Bachelorarbeit im Studiengang Audiovisuelle Medien

Image Driven Procedural City Generation

vorgelegt von

Michael Schieber

am

6. April 2023

zur Erlangung des akademischen Grades eines Bachelor of Engineering

Erst-Prüfer: Prof. Dr.-Ing. Martin Fuchs

Zweit-Prüfer: Jochen Bomm

Ehrenwörtliche Erklärung

Hiermit versichere ich, Michael Schieber, ehrenwörtlich, dass ich die vorliegende Bachelorarbeit mit dem Titel: „Image Driven Procedural City Generation“ selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.

Ich habe die Bedeutung der ehrenwörtlichen Versicherung und die prüfungsrechtlichen Folgen (§26 Abs. 2 Bachelor-SPO (6 Semester), § 24 Abs. 2 Bachelor-SPO (7 Semester), § 23 Abs. 2 Master-SPO (3 Semester) bzw. § 19 Abs. 2 Master-SPO (4 Semester und berufsbegleitend) der HdM) einer unrichtigen oder unvollständigen ehrenwörtlichen Versicherung zur Kenntnis genommen.

Abstract

This work was set out to assess, if it is possible to procedurally create cities from a single 2D image of a city layout. A method has been devised, that is able to interpret an input image and create 3D data corresponding to user-defined zones. The proposed method uses a centerline approach to create road networks and placement maps to place buildings inside their designated areas. This method has then been implemented as a plugin for the open source software Blender. The plugin is able to create corresponding road networks and buildings from input images. While the resulting city models are still relatively simple, this work is a basis for further research and development on the topic of image driven city generation.

Zusammenfassung

In dieser Arbeit wurde eine Möglichkeit untersucht, welche Städte aus einem einzigen zweidimensionalen Eingabebild eines Stadtgrundrisses prozedural erstellen kann. Es wurde eine Methode entwickelt, mit welcher ein Eingabebild interpretiert und 3D-Daten erstellt werden können, welche den benutzerdefinierten Zonen des Eingabebildes entsprechen. Die vorgeschlagene Methode verwendet einen *„centerline“* Ansatz, um Straßennetze zu erstellen, und so genannte *„placement maps“*, um Gebäude in den ihnen zugewiesenen Bereichen zu platzieren. Diese Methode wurde weiterhin als Plugin für die Open-Source-Software Blender implementiert. Dieses Plugin ist in der Lage, entsprechende Straßennetze und Gebäude aus Eingabebildern zu erstellen. Während die resultierenden Stadtmodelle noch relativ simpel sind, stellt diese Arbeit eine Grundlage für weitere Forschung und Entwicklung zum Thema bildgesteuerte Stadtgenerierung dar.

Contents

1 Introduction **1**

2 Related Work **3**

2.1 Procedural Techniques 4

2.2 Procedural City Generation 5

2.2.1 Grid Layout in Undiscovered City Demo 5

2.2.2 L-Systems in CityEngine 6

2.2.3 Agent Based Simulation 7

2.2.4 Template Based Generation 7

2.2.5 Split Grammars 9

2.2.6 Citygen 9

2.2.7 Example-Driven procedural Urban Roads 11

2.2.8 Neural Networks Based City Generation 12

3 General Approach **14**

3.1 Roads 14

3.2 Buildings 18

3.3 Water and Parks 20

4 Implementation in Blender **22**

4.1 Preparing the IDE 22

4.2 Blender API 23

4.3 Plugin Overview 24

4.4 General Image Processing 25

4.5 Road Processor 25
4.6 Building Processor 33
4.7 Water and Park Processors 39
4.8 Using the finished Plugin 40

5 Evaluation **42**
5.1 Roads 42
5.2 Buildings 42

6 Limitations **44**

7 Future Work **46**

8 Conclusion **48**

9 References **49**

1 Introduction

With processing and rendering power increasing, games nowadays have the ability to display increasingly high and complex amounts of content. The creation of said content by hand is costly and time consuming to the point, that for many studios this is not feasible. To accelerate the creation of content, procedural generation methods have been in use for some time now. With these methods a smaller team of artists can create complete cities, planets or even complete universes in a fraction of the time of traditional modeling techniques.

In the case of *No Man's Sky* (Hello Games, 2016) the amount of data required to store the full universe would be taking up a great amount of space on the end users systems, while most of it would not even be explored by any one user. Instead, the parts of the universe the user visits can be procedurally generated on the fly, through a significantly smaller amount of instruction data.

Though the procedural methods are powerful, many of them are difficult to control for the layman and the resulting data is difficult to predict. Better options to control the result could save many iterations and a lot of time for the artists.

In the context of city generation, some generators produce cities through just a set of parameters, which the user can tweak, while some use real world data to either recreate the input cities or produce cities in the style of the input cities. In both cases it is difficult for the user to create road network shapes that follow an exact vision of the user.

To achieve more control in the shaping of the city, the procedural generation could be guided by a simple input image, that includes the shape of roads, lots for buildings, and water and green areas for the city to be generated.

Therefore the question this work seeks to answer is: is it possible to create an image driven city generation tool with the 3D software package Blender?

This question is attempted to be answered by implementing a method to create 3D city data from an input image as a plugin for Blender and then evaluating the results of this approach with multiple input images.

Blender was chosen as the software to be used, because of its open-source nature and therefore its availability to everyone Furthermore, it currently gains a lot of traction in the 3D community and additionally the author of this work already has experience with developing plugins for Blender.

The remainder of this work includes an overview over related work to the topic of procedural city generation, a general method to extract 3D city data from an input image, an overview over the implementation of this general method in Blender, an evaluation of the results of this method, limitations of the approach, possible future work and a conclusion.

2 Related Work

The usage of procedural generation to create data has been around for some time now. Kelly and McCabe (2006) state that the "key property of procedural generation is that it describes the entity, be it geometry, texture or effect, in terms of a sequence of generation instructions rather than as a static block of data".

As stated, procedural generation was not only used to create 3D models of cities or buildings, but to create a multitude of different data. Many of the techniques used for this data creation are still in use today, be it in an evolved form, or as combination of multiple techniques to create increasingly complex data.

There are multiple examples of procedural data generation. One of them is *IPG*, an incremental procedural grammar for sentence formulation (Kempen and Hoenkamp, 1987). Another example is the creation of floral ornaments as described by Wong et al. (1998) and further discussed by Gieseke et al. (2017). Procedural generation is also used for plants (Lintermann and Deussen, 1999), trees (Longay et al., 2012) (Stava et al., 2014), terrains (Smelik et al., 2009) or whole plant ecosystems (Deussen et al., 1998).

In the case of Deussen et al. (1998) multiple procedural modeling techniques are used together to first create a terrain with perlin noise, add plant densities and attributes through simulation and then fill the terrain with procedurally generated plants.

2.1 Procedural Techniques

Procedural generation uses several techniques to create data. Some of the longest used techniques include Fractals, Perlin Noise, Voronoi diagrams and *L-Systems*.

Fractal shapes are often used to imitate natural shapes, since the relatively simple instructions can create intricate nature-like results, such as terrain or clouds (Mandelbrot, 1982). The resulting shapes are characterized by their of self similarity which is also exhibited in a lot of natural phenomena, such as coastlines or plants (e.g. ferns).

Perlin Noise is a type of noise created by Ken Perlin. It consists of several layers of noise stacked on top of each other to create "naturalistic looking textures" (Perlin, 1985). The resulting noise textures can be used to create a multitude of different textures, such as marble or water, and can also be used as displacement textures to create terrain. An advantage to regular textures is, that the procedurally generated Perlin Noise is infinitely tileable and does not repeat itself.

Voronoi Diagrams are used to create another kind of procedural textures, that are based on cells. They were first introduced by Worley (1996) and can be used to create "textured surfaces resembling flagstone-like tiled areas, organic crusty skin, crumpled paper, ice, rock, mountain ranges and craters".

L-Systems, also called Lindenmayer-systems were invented by Lindenmayer (1968). They are a type of formal grammar, that uses an alphabet, a set of symbols, a starting point and a set of rules to create complex objects through an iterative process. They can be used to create 3D models such

as plants and trees or even road networks and buildings (Parish and Müller, 2001).

2.2 Procedural City Generation

ŞAHİNOĞLU and ÇELİKCAN (2022) state that "synthetic city generation is a multifaceted issue composed of multiple subproblems such as road network generation, layouting, parceling and building generation". Different approaches to each of these problems have been explored, based on already well-known procedural methods, newer methods - such as neural networks (e.g. Nishida et al. (2018) and Kim et al. (2019)) - or any combination of these methods.

Kelly and McCabe (2006) have explored multiple city generators with multiple approaches and have evaluated them on multiple factors including, but not limited to realism, variation and control. Some of which will be introduced in the following sections.

2.2.1 Grid Layout in Undiscovered City Demo

One of the first discussed generators is the so called *Undiscovered City Demo* (Greuter et al., 2003). It is based on a regular grid as road structure and then places buildings on each block in the pattern. Buildings itself are based on geometric primitives with the position in the grid dictating the final shape of each building. Kelly and McCabe (2006) have rated this approach as not very realistic with little variation and control.

2.2.2 L-Systems in CityEngine

Another generator covered is the *CityEngine*, presented by Parish and Müller (2001) at SIGGRAPH 2001. This city generator uses so called *Self-sensitive L-Systems* to construct road networks. Buildings are then generated through *L-Systems* as well. The *CityEngine* takes multiple image maps as input, such as geographical information, like elevation, vegetation and water boundaries, but can also use further input information like population density, land usage, street patterns and maximum building height.

The generator then creates the road network while adhering to a defined rule set, e.g. creating streets that follow the path of least elevation, and taking the input data into account.

Once the road network is created, the buildings are constructed in multiple stages. Firstly, building allotments are created by taking blocks out of the road network and then subdividing them into possible building allotments. Allotments with no street access are discarded, just as allotments which are too small. Once the building allotments are determined, buildings of multiple styles are incrementally created in these lots with a *parametric L-System* out of the buildings bounding box. Each of these styles uses its own set of *L-System* instructions.

Kelly and McCabe (2006) have rated the *CityEngine*'s result as being quite realistic, with good variation which is limited by the finite amount of building styles that can be created. Control has been limited to the input maps the user provides and the iterations of the *L-System* which the user can determine.

2.2.3 Agent Based Simulation

The *Agent Based Simulation* in *CityBuilder* (Lechner et al., 2003) simulates the growth of a city over time with different agents.

Each agent is representative of a certain role used in real life city planning. Road agents extend and connect roads in the road network, while developer agents allocate locations for buildings according to their role. For example, industrial developer agents prefer areas with good road access, while residential developer agents prefer areas with less traffic. Each decision made by a single agent is proposed to a "city council" that decides if the decision is implemented in the city. Ultimately no actual buildings are created on these locations though.

Kelly and McCabe (2006) have deemed the resulting road network and building distribution as fairly realistic, but the process can only generate relatively small scale cities and the generation is time consuming due to the evolution over time. They propose that this approach might be better suited for simulations than 3D content creation.

2.2.4 Template Based Generation

Sun et al. (2002) have proposed *Template-Based Generation of Road Networks for City modeling*. This system takes a set of image maps and applies a network template to the given geographic maps. Required input maps are a map that contains land, water and vegetation, a height map and a population density map, which is only required for the population-based template. The templates applied are a radial mode, a raster, or grid-like mode and

a mixed mode of the raster and radial mode. Furthermore, a population based voronoi template can be created. The voronoi diagram required for this approach is created by sampling the population density map and using the extracted points as the input sites for the voronoi diagram. The edges of the voronoi cells then symbolize the roads in the template.

Regardless of the used template, the desired road patterns have to be created according to the practical constraints. This means that the created roads will try to follow the given pattern, but also take elevation and areas that are not suitable for roads, like water, into account.

The *Template Based Generation* does only create rather low detail road networks, most usable as primary roads, and no futher detailed secondary roads, nor any building placement or geometry data.

According to Kelly and McCabe (2006), this technique can create road networks, that are found in real world cities, but "do not achieve the complexity and scale of real city networks". The networks themselves are mostly useful as primary road networks, with the more detailed secondary roads missing. Variation is limited by the only three possible templates and only one mixed mode. Real world cities normally mix multiple road network styles throughout the city, depending on the year of construction, the function and geographical constraints of each of the city's zones. Regarding the control of the approach Kelly and McCabe (2006) state, that the user has no real control over the road network generation, other than the input maps and pattern used. They state that this "would imply that this solution is very rigid and inflexible".

2.2.5 Split Grammars

Wonka et al. (2003) have presented a type of building generation with a new type of grammar called *split grammars*.

This type of grammar is similar but different to *L-Systems*, based on shape grammars (Stiny, 1980). It uses shapes, rather than letters or symbols.

A initial starting geometry, like a bounding box, is transformed in an iterative process with rules, such as splitting buildings into faces, faces into structural sections and then structural sections into components like windows, doors or ornaments.

To create multiple styles of buildings different rules can be added to a database.

Kelly and McCabe (2006) rate this approach to building creation as being able to create "very realistic buildings". A good variation of building styles can be achieved but the creation of rules is relatively complex and "requires a level of expertise", and is therefore not easy to control.

2.2.6 Citygen

In their survey Kelly and McCabe (2006) propose a new real time system to generate cities that incorporates a combination of some of the techniques described. The system is called *Citygen* and is described further in their paper *Citygen: An Interactive System for Procedural City Generation* (Kelly and McCabe, 2007).

The generation process is divided into three high level steps. The creation of primary roads, the generation of secondary roads and the building

allotment and creation.

Kelly and McCabe (2007) describe primary roads as "the main traffic flow arteries of the city whose function it is to transport people around the city and from one district to another (e.g. main roads, motorways etc)."

In *Citygen* the primary roads can be created by the user, by adding nodes onto the terrain. The system then procedurally generates roads between the placed nodes, which adhere to the underlying terrain. Multiple procedural road generation options have been explored, which take the terrain data and elevation changes into account.

After the creation of primary roads, secondary roads are created. They are described as "the roads inside the areas enclosed by primary roads and their function is to service districts" (Kelly and McCabe, 2007).

This process in turn is divided into three smaller parts. First the *City Cells* are extracted from the primary road network graph. City Cells are described as "the enclosed regions of the primary road network" (Kelly and McCabe, 2007). Once the City Cells are found the secondary roads are grown with the use of *L-Systems*. Roads grow inwards from the boundaries of the City Cells. For each Cell the created secondary road network pattern can be influenced by a set of parameters. Preset configurations of these parameters are provided, which can generate a raster, an industrial and an organic pattern.

In each growth cycle the newly proposed roads are checked against the already existing roads in the City Cell by a snapping algorithm. This algorithm snaps roads together, if they are close to each other and creates

intersections in the resulting roads.

With the secondary road network complete, the buildings are able to be placed in the city. This process, again, contains multiple sub-steps.

First, blocks are retrieved. These are described as "enclosed regions of the secondary road network" (Kelly and McCabe, 2007). Besides being the container for the buildings, blocks are also used to add footpaths to the city, at the perimeter of each block.

The found blocks then get subdivided into lots, where one lot is a container for one individual building. Lots without street access are discarded for building use, and rather used for green space.

On each building lot a building is then created, which adheres to a control parameter for each neighborhood. This means, that multiple types of areas can be created, such as downtown, industrial or suburban areas. The buildings themselves are very basic. They are created by extruding the footprint of a building in the lot upward and then adding textures with normal maps for extra detail.

2.2.7 Example-Driven procedural Urban Roads

In *Example-Driven Procedural Urban Roads* Nishida et al. (2015) describe a system to create large-scale road networks from real world road network examples through a mix of example based growth and procedural growth.

The system uses a user-given input map, which describes where the road network can be created as well as geographical features, such as mountains or water areas. Furthermore the user can provide example road networks in

the open street map (OSM) data format, from real world cities.

From the provided examples patches of the road network are extracted, as well as statistical parameters, which can then be used for procedural growth later.

Once the examples are analyzed, procedural city growth takes place in an incremental manner from a user-given start point on the map. From this point patches are set into the map as long as they fit. Should no valid patch fit, then the next increment is created with procedural growth. The procedural growth adapts to local constraints, such as the underlying terrain (e.g. mountains or water) and already created roads.

This process is first used for the primary roads until the user-specified area is filled. Once this is complete, the secondary roads, or "local roads" are created in a similar manner, where the start points are either intersections which coincide with real intersections in the used patches, or newly created intersections in the procedurally created roads, which get subdivided following the statistics of the example cities.

Once all these steps have been completed the user has the option to use post process features, such as an iterative process to remove all dangling roads from the resulting road network.

2.2.8 Neural Networks Based City Generation

In recent years neural networks have also been used to complement the traditional methods of procedural generation. Nishida et al. (2018) have created a system to create procedural buildings out of an example image of

an existing building. The system works by first selecting the silhouette of the building in the input image and then using convolutional neural networks (CNN) to create a grammar.

The generated grammar can then create a multitude of different buildings in a similar style to the given example building.

The technique of analyzing input data to automatically create the rules of a grammar is called *inverse procedural modeling*. The advantage of this process lies in the fact, that the creation of grammars to create different styles of procedural models is time consuming and requires expertise. With *inverse procedural modeling* any user can create grammars for a multitude of different styles by simply providing examples of the desired styles.

Another example of neural networks in city generation is *CityCraft: 3D virtual city creation from a single image* (Kim et al., 2019). This system can create a city in the style of a given input image of a street. It uses general adversarial networks (GAN) to create a terrain and CNNs to extract the city style from the input picture. The system then uses a combination of procedural and inverse procedural modeling techniques to create a road network and buildings according to the style that was retrieved through the CNNs.

3 General Approach

This work presents a method to convert input images into a three-dimensional representation of a city. The steps of this method can be used in a general manner, regardless of the software chosen in which they are implemented. As part of this work, the method has been implemented as a python plugin for the open-source software Blender (Blender Foundation, 2022).

The general approach to creating a three-dimensional City from an input image starts with the creation of said input image. This image includes multiple components of a city, which are represented by individual colors. Once the input image is created it is separated into multiple black and white images corresponding to their city components. Each of these images is then processed into three-dimensional data with multiple techniques. Which techniques are applicable is depending on the individual component. In this work, the main focus are the roads and buildings of the city.

3.1 Roads

The black and white image representing the *road network* of the city needs to be processed further before being converted into three-dimensional data.

The objective of this processing is to get data structures representing the *intersections* inside a *road network* and all the *roads* connecting to the *intersections*. The first step for this is to get the contours of all *road networks* in the image.

Per *road network* there are two types of contours. A shell, a contour that surrounds the complete road network, and one or multiple holes inside

the shell. The holes represent other city components, such as building lots, water or parks.

Road Network Creation

With the shell and holes together it is possible to calculate a *centerline*. This is a sequence of small segments which form lines in the center of each road in the input image. To achieve this, a voronoi diagram can be used (Chen et al., 2022).

The *centerline* is an unsorted sequence of segments, which need to be further processed. To find out where these segments form *Roads* and *intersections*, each segment needs to be annotated with its neighbors. A segment is represented by two points, a start point and an end point. If the start point or end point of another segment coincides with the start point or end point of the original segment, these two segments are adjacent to each other and are neighbors.

Once each segment has been marked with its neighbors, the *intersections* and *roads* can be determined. Each segment that has multiple neighbors at the start point or end point constitutes an *intersection* at this point. Each *intersection* is marked with the adjacent segments. A *road* is a sequence of segments that starts with an *intersection* and ends with either another *intersection* or a segment, that has no further neighbors.

To calculate a *road*, one adjacent segment of an *intersection* is taken. One of the points of this segment is coinciding with the *intersection*. Consequently, the neighbor of other point of the segment is the next part of the

road. Then again, the end point neighbor of this next segment is the third part of this *road*.

The *road* is traversed in this manner until the end point of a segment either is an *intersection*, or has no more neighbors and therefore constitutes a *dead-end*.

This traversal is repeated for all adjacent segments of all *intersections* in the network. If an adjacent segment of an *intersection* is already part of a road, then this would constitute a duplicate road and will not be traversed.

Road Network Refinement

Through the *centerline* creation with the voronoi diagram it is possible that the calculated *road network* contains certain artifacts. The network can contain very short *roads* with a dead-end (*short dead-ends*). These often, but not exclusively, appear in a split shape at the end of *roads* in the network, or at corners with high angles.

Another artifact that often appears are two *intersections*, that are very close to each other (*duplicate intersections*).

Short dead-ends can be identified by first calculating the length of all *roads*. To do this, the length of each segment in the *road* is calculated and added up. The *short dead-ends* are now every *road* below a given length and with only one *intersection* at either the start or end point of the *road*, and no *intersection* at the opposite end. All *short dead-ends* found by this metric can be deleted. The removal of these *roads* leaves other artifacts behind, which have to be cleaned up. These artifacts are *intersections* with

only one or two, or possibly no adjacent *roads* (*invalid intersections*). *Invalid intersections* with no, or one adjacent *road* can be safely removed. At *invalid intersections* with two adjacent *roads*, the adjacent *roads* have to be merged together after the *invalid intersection* is deleted.

It is possible that after the cleanup there will be another set of *short dead-ends*. So the process of removing the *short dead-ends*, removing *invalid intersections* and merging *roads* has to be repeated, until no more *short dead-ends* are found in the *road network*.

Duplicate intersections can be detected in a similar way to *short dead-ends*. A *road*, that is shorter than a given length and has an *intersection* on both ends constitutes that both of these *intersections* are *duplicate intersections*.

The *road* in between the *duplicate intersections* can be removed. Both *duplicate intersections* can be removed and instead a new *intersection* can be created at the middle point between the *duplicate intersections*. The end points of all adjacent *roads* of the *duplicate intersections* can also be set to the coordinate of the new *intersection*.

Road Creation in 3D Software

After removing both of these artifacts from the *road network*, the *roads* can be transferred into a 3D software. Each point on the segments of the *roads* is added to a spline inside the software. Then a 3D mesh of a short road piece can be distributed along each spline. A mesh representing an *intersection* can be created at each *intersection* point in the *road network*. The

specific implementation of these processes differentiates between different 3D software packages. An implementation in Blender with the Blender API is covered in Implementation in Blender.

3.2 Buildings

The process to place buildings into the user-given areas of the picture is based on *Organized Order in Ornamentation* (Gieseke et al., 2017). Similar as for the *road network*, a black and white image containing all areas in the city, which were marked for buildings is created and then further processed. This image is separated into multiple images, each containing one *lot* - one contiguous area which is marked for buildings.

To create these *lots*, the contours in the image have to be calculated. Similar to the *road* contours, these contours have to be calculated as shells and holes. A *lot* is now all parts of the image, that are inside a shell-contour but not inside any hole-contours. Once all *lot* images are created, each of the *lots* is further processed to be filled with viable points on which buildings can then be created in the 3D software.

Building Point Placement

Similar to *Organized Order in Ornamentation* Gieseke et al. (2017) several *Placement Maps* are used to determine the best points to create buildings in the *lots*. The *lot* image is used as a stencil map (*Lot Stencil Map*). Furthermore a distance map is created. This map contains the normalized inverse distance of each pixel to the edge of the *lot* (*Inverse Distance Map*).

This map ensures, that buildings are placed towards the edge of the *lot*. To make sure that no building clips through the edge of the *lot*, another stencil map is created (*Edge Stencil Map*). This map is created by calculating the absolute distance of each pixel to the edge of the *lot* and then using a threshold to set the values of all pixels with a distance value lower than the radius of a building bounding box to zero and the values of all remaining pixels to one. The last map that is created is the *Building Stencil Map*. This map is initialized with all pixels at the value of one. Once a building is placed into the lot all pixels with a distance lower than the buildings radius added to the next buildings radius will be set to zero. This ensures, that buildings cannot overlap with each other.

If desired, more *Placement Maps* with different rules can be created to ensure that buildings are placed according to those rules.

Once all the *Placement Maps* are calculated, they are multiplied with each other. In the resulting image (*Lot Placement Map*) the highest value indicates the next point a building can be placed.

Multiple sizes of buildings are placed into each *lot*, from largest size to smallest. Once no point with a value higher than zero can be found in the *Lot Placement Map* for a building of one size, the next smaller size of buildings will be placed. When this happens the *Edge Stencil* and *Building Stencil Maps* have to be updated with the new smaller building radius.

Buildings are placed into the *lot* until no more buildings of the smallest size can be placed into the *lot*.

This technique of placing buildings into a *lot* is repeated for each *lot*,

until all *lots* are filled.

Building Creation in 3D Software

Once all *lots* are filled, the buildings are placed into the 3D software. Each *lot* has a list of buildings, which contains their position and radius (r). Inside the 3D software a bounding box with the width and length of $\frac{r}{\sqrt{2}}$ is placed for each building. Each bounding box can be rotated, so that it is facing the closest edge of the *lot* contour. The height of the bounding box can be randomly generated or possibly specified by the user through an image containing height data. The specific implementation of these processes differentiates between different 3D software packages. An implementation in Blender with the Blender API is covered in Implementation in Blender.

The bounding boxes can be filled with either preexisting 3D models of buildings or procedurally generated buildings.

3.3 Water and Parks

As part of this work only a rudimentary method to create water-like and park-like city components was explored.

As with buildings, the different areas of water-like and park-like structures are retrieved by calculating their contours from their respective black and white images. These contours are once again calculated with shells and holes.

Once these contours are calculated, all points on a shell-contour can be input into the 3D software. Together these form one planar polygon. All

hole-contours inside the shell are also created as planar polygons.

All hole-polygons are then subtracted from the shell polygon with a boolean operation. The resulting mesh represents the surface of a water-like or park-like structure.

The specific implementation of these processes differentiates between different 3D software packages. An implementation in Blender with the Blender API is covered in Implementation in Blender.

4 Implementation in Blender

As part of this work the method discussed in General Approach is implemented in the open-source 3D software package Blender (Blender Foundation, 2022). The implementation uses the Blender version 3.4.0, which includes a python API (Application Programming Interface) called *bpy*, using the python version 3.10.8. The implemented plugin is called *pictureCity*.

4.1 Preparing the IDE

The integrated development environment (IDE) used is Microsofts Visual Studio Code citepVSCode with the extensions Python, Pylance and the *Blender Developement Extension* by Jaques Lucke. This extension enables Visual Studio Code to build the written plugin and install it into Blender, as well as update it with new code while Blender is running. Through the extension it is also possible to debug the plugin via the Visual Studio Code debugger.

Notably Visual Studio Code runs on a separate python environment than Blender itself. Therefore a python version that corresponds with Blenders python version should be installed. In this case version 3.10.8.

To enable code completion inside the IDE the module fake-bpy-module-3.4 (nutti, 2023) can be installed. This module does not have the same functionality as the Blender API, but only fakes the existence of the API in the IDEs python environment and enables code completion and can show definitions of the functions of the actual Blender API.

The *pictureCity* plugin uses multiple libraries to implement the city gen-

eration from an image. The most important libraries are OpenCV (OpenCV Team, 2023), Shapely (Gillies, 2023) and Centerline (Todic, 2023). These libraries can be installed into the IDEs python environment through *pip*, but they have to be installed into Blenders python environment as well. For this purpose a python script has been created, that installs these libraries into Blenders python environment before the *pictureCity* plugin can be installed into Blender.

4.2 Blender API

For the plugin to be accessible inside Blender various *Panels* and *Operators* have to be created.

A *Panel* is a subclass of the *"bpy.types.Panel"* class and contains user interface elements, that can be shown in various locations inside the Blender user interface. In this case the *"ImageLoader"* Panel can be accessed in the toolbar of the *"3D-Viewport"* inside Blender. This *Panel* then can contain text, *Properties* and buttons. *Properties* can be manipulated by the user and can then be accessed inside *Operators*. Buttons execute the *Operators* themselves.

An *Operator* is a subclass of the *"bpy.types.Operator"* class and contains the functionality of the plugin, which is executed, whenever the *Operator* is called inside the Blender user interface.

A *Property* is a type of variable that is stored inside the Blender file, whereby there are two types of *Properties*. *WindowManager Properties* are stored inside the *"WindowManager"* and are only present in the file until

it is closed, whereas *"Scene Properties"* are stored in the current Blender *"Scene"* and are persistent between multiple sessions. *Properties* can be of one of multiple data types, such as boolean, integer, float, String or more complex types, such as vectors or collections.

Panels, *Operators* and *Properties* have to be registered, so that they can be accessed inside Blender.

4.3 Plugin Overview

The plugin contains an *"__init__.py"* file that contains details, such as the name, the author and version of the plugin and also registers all *Panels*, *Operators* and *Properties* with Blender whenever the plugin is enabled.

Only one *Panel*, the *"PCT_PT_ImageLoader_PNL"*, is used in the *pictureCity* plugin. It contains one *Property* for the city width and city height respectively. It also contains a file chooser, for the user to select an input image, a boolean *Property* which indicates, whether or not intermediate images are to be saved while processing the input image. Finally the *Panel* contains multiple buttons, which correspond to the *Operators* that process the buildings, road network, water and parks respectively.

Each *Operator* uses a *"Processor"* class, corresponding to the city component to process the input image into 3D-data, whereas each *Processor* uses multiple classes to structure data.

"pictureCity" also contains a module with utility functions that are used on multiple occasions, called *"utility_functions.py"*.

4.4 General Image Processing

Each *Operator* first creates a binary black and white image from the input image, where all pixels from the input image, that are the corresponding color of the city component, are set to white and all other pixels are set to black. Then this component image is input into the corresponding *"Processor"*.

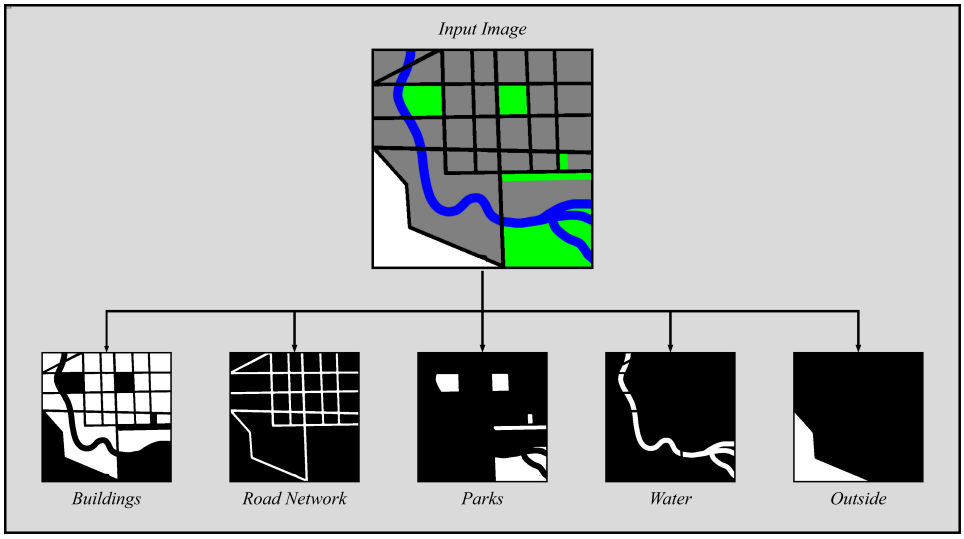

Figure 1: Image component splitting.

4.5 Road Processor

The *Road Processor* creates a *road network* out of an binary input black and white image containing the user-generated roads.

The *shell* and *holes* contours are calculated with the *OpenCV2* function *"findContours"*. This function returns a list of contours in the image and a hierarchy of these contours. This hierarchy is used to determine which

contours are *shells* and which contours are *holes*.

Currently *"pictureCity"* only supports one *road network* and therefore expects to find only one *shell* and a list of *holes* inside that *shell*.

Road Network Creation

From the *shell* and the list of *holes* a *"MultiPolygon"*, of the *Shapely* library is created.

With this *"MultiPolygon"* a *centerline* is calculated with the *"Centerline"* library. The *"Centerline"* class takes a *"MultiPolygon"* and an *interpolation distance* and returns a list of *"Shapely" linestrings*. The higher the *interpolation distance*, the longer the individual *linestrings* in the result. This creates a faster run-time, but makes for more inaccurate results of the calculated *centerline*.

For each individual *linestring* of the *centerline* a *road segment* is created. The *"RoadSegment"* class holds fields for the start and end point of the *linestring*, but also two separate lists for neighboring *road segments* at the start and end point of the *road segment*. These lists are initially empty and need to be filled, so *intersections* and *roads* can be calculated.

The initial method of finding these neighboring *road segments* was to compare each *road segment* with all other *road segments* and search for equal start or end points.

This proved to be very inefficient on account of the polynomial run-time of this approach. To improve the run-time of this approach, any time a neighbor of a *road segment* is found the current *road segment* is appended

to the neighbors list of neighbors as well. This implies, that the currently looked at *road segment* can be removed from the list of *road segments* with which each *road segment* is compared.

Through this method the run-time of the comparison was cut roughly in half, but still behaved polynomial, and proved therefore too slow for large *road networks*.

Ultimately a grid-based acceleration-structure is used. It works by sorting all *road segments* into a grid containing 3 pixel by 3 pixel cells, based on their start- and end-points. The grid itself uses a python dictionary, so only non-empty cells are created.

Once all *road segments* are sorted into the grid, for each *road segment* of the *centerline* all *road segments* in the corresponding cells to their start and end points are compared with the initial *road segment*. If any points are equal, the *road segments* are added to their respective neighbor lists.

This approach resulted in a run-time, that was deemed sufficiently fast for the plugin, since the run-time from the test image was reduced from roughly fifteen minutes to under a second.

With all neighbor lists filled, the *intersections* in the *road network* can be determined. Each *road segment* with two or more neighbors at the start or end point constitutes an *intersection* at the respective point.

For each *intersection* through this method an object of the *Intersection* class is created and stored in a list of all *intersections* in the *road processor*. Each of these *Intersection Objects* contains the location of the *intersection*, a list of the adjacent *road segments* and a list of adjacent *roads*, which is

empty at the time of initialization.

To calculate the *roads* in the *road network*, for each adjacent *road segment* of each *intersection* a road object of the type *Road* is created and stored in a list in the *RoadProcessor*. If a *road segment* is already part of a *road*, no extra *road* is created. This prevents the creation of duplicate *roads*.

Each *Road* object is filled with a list of *road segments* that are part of the *road*. To fill this list, a recursive function, which starts at the *road segment* adjacent to the *intersection* is traversed. This means, that a start point is set, in this case the point of the *intersection*, and the end point is defined as the other point of the *road segment*. Then the next *road segment* to be traversed is fetched by taking the neighboring *road segment* at the end point of the initial *road segment*. In the same way this next *road segment* is traversed, with the start point being set to the end point of the previous *road segment*. In this manner the *road segments* are traversed until the neighbor list at the end of a *road segment* is either empty, or contains more than one neighbor. This constitutes the end of a *road*, where no neighbors in the list means, that the *road* ends in a *dead end* and multiple neighbors means, that the *road* ends in an *intersection*.

After the *road* is traversed, the *Road* object is filled with the list of *road segments*, the start and end *intersections* (or *None* if the *road* ends in a *dead end*) and the start and end point coordinates.

On creation of the *Road* object or on altering the list of *road segments*, the length of the *road* is set by calculating the length of each *road segment* in the list and adding up the results.

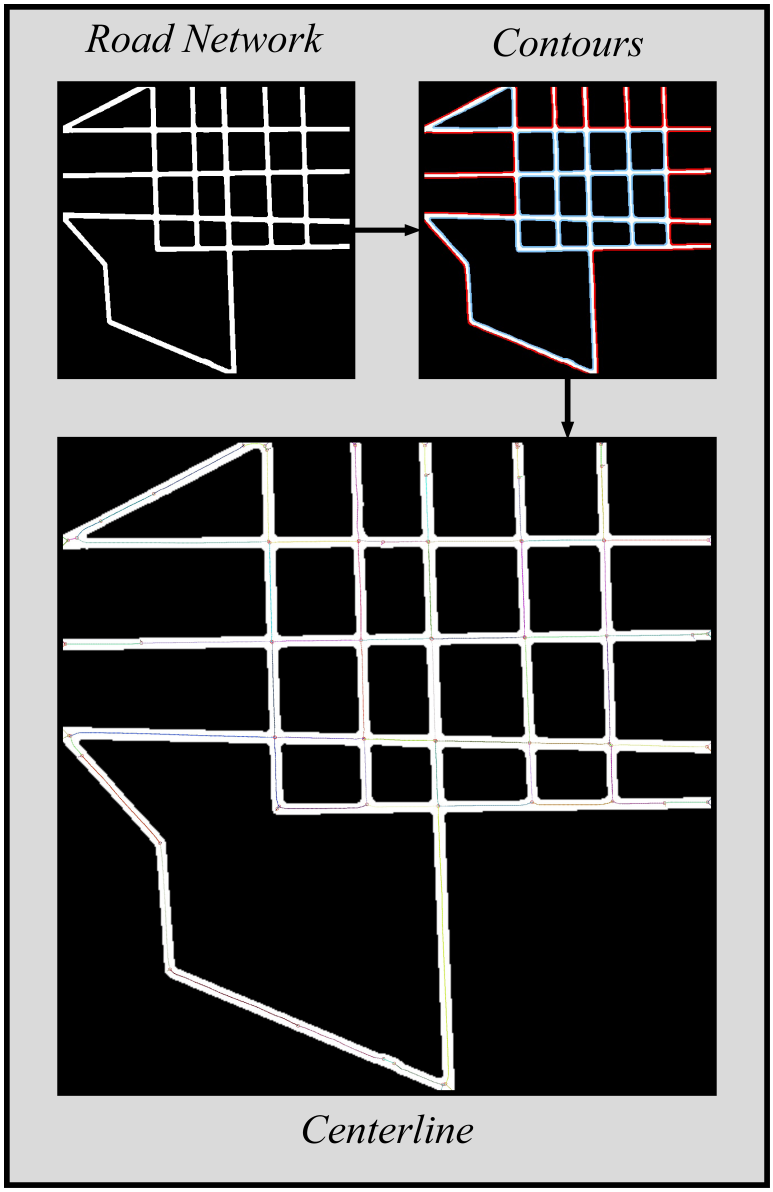

Figure 2: From the *Road Network* image, the *Shell* (red) and *Hole* (blue) contours are calculated. With these contours the *centerline* is calculated.

With this step, the *road network*, consisting of *intersections* and *roads*, is calculated. As described in section 3 (General Approach), the network still contains multiple artifacts, that need to be removed.

Refining the Road Network

Since the length of each *Road* object is known, and the *dead ends* are signified by one of the *intersections* of a *road* having the *None* value, *short dead ends* are easily identifiable. The *RoadProcessor* checks all *roads* for this and then removes all *short dead ends* from the road list and also from the *intersections* these *roads* are adjacent to. This can create *invalid intersections*, which can be identified by the amount of adjacent *roads* an *intersection* has stored. *Intersections* with no adjacent *roads* can be safely deleted from the *RoadProcessor*'s intersection list. *Intersections* with exactly two adjacent *roads*, can also be deleted, but both *roads* need to be merged together.

To merge two *roads* together, a new *road* is created. The start and end points of this *road* are determined by using the start or end point of both *roads* respectively, that do not coincide with the deleted *intersection* in between the *roads*. In a similar manner the start and end *intersections* are set, by checking the start and end *intersections* of each *road*. The last step is to merge the lists of *road segments* from both old *roads*. Since the lists of *road segments* are sorted from start to end point of a *road*, these lists have to be selectively reversed, before being merged together. This happens in such a manner, that the new list, also, is sorted from the start point of the new *road*, to the end point of the new *road*.

After the *short dead ends* and *invalid intersections* are removed, and the *roads* are merged together, it is possible that new *short dead ends* have appeared inside the *road network*. Therefore the process of removing *short dead ends*, *invalid intersections* and merging *roads* has to be repeated, until no more *short dead ends* are identified in the *road network*.

Once all *short dead ends* have been removed from the *road network*, short *roads* and their *intersections* have to be addressed - the *duplicate intersections*. These are also easily identifiable by the length of the *road*, and their start and end *intersections*. These *intersections* are merged by first deleting the short *road*, and then creating a new *intersection* at the midpoint between the two old *intersections*. Then, for each *road* adjacent to the old *intersections*, the old *intersection* and its position is updated with the new *intersection* and its position. The same update happens with the adjacent *road segments* that are part of each of these *roads*.

With all of these artifacts removed, the resulting *road network* can be transferred into Blender.

Creating the Road Network in Blender

The *bpy*-module provides the tools to create bezier curves. For each *road* in the *Road Processor* a curve of the type *BEZIER* is created in the Blender file. To the curve as many control points are added, as there are unique points in the respective *road*.

The coordinates of the bezier points cannot be applied directly from the coordinates of the road points, since these are still in the pictures coordinate

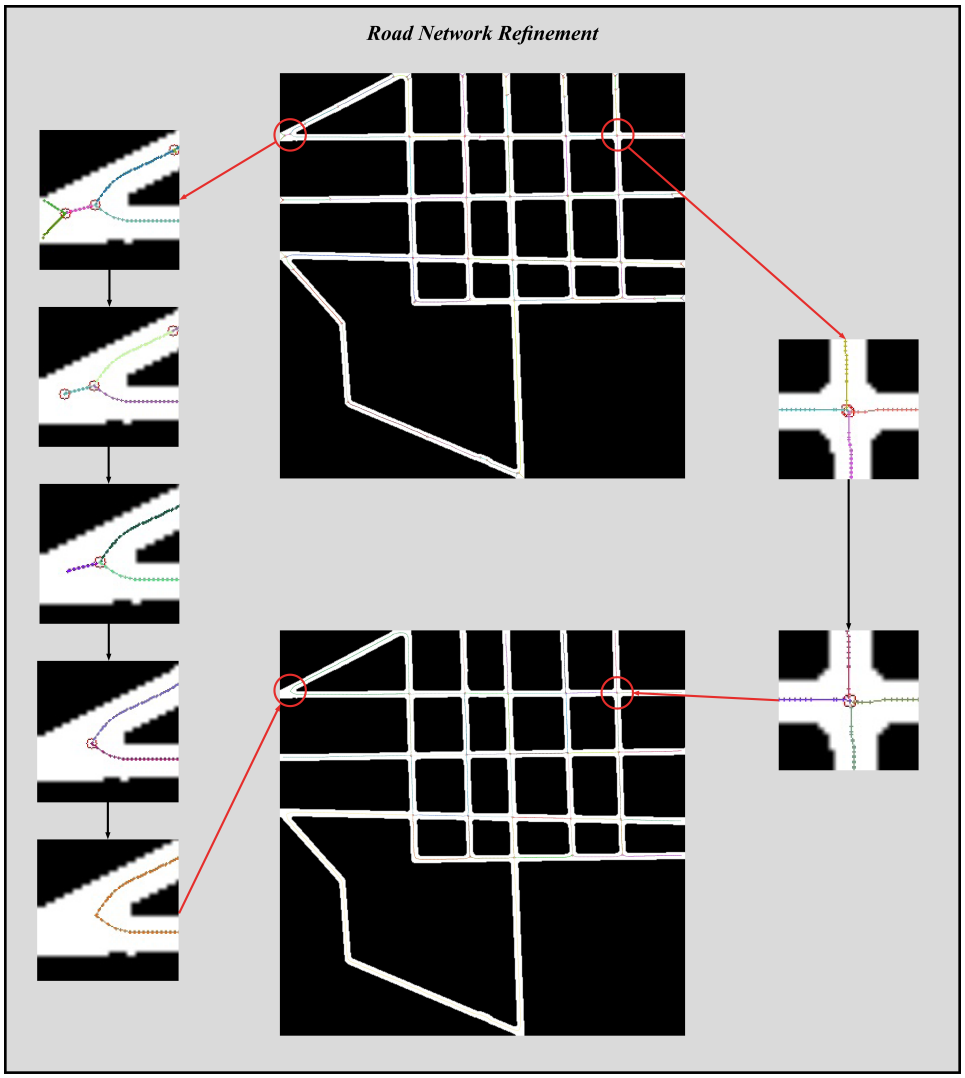

Figure 3: *Short dead ends* and *invalid intersections* (left) are removed and *duplicate intersections* (right) are merged.

space. Instead each point has to be transferred into the city's coordinate space. This is possible by sampling the coordinate in picture space and calculating the coordinate in city space by multiplying it with the factor between the picture's resolution and the user-given city width and height.

Once the city space coordinates are calculated, the coordinates of the points of the bezier curve can be set and the curve can be linked into the Blender scene.

This method is repeated for all *roads* in the *road network*. The resulting 3D data does not have any meshes attached to it yet. Therefore a plane is created for each *road*, which represents one part of the *road*. The plane could be detailed with road markings and a sidewalk or street lights. To distribute the plane along a curve, a combination of Blenders *array* and *curve* modifiers is used. These can be created through the Blender API. The target of the *curve* modifier is the respective bezier curve. The amount of the array modifier depends on the length of the road inside Blender.

With this step completed for all curves, the *road network* is created inside Blender.

4.6 Building Processor

Similar to the *Road Processor*, the *Building Processor* creates multiple *lots* out of the binary black and white input image containing the user-generated building areas and then distributes building points in these *lots*. These points are then transferred into Blender, where bounding boxes are generated, which can later be filled with either preexisting 3D meshes of buildings,

or procedurally generated building objects.

Lot Creation

To calculate the *lots* in the picture, once again *OpenCV2*'s *"findContours"* function is used on the input image. For now the *"pictureCity"* plugin does not use the hierarchy of the retrieved contours, which means, that only *lots* without *holes* are supported. This can be expanded by using the returned hierarchy to determine *shells* and *holes* in the retrieved contours.

For each retrieved contour a bounding box is calculated, so that further operations do not have to be calculated over the whole image, but only inside the bounding boxes.

With the calculated bounding boxes, for each contour a *Lot Image* is created. As previously mentioned, this image contains zero values for each pixel outside the contour and one values for each pixel inside the contour. To test whether a pixel is inside the contour the *OpenCV2* function *"pointPolygonTest"* is used. This function takes a contour and a point and returns the distance to the nearest contour edge. If the distance is negative, this means, that the point is outside of the given contour.

Only pixels inside the bounding box of each contour are tested. Since each lot image is initialized with black pixels and every pixel outside of the bounding box is also outside of their respective contour, only pixels inside the bounding box need to be tested, and only pixels with a positive distance value need to be set to white.

Once a *Lot Image* is calculated, a new object of the *Lot* class is created,

which takes the *Lot Image*, the contour and the bounding box as initial values. The *Lot* object is stored in a list in the *Building Processor*.

After creating all *Lot* objects with the contours, the list of *Lot* objects is iterated through and for each *lot* the buildings are placed.

Building Point Placement

The placement occurs inside the *Lot* class and follows the procedure described in subsection 3.2 (Buildings), which is based on *Organized Order in Ornamentation* by Gieseke et al. (2017).

Multiple maps are generated and then combined to a *Placement Map*. Each pixel in the *Placement Map* has a float value. The higher the value, the more desirable the location is to place a building. Once a building is placed on the pixel with the highest value, the maps are regenerated, so the next building can be placed. This process is repeated until no more buildings can be placed inside a *lot*.

The currently generated maps are: a *Lot Stencil Map*, an *Inverse Distance Map*, an *Edge Stencil Map* and a *Building Stencil Map*.

For the *Lot Stencil Map* the lot image can be used. This map simply ensures that all buildings are placed inside the lot boundaries.

The *Inverse Distance Map* uses the lot image as base and then calculates, for each pixel, the distance to the nearest zero pixel, by using the *OpenCV2* function *"distanceTransform"*. This is called the *Distance Map*. The *Distance Map* is not used in the *Placement Map* itself, but is used in the creation of further maps. The resulting values are then normalized and

inverted.

The *Edge Stencil Map* will use the aforementioned *Distance Map* to create a stencil, that prevents buildings from intersecting with the edge of the contour. As described in subsection 3.2 Buildings buildings of various sizes are placed inside the *lot*, starting with bigger buildings to smaller buildings. That means the *Edge Stencil Map* has to be recalculated each time a new building size is used. The *Edge Stencil Map* is calculated by using a threshold on the *Distance Map* to set all pixels with a value lower than the threshold to zero and all pixels with a value higher than the threshold to one. The threshold is the distance of the center of a bounding box to one of its edges. This will be called the *radius* (r) of a building.

At last the *Building Stencil Map* is created. This map exists to prevent buildings from overlapping with each other. It contains stencils for each building that is already placed in the *lot*. This means, this map is initially filled with one value pixels. Once a building is placed two intermediate maps are created.

The first, temporary, map is created with all one value pixel and then a singular black pixel is added at the point, where the building is placed. Then the *"distanceTransform"* function is used to determine the distance of each pixel to the building point. A threshold is used to create a stencil for the building, where the threshold is set to the radius of the building.

The second map created is used as an intermediate map and is updated every time a new building is added. It is called the *Intermediate Building Stencil Map*. It is initialized with all one values as well and anytime a new

building is placed into the *lot*, the temporary building map is multiplied with the *Intermediate Building Stencil Map*. This map then holds the footprints of all buildings which were already created with their correct sizes. To prevent new buildings from overlapping with the already placed buildings the stencils in the *Intermediate Building Stencil Map* have to be expanded with the current building radius. This, again, happens by using the *"distanceTransform"* function and then applying a threshold to the result, wherein the threshold is the current building radius.

The resulting map is the *Building Stencil Map* that is used in the *Placement Map*. it will have to be recalculated from the *Intermediate Building Stencil Map* each time a new building is placed, with the current building radius.

With all individual maps calculated, the *Placement Map* is created by multiplying the *Lot Stencil Map*, the *Inverse Distance Map*, the *Edge Stencil Map* and the *Building Stencil Map*.

On the resulting *Placement Map* the highest value pixel is determined and a building point is set at that pixels location. Then the individual maps are recalculated as needed and the next best point is retrieved and another building point is set. This is repeated until no further buildings of the current size can be placed. If this is the case, the next lower size of buildings is placed, until no more suitable places for this size are found as well. This process in turn is repeated until all building sizes are placed.

Each building point is stored in a list in the *Lot* object, together with the associated building radius.

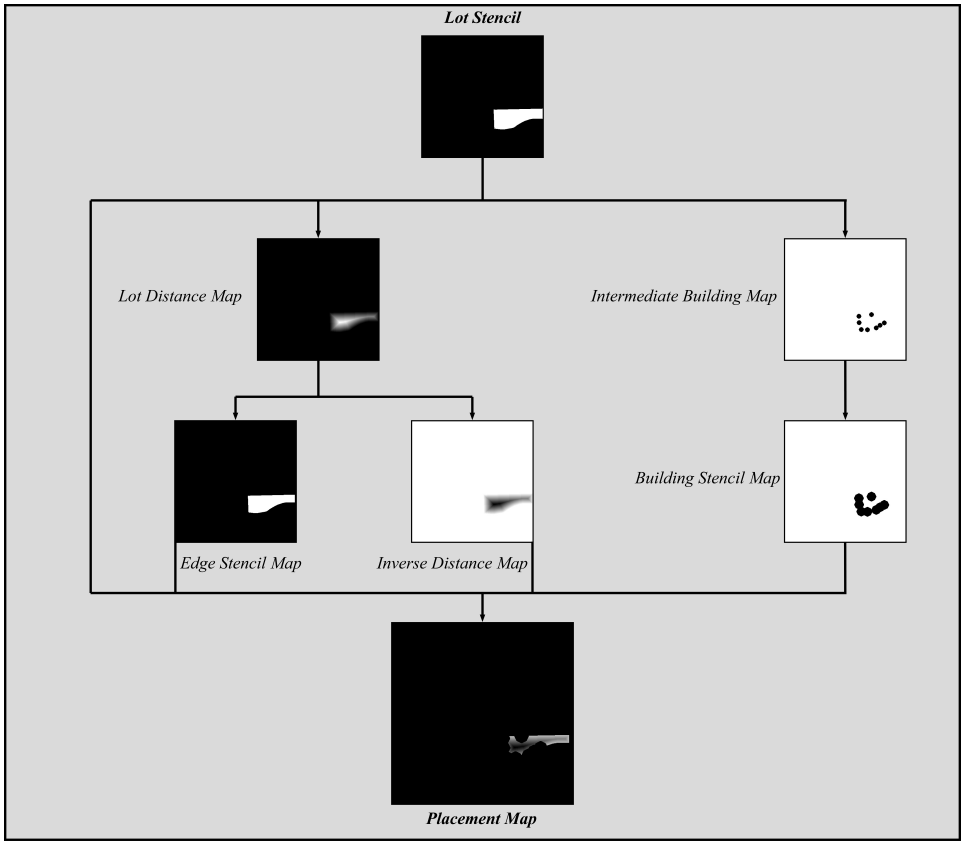

Figure 4: The maps after eight buildings have been placed. The maps are multiplied to create the *Placement Map* and find the next best building point.

Creating the Buildings in Blender

Once all building points are set in a *lot*, the buildings can be created inside Blender.

To determine the coordinate in Blender on which the building is to be placed, the quotients in x- and y-direction between the image resolution and the user-given city size is calculated and then the building coordinates are multiplied by these quotients. To the resulting 2D-point the height

coordinate is appended. This coordinate is half of the building height, so that the bottom side of the placed building bounding box is even with the ground plane. For now all buildings are a random height, but the height could be defined by the user through another image input with height values in the future.

Once these coordinates are calculated, all buildings of all *lots* are placed inside Blender as bounding boxes. This is achieved by using the *bpy* operator *"mesh.primitive_cube_add"* from the *"bpy.ops"* module. This operator takes the calculated location coordinate and scale and creates a cube of the given scale at the given coordinates. The x and y scales are both equal and are set to $\frac{r}{\sqrt{2}}$, where r is the building radius. The cubes could also be rotated on creation, so they would face the nearest contour edge as described in subsection 3.2 (Buildings) but this is not implemented yet.

Once all these bounding boxes are created inside Blender, they can be filled with either preexisting meshes, or procedurally generated meshes. This, however, is not implemented as part of this work.

4.7 Water and Park Processors

The *Water and Park Processors* are currently not completely implemented into the *pictureCity* plugin, since the main focus of this work were the *Road and Building Processors*. For now the *Water and Park Processors* merely take their respective black and white input images and use the *OpenCV2* function *"findContours"* to retrieve the contours of the parks and water areas in the user-given input image. A hierarchy is also retrieved in both

cases, so that nested areas can be treated correctly.

With the contours as base the water and park areas could be further processed.

With the *bpy* API the edges of the shell and hole contours could be created in Blender by placing vertices and edges in the scene at the corresponding coordinates to the contours. After these edges are placed, the fill polygon operator can be used to create polygons that span the areas of the shells and holes. Once these polygons are created the boolean modifier could be created on the hole contours in the *"difference"* mode. The target of this modifier is then a collection with all the polygons of the hole contours that are inside the shell contour.

This operation creates a polygon that spans the area of a park, or water like area, but leaves holes where the holes in the input image have been.

Onto this polygon multiple geometric features could be added, such as trees, bushes or grass through a process that is similar to the building placement process, if the area is a park.

4.8 Using the finished Plugin

To install and use the *pictureCity* plugin in Blender, first the "install-prereqs.py" script has to be executed inside Blender. The script can be opened inside the scripting workspace and then executed to install all necessary libraries for the *pictureCity* plugin.

Once the required libraries are installed, the *pictureCity* plugin can be installed through the user preferences.

The *pictureCity* plugin can then be found in the toolbar (Shortcut N) on the right side of the 3D-Viewport under the tab *"PCITY"*. In there, the user can select an input image and then has the options to create the different components of the city.

An option to save intermediate images can be checked, so the different image processing steps can be retraced.

5 Evaluation

The goal of this work was to create and implement a method to create a three-dimensional representation of a city from a single input image. In order to meet this goal the *pictureCity* plugin was created. It is capable of reading an input image, processing the image and its individual components and creating three-dimensional representations of roads and buildings in the 3D software Blender. A method to create data for water and parks was also described but not implemented and the results of this can therefore not be evaluated at this time.

5.1 Roads

The resulting roads of the *pictureCity* follow the overall form of the roads in the input image, but differ in the exact road shape. Straight roads in the input image are not completely straight in the output data and rather sometimes take a more stepped or wavy form. Sharp corners result in substantially more rounded corners, and towards intersections the roads sometimes first turn away from the intersection before connecting to it. Intersections themselves are found with a high accuracy. In the test images used all intersections have been correctly determined by the program.

5.2 Buildings

The *pictureCity* plugin can populate the area of the output city with buildings that are situated inside the allocated lots specified in the input image. Buildings are represented by cuboid bounding boxes and are primarily

placed towards the outside of their respective lots. Multiple sizes of buildings are placed in the scene.

Each building is placed in a manner to not intersect any other buildings, but the buildings do also not connect to each other as, for example, buildings in inner city blocks of real world cities would. It is possible that buildings intersect with roads, since some roads do not completely follow the shape of the marked roads in the input images. The building placement algorithm should prevent intersections with roads, would the roads not deviate in shape.

Currently buildings are all rotated in the same direction.

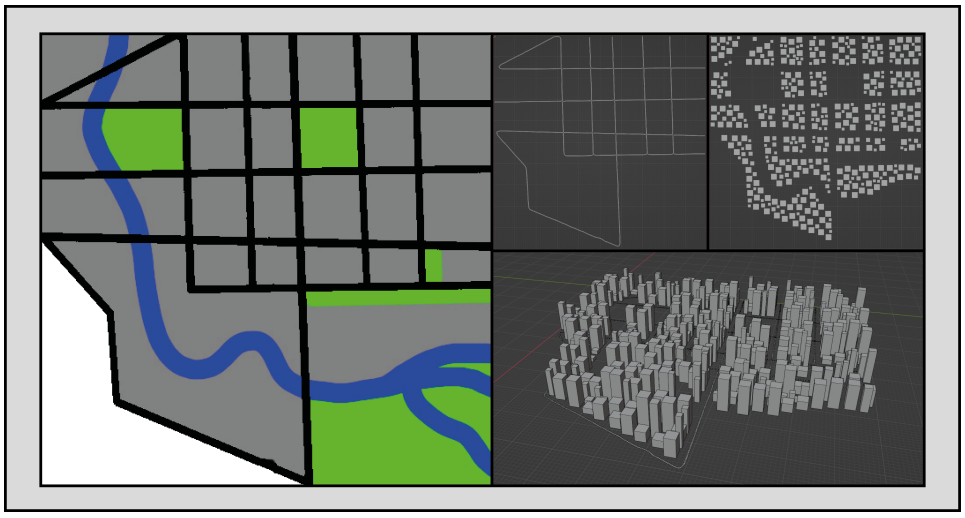

Figure 5: The input image and the resulting 3D city inside the Blender software.

6 Limitations

The method to create a three-dimensional city from an input image, described in this work, has certain limitations, that are partly based on the problem definition and partly on the chosen approaches to solve the problem.

Since the input image is a two-dimensional map, the plugin cannot create any bridges over water or other roads. For now the system only accepts one picture input, so neither terrain height, building height or different city zones can be controlled by the user yet. The sketching phase of the input image itself can take a lot of time for bigger cities as well.

Regarding bigger cities, the *centerline* calculation itself can take a lot of time on bigger road networks. Furthermore, as described in Evaluation, the *centerline* approach can create roads whose shape do not completely follow the shape of the input road network.

A computational limitation arises from the algorithm which traverses the road segments to create roads. The algorithm uses a recursion based approach. This means that, if the amount of road segments in a road is too high, the maximum recursion depth in python can be exceeded. This can happen if a road in the network is very long, or if the *centerline* is created with a very small interpolation distance parameter. The *centerline* also does not have any system in place to measure the width of the roads, so the created roads all are a similar width currently.

The building placement approach similar to Organized Order in Ornamentation (Gieseke et al., 2017) results in buildings that are not connected at any point and can therefore not represent blocks of houses as often seen

in real world inner cities.

Currently only one coherent road network is supported in the input image. Multiple disconnected or nested networks result in an error. Nested building lots are not supported currently either. The plugin will create buildings still, but the result may be unexpected or undesired.

7 Future Work

The image-driven procedural city generation leaves multiple options for future work to be created. The *pictureCity* plugin itself could be further refined to produce better results with the current approaches.

The resulting *centerline* can be further refined by smoothing out the result to prevent stepped or wavy roads. The width of the roads can be calculated and transferred onto the bezier spline points in Blender to produce accurately wide roads. Corner detection could be used to more accurately represent the shapes of intersections and sharp road corners. To do this a structure tensor could be used.

Buildings could be rotated towards the nearest contour edge. The currently implemented building placement method, based on *Organized Order in Ornamentation* (Gieseke et al., 2017), could be complemented through a different method, like lot subdivision or by using parts of the segments of the contour to place buildings. Different styles of placement could be specified by a user-given zone input map. Furthermore, the plugin could accept further user-created input maps, such as a height map for the terrain and a height map for buildings. The buildings themselves could be further refined from their current bounding boxes by using procedural modeling techniques, such as *L-Systems* or *split grammars*.

Since a big city takes a lot of time in the sketching phase, the user could define a part of the city that is important in the end application, such as the shape of a race track through the city for a racing game (*game area*). Then a completely procedural approach could be used to create parts of the city

that are further away from the *game area*, to fill up the scene. Furthermore, the further away these generated parts are from the *game area*, the less detailed they would need to be. Level of Detail meshes could therefore be used to create a performant result.

8 Conclusion

This work was set out to answer the question if it is possible to create an image driven city generation tool for the 3D software package Blender. Over the course of this work a method was devised to recognize building lots, park and water areas and roads from an user-generated input image. From this data a road network and populated building areas have been created in the 3D scene.

However, both the approach for building distribution (*centerline*) and the approach for road network creation (*Placement Maps*) have been found to be further improvable. This is possible by further refining their results, or by supplementing them with different approaches.

Furthermore, it was devised, that a city generation approach purely based on an input image may give the user a lot of control, but may be too costly to create real world scale cities. Therefore, it is suggested to complement the image-based approach with purely procedural city generation.

It can be concluded, that it is indeed possible to create an image driven city generation tool for Blender. This work and the created python program are a starting point that can be used for further research about image driven procedural city generation.

9 References

Blender Foundation (2022). Blender - a 3D modelling and rendering package. `https://www.blender.org/`. 14, 22

Chen, C., Mei, X., Hou, D., Fan, Z., and Huang, W. (2022). A voronoi-diagram-based method for centerline extraction in 3d industrial line-laser reconstruction using a graph-centrality-based pruning algorithm. *Optik*, 261:169179. 15

Deussen, O., Hanrahan, P., Lintermann, B., Měch, R., Pharr, M., and Prusinkiewicz, P. (1998). Realistic modeling and rendering of plant ecosystems. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques - SIGGRAPH '98*. ACM Press. 3

Gieseke, L., Asente, P., Lu, J., and Fuchs, M. (2017). Organized order in ornamentation. In *Proceedings of the symposium on Computational Aesthetics*. ACM. 3, 18, 35, 44, 46

Gillies, S. (2023). Shapely. `https://pypi.org/project/shapely/`. 23

Greuter, S., Parker, J., Stewart, N., and Leach, G. (2003). Undiscovered worlds–towards a framework for real-time procedural world generation. In *Fifth International Digital Arts and Culture Conference, Melbourne, Australia*, volume 5, page 5. 5

Hello Games (2016). No man's sky. `https://www.nomanssky.com/`. 1

Kelly, G. and McCabe, H. (2006). A survey of procedural techniques for city generation. *ITB Journal*, 14(3):342–351. 3, 5, 6, 7, 8, 9

Kelly, G. and McCabe, H. (2007). Citygen: An interactive system for procedural city generation. In *Fifth International Conference on Game Design and Technology*, pages 8–16. 9, 10, 11

Kempen, G. and Hoenkamp, E. (1987). An incremental procedural grammar for sentence formulation. *Cognitive Science*, 11(2):201–258. 3

Kim, S., Kim, D., and Choi, S. (2019). CityCraft: 3d virtual city creation from a single image. *The Visual Computer*, 36(5):911–924. 5, 13

Lechner, T., Watson, B., Wilensky, U., and Felsen, M. (2003). Procedural city modeling. In *1st Midwestern Graphics Conference*, volume 4. 7

Lindenmayer, A. (1968). Mathematical models for cellular interactions in development i. filaments with one-sided inputs. *Journal of Theoretical Biology*, 18(3):280–299. 4

Lintermann, B. and Deussen, O. (1999). Interactive modeling of plants. *IEEE Computer Graphics and Applications*, 19(1):56–65. 3

Longay, S., Runions, A., Boudon, F., and Prusinkiewicz, P. (2012). Treesketch: Interactive procedural modeling of trees on a tablet. In *SBIM@ Expressive*, pages 107–120. Citeseer. 3

Mandelbrot, B. B. (1982). *The fractal geometry of nature*, volume 1. WH freeman New York. 4

Nishida, G., Bousseau, A., and Aliaga, D. G. (2018). Procedural modeling of a building from a single image. *Computer Graphics Forum*, 37(2):415–429. 5, 12

Nishida, G., Garcia-Dorado, I., and Aliaga, D. G. (2015). Example-driven procedural urban roads. *Computer Graphics Forum*, 35(6):5–17. 11

nutti (2023). fake-bpy-module-3.4. `https://pypi.org/project/fake-bpy-module-3.4/`. 22

OpenCV Team (2023). Opencv. `https://pypi.org/project/opencv-python/`. 23

Parish, Y. I. H. and Müller, P. (2001). Procedural modeling of cities. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. ACM. 5, 6

Perlin, K. (1985). An image synthesizer. *ACM SIGGRAPH Computer Graphics*, 19(3):287–296. 4

ŞAHİNOĞLU, B. Y. and ÇELİKCAN, U. (2022). A grid-based multi-zone burgess approach for fast procedural city generation from scratch. *Mugla Journal of Science and Technology*. 5

Smelik, R. M., De Kraker, K. J., Tutenel, T., Bidarra, R., and Groenewegen, S. A. (2009). A survey of procedural methods for terrain modelling. In *Proceedings of the CASA workshop on 3D advanced media in gaming and simulation (3AMIGAS)*, volume 2009, pages 25–34. sn. 3

Stava, O., Pirk, S., Kratt, J., Chen, B., Měch, R., Deussen, O., and Benes, B. (2014). Inverse procedural modelling of trees. *Computer Graphics Forum*, 33(6):118–131. 3

Stiny, G. (1980). Introduction to shape and shape grammars. *Environment and Planning B: Planning and Design*, 7(3):343–351. 9

Sun, J., Yu, X., Baciu, G., and Green, M. (2002). Template-based generation of road networks for virtual city modeling. In *Proceedings of the ACM symposium on Virtual reality software and technology*. ACM. 7

Todic, F. (2023). Centerline 1.0.1. `https://pypi.org/project/centerline/`. 23

Wong, M. T., Zongker, D. E., and Salesin, D. H. (1998). Computer-generated floral ornament. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques - SIGGRAPH '98*. ACM Press. 3

Wonka, P., Wimmer, M., Sillion, F., and Ribarsky, W. (2003). Instant architecture. *ACM Transactions on Graphics*, 22(3):669–677. 9

Worley, S. (1996). A cellular texture basis function. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques - SIGGRAPH '96*. ACM Press. 4