff_avg = 17;
int diff_dev = 14;
int thresh_sigma = 26;
int thresh_area = 1500;
int thresh_diff = -10;
float thresh_shade = 0.02;

cout << "Please give the name of the image" << endl;
string imgName;
cin >> imgName;
//directory where the original images are
string directory =
"C:/Users/Chloé/Documents/Bristol/Project/C++/images/Etapes/0 -
test/";
//directory to save the images (a folder with the name of the image
has to have been created there)
string saveDir = "C:/Users/Chloé/Documents/Bristol/Project/C++/" +
imgName;
string path = directory + imgName + ".jpg";
cout << "Display the output images ? Y/N" << endl;
string dis;
bool display;
cin >> dis;
if (dis.compare("Y") == 0 || dis.compare("y") == 0) display = true;
else display = false;
cout << "Save the output files and images ? Y/N" << endl;
string sav;
bool save;
cin >> sav;
if (sav.compare("Y") == 0 || sav.compare("y") == 0) save = true;
else save = false;

Image img = cvLoadImage(path.c_str());
Image imgGray = img.convertToGray();

//edge detection step
//canny algorithm
string sCanny = "/0 - Canny.jpg";
Image imgCanny = img.canny(200,thresh_canny);
if (save == true)
        imgCanny.save((saveDir + sCanny).c_str());
if (display == true)
        imgCanny.display("Canny algorithm");
cout << "Canny ok" << endl;

//Hough algorithm
CvSeq* lines = imgCanny.lines(20,thresh_hough,5);

imgCanny.release();

VToken vToken = VToken(lines,imgGray);
cvReleaseMemStorage(&(lines->storage));
if(save == true)
{
        string sTokens = "/1 - Tokens.txt";
        vToken.Print((saveDir + sTokens).c_str());
}
Image imgTok = vToken.draw(img);
if(save == true)
{
        string sTokensImg = "/1 - Tokens.jpg";
        imgTok.save((saveDir + sTokensImg).c_str());
}
if (display == true)
        imgTok.display("Original Tokens");
imgTok.release();
cout << "Hough algorithm ok" << endl;

//graph creation
Graph* graph = vToken.createGraph(imgGray);
VToken* vU = new VToken();

//iterations of the linking stage
cout << "Linking " << graph->getSize() << " tokens..." << endl;
int nbIter = 5;
int rInit = 30;
int rInc = 10;
```

```
for (int i=0; i <nbIter; i++)
{
	VToken* v = graph->createVToken(rInit + i * rInc);
	cout << "number of tokens before " << v->getSize() << endl;
	delete graph;

	vU = v->SetUnique((rInit + i * rInc)/8);
	cout << "number of tokens after " << vU->getSize() << endl;
	delete v;
	graph = vU->createGraph(imgGray);
	}
Image imgFin = vU->draw(img);
if (save == true)
{
	string sTokensLinked = "/2 - Linked tokens.txt";
	vU->Print((saveDir + sTokensLinked).c_str());
	string sTokensLinkedImg = "/2 - Linked Tokens.jpg";
	imgFin.save((saveDir + sTokensLinkedImg).c_str());
}

if (display == true)
{
	imgFin.display("Linked Tokens");
}
imgFin.release();
delete vU;

cout << "Linked tokens ok" << endl;

string sTokensLinked = "/2 - Linked tokens.txt";
VToken v = VToken((saveDir + sTokensLinked).c_str());

//sorting of the tokens
VToken* vSorted = 0;

//rectification of the tokens
if (rectify == true)
{
	VToken* vrectified = v.Rectify();
	Image imgvrec = img;
	imgvrec = vrectified->draw(img);
	if (display == true)
	{
		imgvrec.display("Tokens rectified");
	}
	if(save == true)
	{
		imgvrec.save((saveDir + "/2 - Tokens
rectified.jpg").c_str());
	}
	vSorted = vrectified->Sort();
}
if (rectify == false)
{
	vSorted = v.Sort();
}

cout << "Tokens sorting ok" << endl;

//binary tree creation
NodeBSP N0 = NodeBSP(img);
N0.createBSPTree(vSorted, img);
delete vSorted;
BUSes b;

//list of BUS (Building Unit Shape) creation
b.createBUSes(&N0);

Image imgb = img;
imgb = b.Draw(img);
if (save == true)
{
	string sBuses = "/3 - Buses.txt";
	b.Print((saveDir + sBuses).c_str());
	string sBusesImg = "/3 - Buses.jpg";
	imgb.save((saveDir + sBusesImg).c_str());
}
if (display == true)
	imgb.display("Original BUSes");

cout << "Original buses ok" << endl;

b.Number();

//creation of the adjacency graph
b.createAdjGraph();

//grouping of the images
BUSes b1 = b.group(img,diff_avg,diff_dev);

if (save == true)
{
	string sGroupedBuses = "/4 - Grouped buses.txt";
	b1.Print((saveDir + sGroupedBuses).c_str());
}

string sGroupedBuses = "/4 - Grouped buses.txt";
BUSes bG = BUSes((saveDir + sGroupedBuses).c_str());
Image imgGrouped = img;
imgGrouped = bG.Draw(img);
```

```
if (save == true)
{
	string sGroupedBusesImg = "/4 - Grouped buses.jpg";
	imgGrouped.save((saveDir + sGroupedBusesImg).c_str());
}
if (display == true)
	imgGrouped.display("Grouped Buses");

cout << "Grouped buses ok" << endl;
cout << "Buses selection..." << endl;

BUSes* bbat = new BUSes();

//first selection : texture homogeneity and area
bbat = bG.select(img, thresh_sigma, thresh_area);
cout << "First selection : area and homogeneous texture" << endl;
Image imgSelect = bbat->Draw(img);
if (save == true)
{
	string sSelectedBuses = "/5 - Selected Buses.txt";
	bbat->Print((saveDir + sSelectedBuses).c_str());
	string sSelectedBusesImg = "/5 - Selected Buses.jpg";
	imgSelect.save((saveDir + sSelectedBusesImg).c_str());
}
if(display == true)
	imgSelect.display("Selected Buses");

//BUSes simplification
BUSes* bSimple = bbat->Simplify();
if(save == true)
{
	string sSimpleBuses = "/6 - Simplified buses.txt";
	bSimple->Print((saveDir + sSimpleBuses).c_str());
}

cout << "Buses simplification ok" << endl;

//selection with the "stand out" criterion
BUSes* bAvg = new BUSes();
for (int i = 0; i < bSimple->getSize(); i++)
{
	Colour c = bSimple->getBuses()[i]->DiffColour(img);
	if (c.getR() <thresh_diff || c.getG() <thresh_diff ||
c.getB() <thresh_diff)
	{
		bAvg->addBus(bSimple->getBuses()[i]);
	}
}

Image Iselect = img;
Iselect = bAvg->Draw(img);
if (display == true)
	Iselect.display("Stand out buses");
if ( save == true)
{
	string sStandOutImg = "/7 - Stand out Buses.jpg";
	Iselect.save((saveDir + sStandOutImg).c_str());
	string sStandOut = "/7 - Stand out Buses.txt";
	bAvg->Print((saveDir + sStandOut).c_str());
}

cout << "Second selection : difference of colour between polygon and
surroundings" << endl;

//selection with the cast shadow criterion
BUSes finalB = BUSes();
for (int i = 0; i < bAvg->getSize(); i++)
{
	if(bAvg->getBuses()[i]->ShadeArea(imgGray) > thresh_shade)
	{
		finalB.addBus(bAvg->getBuses()[i]);
	}
}
Image final = img;
final = finalB.Draw(img);
if (display == true)
	final.display("Final BUSes");
if (save == true)
{
	string sFinalBuses = "/8 - Final buses.txt";
	finalB.Print((saveDir + sFinalBuses).c_str());
	string sFinalBusesImg = "/8 - Final buses.jpg";
	final.save((saveDir + sFinalBusesImg).c_str());
}

//creation of the VRML file
finalB.create3Dfile(img,(saveDir + "/9 - 3D.vrml").c_str());

system("PAUSE");

return 0;

}
```

Part 2

The code below is the code of the function `createVToken`, that creates a new list of linked tokens from the original tokens.

```
VToken*
Graph::createVToken(int r)
{
    VToken* v = new VToken();

    vector<PathDir> tab;
    vector<int> mark(getSize(),0);
    vector<int> WithinRadius(getSize(),0);
    list<Path>::iterator it;
    vector<int> replaced(getSize(),0);
    Path temp_path;
    int nb_path_size1 = 0;

    for (int nodenb = 0; nodenb < getSize(); nodenb++)
    {
       //initialize the WithinRadius array for endpoint 1
       for(int i = 0; i < getSize();i++)
       {
          if (getNode(nodenb)-
>getVertex().IsWithinRadius(getTokenByNumber(i),1,r))
                WithinRadius[i] = 1;
          else WithinRadius[i] = 0;
          mark[i] = 0;
        }

       //calls the findPath function for endpoint 1 of this node
       getNode(nodenb)->findPath(mark,tab,1,WithinRadius);

       //stores the retrived paths from endpoint 1
       list<Path> p1 = getNode(nodenb)->StorePaths();

       // clear the global variable
       Node::paths.clear();
       tab.clear();

        //initialize the WithinRadius array for endpoint 2
       for(int i = 0; i < getSize();i++)
       {
          if (getNode(nodenb)-
>getVertex().IsWithinRadius(getTokenByNumber(i),2,r))
                WithinRadius[i] = 1;
          else WithinRadius[i] = 0;
          mark[i] = 0;
       }

       //calls the findPath function for endpoint 2 of this node
       getNode(nodenb)->findPath(mark,tab,2,WithinRadius);

       //stores the retrived paths from endpoint 2
       list<Path> p2 = getNode(nodenb)->StorePaths();

       //create a list of unique paths from the previous sets of
       paths
       list<Path> paths = getNode(nodenb)->GetPathsCombi(p1, p2);

       if (paths.size() == 1 && paths.begin()->getSize() == 1)
       {
              nb_path_size1 ++;
              v->addToken(getNode(nodenb)->getVertex());
       }
       else
       {
              float min = paths.begin()->straightness(this);
              float temp;
              int compt = 0;
              Token t = paths.begin()->replaceByToken(this);
              for (it=paths.begin(); it!=paths.end(); ++it)
             {
                     compt++;
                     temp = it->straightness(this);
                     if (temp < min)
                     {
                            min = temp;
                            t = it->replaceByToken(this);
                     }
              }
              if (min < s_threshold)
              {
                     v->addToken(t);
              }
              else v->addToken(getNode(nodenb)->getVertex());
       }
       Node::paths.clear();
       tab.clear();
}

return v;

}
```