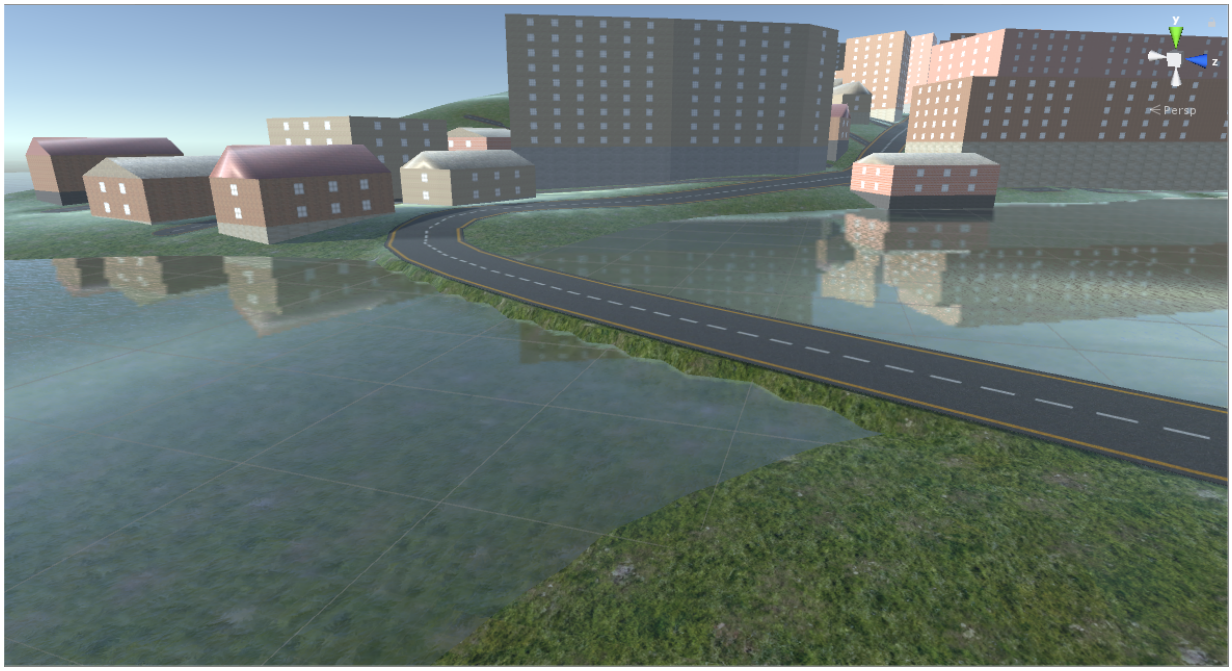
Procedural City Generation

R.J.P. Mennens 0845950
M. Russo 1192259
N. De Jong 0844786

April 6, 2017



Contents

1 Introduction

The primary goal of this project is to design and develop an application that generates a large city plan and visualizes it in 3D. This is an interesting yet complex task for computer graphics. In the years, several approaches and algorithms have been created to solve this problem, leading to a large variety of results. This report will give an overview of the most relevant papers, and from these, a selected approach will be determined as most viable to our goal. In the sections after this, the architecture of our designed system will be described. After this, the focus will be on the major components independently and presenting the concepts and the algorithms behind every component. In the end the results and possible improvements for future work are discussed.

1.1 Related Work in Literature

By looking into different papers related to city generation we concluded that road network generation is the core element and the main challenge for research related in creating a realistic city. A possible approach is to use L-Systems, like [1] does. L-Systems have traditionally been used to model natural phenomena, but Parish and Muller show a way to apply this system to road generation. This is a popular approach and has been cited by many other papers. Interesting pros are its efficiency and its versatility, as a good range of road networks can be obtained from these systems.

Another popular approach is using use land models, such as [2] and [3]. These approaches, however, are either little documented or focus on cultural elements like the geographic area or the historic period. These elements are often too complex to model or require a specific knowledge (about history, art, etc.). Differently, [4] generates roads by growing a graph using a random walk algorithm and iteratively adding roads using statistical properties of existing road networks. The result is one of the most accurate and realistic road networks, but for the same reason it is also one of the most complex approaches. Finally, [5] uses an interactive approach to generate a road map from scratch using a tensor field that can be manipulated. However, this approach requires the user to know in advance the general layout of the desired road network and for this reason it does not suit randomized city generation for our purpose.

1.2 Chosen Approach

Among all the options described above, our chosen approach is mainly inspired by [1]. The idea behind it is that every urban area has a road network that is influenced by population levels, environmental constraints and other cultural factors that determine road layout (think about the typical American grid of roads and the more European city center without superimposed road patterns). Thus, starting from a set of three maps given as input (specifying respectively the height of the terrain, the density of population and the type of road construction), the proposed system uses a simplified L-System to generate a

road network that will be finally divided into lots onto which buildings of appropriate size and geometry will be created and placed.

1.3 Requirements

The proposed solution has been coded in C# and has been rendered using Unity 3D. The choice of this game engine is motivated by its ease of use, the high number of useful built ins and components (like colliders) and the extensive amount of documentation available. As Unity handles all lower level graphical problems, we can focus on the algorithms behind the city generation. In order to compile and execute the source code of this project, all that is needed is an installation of the Unity Game Engine (Version 5.5+) with its dependencies, and the Unity Project folder containing the source code. Other than that, no libraries or other external sources are needed.

2 System Architecture

Figure 1 shows the work flow of our solution. Simply put, every step of the work flow represents a component of the system. All of these components will be described in more detail in the following sections.

2.1 User Interface

A screenshot of the constructed UI is visible in Figure 2. This user interface is integrated within the Unity Editor by using the Editor Scripting API. Editor scripting is a way of scripting that allows one to add functionalities to the Unity Editor itself. So in this specific application, a user interface has been created that can be opened within the editor. This user interface then allows one to generate a city in six main steps: terrain generation, population map generation, growth map generation, road map generation, road mesh generation and building generation. This structure reflects the generation steps of the work flow described in the previous section. The UI is designed in such a way that every step requires the previous step to be finished. For example, in order to generate the population and growth map, the terrain must be generated.

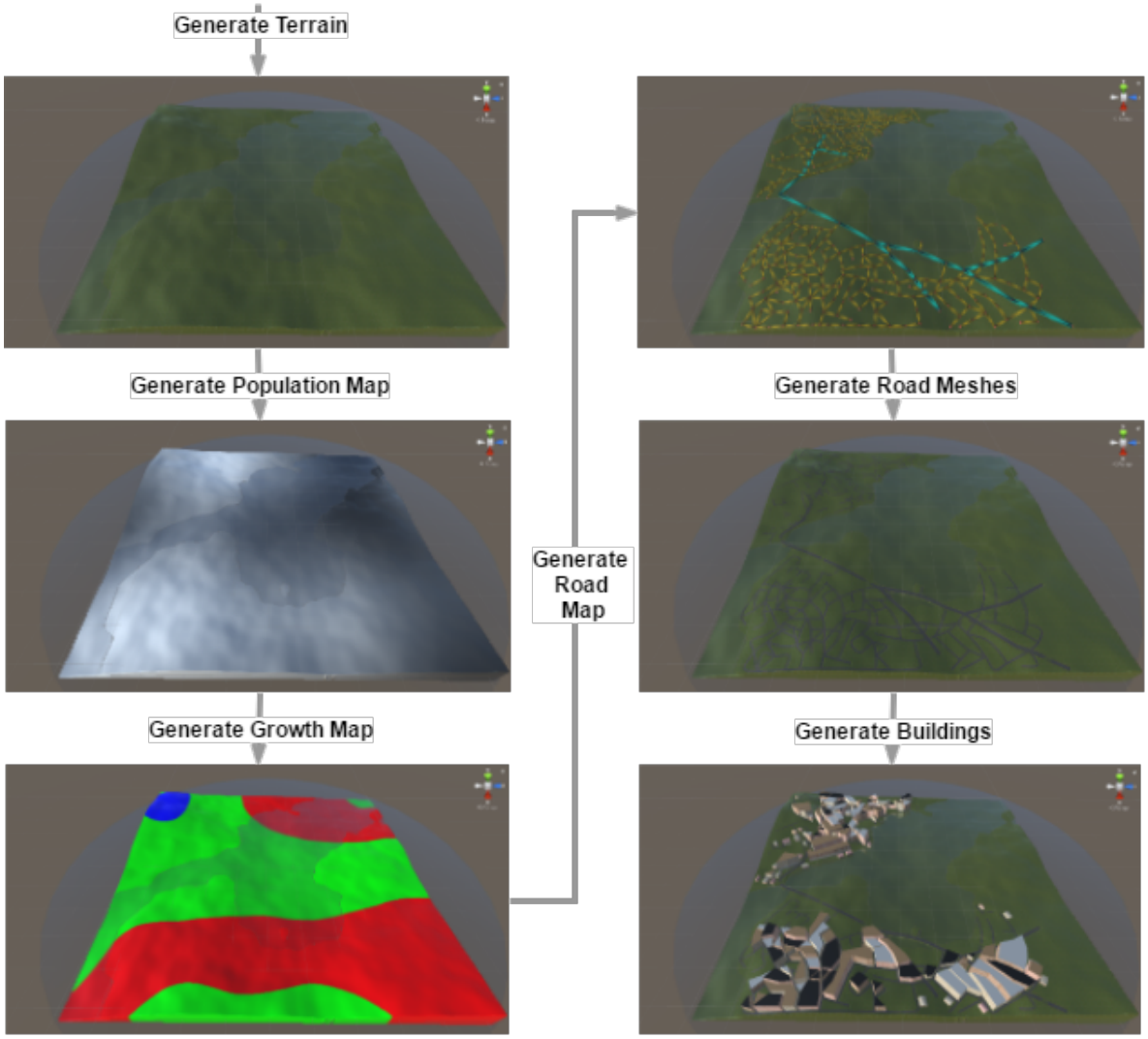


Figure 1: Workflow of city generation.

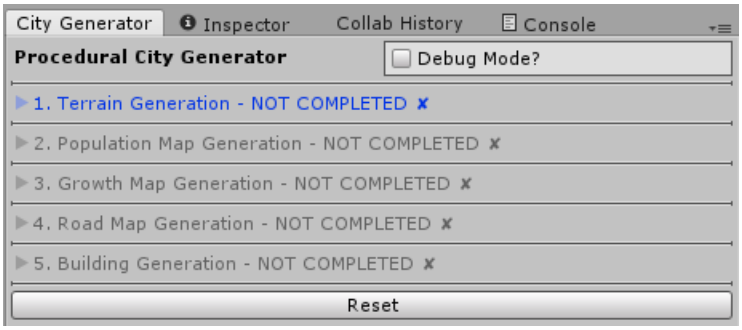


Figure 2: User Interface initial state. Terrain must be generated first. All main steps are visible, where generation of the road map and road meshes are merged.

2.2 General Overview of Solution

Shown below, pseudocode for general algorithm of the city generation process is presented. This can directly be linked to the workflow described in Figure 1.

```
function CityGeneration(parameters)
    if Procedural Terrain = True then
        Terrain = GenerateTerrain(Perlin Noise)
    else
        Terrain = GenerateTerrain(Custom Height map)
    end if
    if Procedural Population Map = True then
        Population Map = GeneratePopulationMap(Perlin Noise)
    else
        Population Map = GeneratePopulationMap(Custom Population Map)
    end if
    if Procedural Growth Map = True then
        Growth Rule Map = GeneratGrowthRuleMap(Perlin Noise)
    else
        Growth Rule Map = GenerateGrowthRuleMap(Custom Growth Map)
    end if
    Road Map = GenerateRoadMap(Terrain, Population Map, Growth Rule Map)
    Blocks = GenerateBlocks(Road Map)
    Buildings = GenerateBuildings(Blocks, Population Map)
    return Terrain, Road Map, Buildings
end function
```

3 Heightmap, Roadmap and Growthmap Generation

This section describes the generation of input maps for the main roadmap generation algorithm. After all input variables have been generated, the roadmap generation component of the application can commence. The application supports both custom input variables provided by the user as well as randomly generated input.

3.1 Perlin Noise

A Perlin Noise [6] function is used extensively in this part of the algorithm. Combining octaves of Perlin Noise is an excellent way to generate interesting heightmaps, as well as population density maps. More about this later. An octave of Perlin Noise is essentially random noise that can be combined with other octaves to create interesting maps. When a component uses Perlin Noise, the user can tweak the following parameters:

- **Octaves** - the number of octaves of Perlin Noise to generate.

- **Persistance** - the persistance of every octave of noise. A low persistance means that the octave fades faster when adding more octaves to it. The initial octaves is therefore less apparent in the final result.
- **Zoom** - the zoom value of the Perlin noise map. A higher value means that the generated terrain has more variance, e.g. the hills are closer together.
- **Seed** - the seed for the random number generator. This is randomized by default. The user has the option to make this a static value so the same result can be reached.

Note that the width and height of the noise map is defined by the component itself.

3.2 Terrain Generation

The terrain generation component uses the previously described Perlin Noise generator to generate a terrain for the current Terrain object in the Unity scene. The user can specify the following parameters for the terrain:

- **Size** - the width and length of the terrain. As an assumption, the width of the terrain is always equal to the length, such that the terrain is square. This makes the use of the population- and growth-map a lot easier.
- **Maximum and mininum height** - the maximum and minimum height values the terrain can have. After generating the noise map, the terrain's height is scaled to match these values.
- **Perlin Noise parameters** - the Perlin noise parameters as described in section 3.1.
- **Generate water (True/False)** - determines whether a water surface will be placed at $y = 0$ on the terrain.

An example of a randomly generated terrain can be found in Figure 3. Next to randomly generating the terrain, there is also the option for the user to upload their own height map files. These are grayscale images which will be converted to a heightmap.

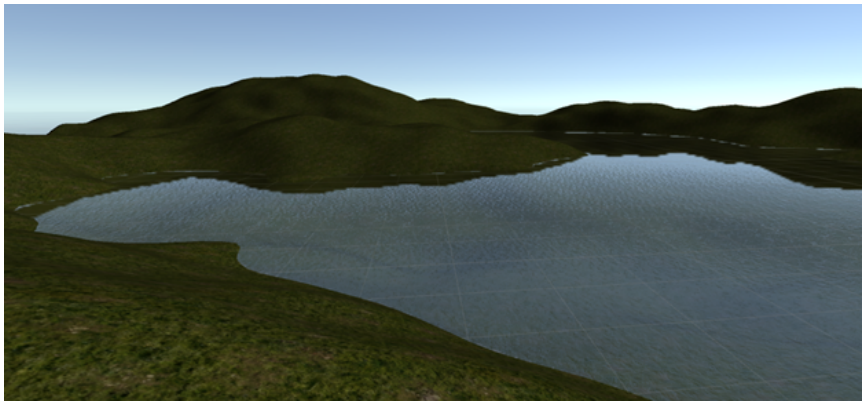


Figure 3: Randomly generated terrain.

3.3 Population Map

In order to build the roadmap, the algorithm requires a population map. This is a two-dimensional grayscale image that stores for each pixel a value between 0 and 1. Here 1 is shown as white, 0 as black, and everything in between in as a combination of the two. White means high population density while black means low population density. An example population map can be seen in Figure 4a. In order to use the population map, the map is mapped to the terrain such that at each terrain coordinate, a population value is present.

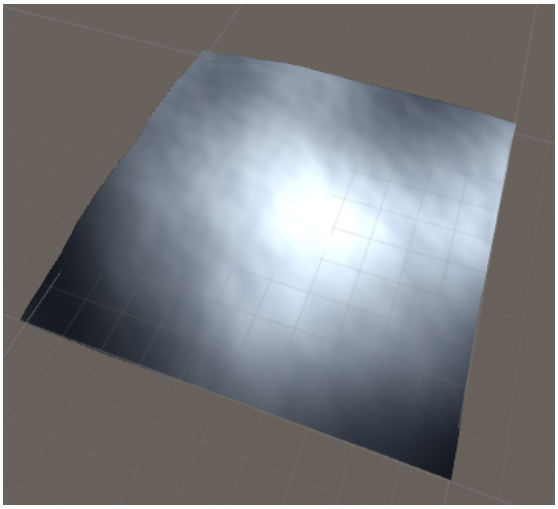


Figure 4a: Randomly generated population map. It resembles an highly urbanized region with a single center, possibly a city.

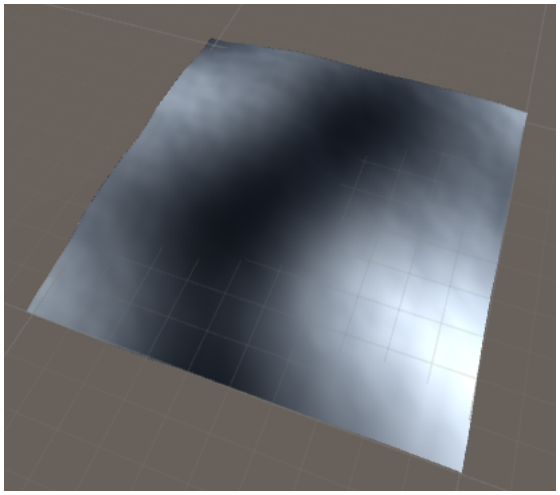


Figure 4b: Randomly generated population map. It resembles a region with multiple and distinct built-up areas.

Providing a population map for the algorithm can be done in two ways. Either the user provides a grayscale image which will be used as a population map or he can generate a random population map based on perlin noise. For the sake of city generation, it is interesting to notice that by changing the parameters of the perlin noise (see Figure 5a) the user can get population maps resembling more single cities (Figure 4a) or whole regions with multiple inhabited areas (Figure 4b).

3.4 Growth Map

Next to a population map, the algorithm also requires a growth map. The growth map is a two-dimensional image containing three colors: Red, Green and Blue. Each of these colors represents a different growth rule. Each rule is defined in such a way that they will generate certain street patterns. These street patterns will be described in more detail in Section 4.2.3. An example growth map is shown in Figure 6a. Like the population map, also the growth map is mapped to the terrain such that at each terrain coordinate a growth value is present. This can be seen in Figure 6b.

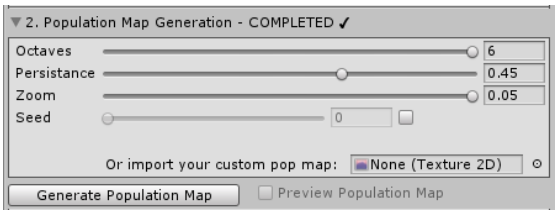


Figure 5a: UI used to generate the population map. Users have the option to select their own seed, or to use a random one. The "Preview Population Map" option allows one to show the population map on the terrain (see Figure 4b).

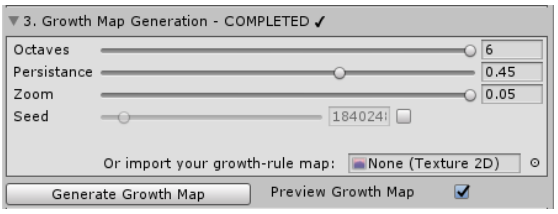


Figure 5b: UI used to generate the growth map. Users have the option to select their own seed, or to use a random one. The "Preview Growth Map" option allows one to show the growth map on the terrain (see Figure 6b).

Figure 6a: Randomly generated growth map.

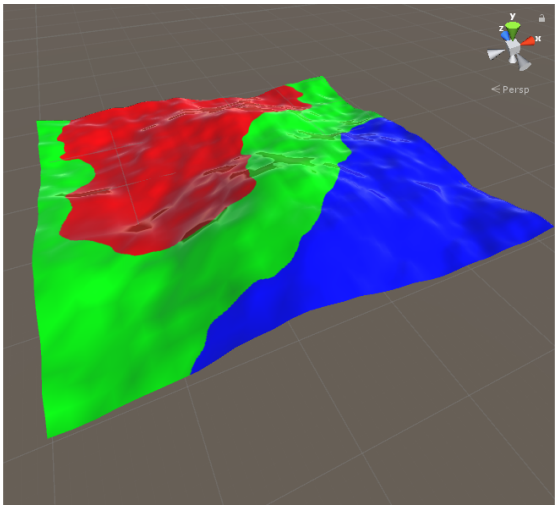


Figure 6b: Growth map mapped to terrain.

Like the population map, providing a growth map for the algorithm can be done in two ways. Either the user provides an image which will be used as a growth map, or the user can generate a random growth map based on perlin noise. The user interface used for these actions can be seen in Figure 5b.

3.5 Map Visualizations

As one can see in Figure 4b and Figure 6b, it is possible to show the 2-D maps on the terrain such that it becomes easy to interpret what these maps exactly imply. Visualizing this proved to be a non-trivial task, hence the process is described here.
In order to paint textures on a Unity terrain object, these textures need to be added to the terrain's *splatprototypes* (basically just a set of textures). So when painting the population

map or growth map (or both) on the terrain, the first step is adding 5 textures: red, green, blue, white and black to the splatprototypes. Now the actual texturing of the terrain is defined by the so called *alphamap*. This is a three dimensional float array. The first dimension represents the x-coordinate, the second dimension the y- coordinate and the third dimension is an index for the textures in the splatprototypes. So the size of the third dimension is equal to the number of textures in the splatprototypes. So basically at every coordinate you have a float array where a value represents the fraction of how much a certain texture should be present. For example, if our splatprototypes contains: {white, black, red, green, blue} then the following values in the alphamap mean that at coordinate (40, 40) the texture is white (note that the sum of values at a certain coordinate should always be 1).

$[40, 40, 0] = 1;$	$[40, 40, 0] = 0.2;$
$[40, 40, 1] = 0;$	$[40, 40, 1] = 0.2;$
$[40, 40, 2] = 0;$	$[40, 40, 2] = 0.2;$
$[40, 40, 3] = 0;$	$[40, 40, 3] = 0.2;$
$[40, 40, 4] = 0;$	$[40, 40, 4] = 0.2;$
only white	*all colors*

So now the visualization of the population/growth-map is implemented as follows.

```
function VisualizeMap(parameters)
    SetUpSplatPrototypes(textures)
    if showPopulationMap = False AND showGrowthMap = False then
        RestoreOldTexturing(oldTexturing)
    else
        for all AlphaMap coordinates do
            if showPopulationMap = True then
                Read population map value and set the AlphaMap values accordingly.
            end if
            if showGrowthMap = True then
                Read growth map value and set the AlphaMap values accordingly.
            end if
        end for
    end if
end function
```

Restoring the old texturing is done by simply setting the alphamap and splatprototypes to previously stored values. Also note that because of this approach, it is possible to visualize both the population and growth map at the same time.

4 Roadmap Generation

Once all algorithm inputs have been generated and loaded, the roadmap generation can be performed. The algorithm is based on the L-System provided in [1]. We have however simplified this system such that it is easier to implement and also becomes more intuitive for our goal. This simplification is described in Section 4.1. One important aspect of the algorithm is the distinction between highways and streets. Highways are considered to be the connections between population peeks, hence they are wider and less likely to branch. Streets on the other hand adhere more to local population values and are therefore smaller in length and branch more frequently. General pseudocode for the road map generator is shown below. The functions described in this code are eleborated in sections 4.2 and 4.3 in more detail.

```
function RoadMapGeneration(parameters)
    Roads = [ ]
    Priority Queue = [ ]
    Initial Edges = GenerateInitialRoads(Population Map)
    PriorityQueue.Add(Initial Edges)
    while PriorityQueue.IsNotEmpty do
        Edge = PriorityQueue.Pop
        Edge = LocalConstraints.ValidateRoad(Edge)
        if Road != NULL then
            Roads.Add(Road)
            VisualizeRoad(Road)
            Successors = GlobalGoals.GenerateSuccessorsFor(Road)
            PriorityQueue.Add(Successors, Priority)
        end if
    end while
    return Roads
```

4.1 Simplified L-System

In [1], an L-System is described that forms the basis of the whole algorithm. However, since L-Systems are based on string rewriting (which is quite inefficient in programming), the system has been simplified in order to make it more suitable for our implementation. This simplification was mainly inspired by the approach described in [8]. After this simplification, the algorithm is built around a priority queue which contains roads. The placement of a single road then follows the following steps:

1. Take the road segment with highest priority from the queue.

2. Feed this road segment to the local constraints in order to verify if it can be placed. If it cannot be placed, discard this road segment, else, continue.

3. Place to road segment on the map.
4. Now feed this road segment to the 'Global Goals' component such that new roads will be added to the queue.

Initially, the first road segment will be added to the queue by the Global Goals component (see Section 4.2). Then from this initial road segment, the roadmap can be will be constructed. Examples of road networks can be seen in Figure 7a and 7b.

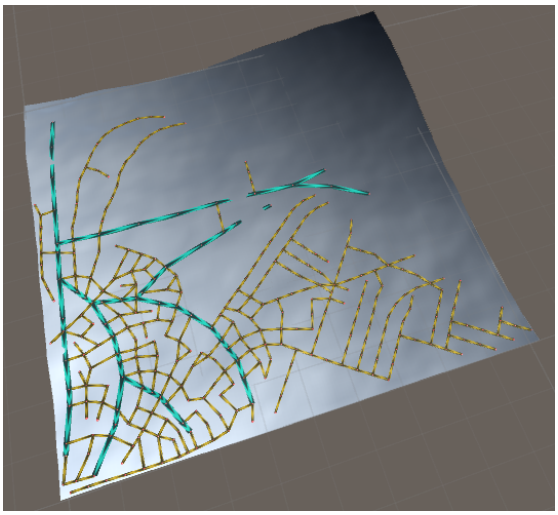


Figure 7a: Randomly generated road network for a single urban center map. In this scenario highways try to serve the most populated areas within the city.

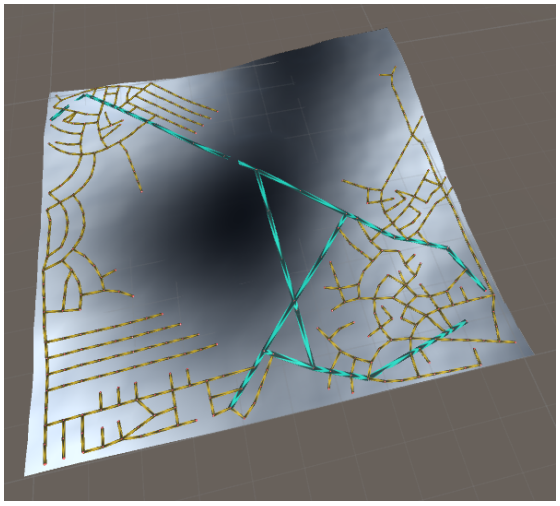


Figure 7b: Randomly generated road network for a multiple urban centers map. In this scenario highways connect different built-up areas.

4.2 Global Goals

The global goals component is one of the two main parts of the roadmap generation functionality. Its main task is to determine the next road to generate, based on a previous road segment. The main roadmap generation component calls the global goals instance whenever it pops a new road edge from the queue. The global goals component then returns a list of new potential road edges to consider. Global goals thus has two main functionalities:

- Generate the two initial roads from which the road map is constructed. The global goals component does this by finding the highest population peak, and placing a highway in a random direction. It additionally generates a road facing the other way as the first road. From these two highways the rest of the road map is then generated.
- Based on a previous road, build additional roads depending primarily on the growth rule map. It is important to notice that at this point the only pieces of information available to the growth rule are the population density at that same location and the

type of road that it is intended to be built. Clearly, this information is insufficient because it is only local and it lacks knowledge about the surrounding area. For this reason each growth rule has an in-built functionality of ray-casting.

4.2.1 Ray-Casting

Ray-casting is required to provide a growth rule with information about the surrounding area. Indeed, this information is important to direct the new road properly according to the population levels. Important pieces of information are:

- the **angle range** (α, β) in which the rays must be thrown: as it will be shown in the next subsection, different rules use different ranges.
- the **number of rays** (R) to cast in the delimited angle range. The angle range is thus divided into R smaller ranges $|\beta - \alpha|/R$ wide (γ_r, with $1 \leq r \leq R$), and within each of them one ray is thrown at a random angle. - N.B. this value can be changed by the user (see *Ray Count*)
- the **length** (n_{ray}) of the ray. The idea is then to look up to k streets/highways (their lengths are referred to as n_{road} here) ahead to determine the best direction. The formula used is then:

  $$n_{ray} = n_{road} * k_{road}$$

  By changing the value of the *lookahead* the user can thus determine separately how far in space highways and streets should look. The default setting allow highways to look through all the map, while streets are more short-sighted and can look only two streets ahead.
- the **number of samples** (S) determines the size of each step along the ray (n_{step}). Indeed,

  $$n_{step} = \frac{n_{ray}}{S}$$

  By default $S = \sqrt{n_{ray}}$, but this value can be changed by the user (see *Ray Samples*).
- the chosen **algorithm**. In this sense several approaches have been tested, but in the end the **maximum** population density value along the ray has been chosen. In this scenario no normalization is required and (especially for highways) there is the absolute certainty that a peak is detected even if preceded by an uninhabited area (average sum does not guarantee it, as was tested before).

Let p_0 be the considered starting point and $\pi(p, \theta)$ the level of population at location p along the ray with angle θ (with $\theta = 0$ being the line directly in front of p_0) and let us define the ray-casting function as $\rho(p_0, \alpha, \beta, R, n_{ray}, S)$. Its value can be determined as:

$$\rho(p_0, \alpha, \beta, R, n_{ray}, S) = \max_{\gamma_r, r \in \{1 \ldots R\}} \left\{ \max_{s \in \{1 \ldots S\}} \left\{ \pi(p_0 + s \cdot n_{step}, \gamma_r) \right\} \right\} \in [0, 1]$$

To simplify the notation $\rho_{straight}$ will be used to indicate the result of ray casting in a frontal angle range. Similarly, ρ_{left} and ρ_{right} will indicate the result of ray-casting in the respective areas.

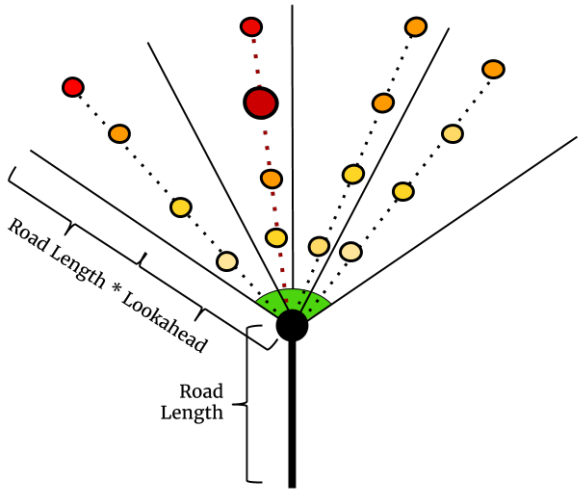


Figure 8: Raycasting example with a road with length $n_{road} = 8$, lookahead $k_{road} = 2$ and $R = 4$. The angle range (in green) is thus divided into R areas, and in each of them a ray is thrown at a random angle. $\sqrt{8 \cdot 2}$ samples are taken on each ray. In the end the chosen direction (red-dotted line) is the one where the sample with the highest value (highlighted dark-red dot) is encountered.

4.2.2 Branching Probability

Before proceeding with the description of the different growth rules, another parameter must be introduced: the branching probability. Indeed, population level thresholds cannot be considered as the only decision factors. To avoid that a street/highway branches every time that a population density value over the threshold is detected, a probabilistic measure has to be introduced. While in previous versions of the tool this probability was left as a customizable parameter, in this last implementation a new approach is used for streets: the idea is to relate the probability of street branching to the population density, having less intersections in peripheral areas and more clustered streets in the center. To emphasize the difference between suburbs and center an exponential relation has been chosen to reach this goal.

However, this approach proved to be inefficient with highways: highways generally do not branch according to the local population level but rather according to the level of population detected along the predefined directions; for this reason a customizable parameter has been left for the user to change. This value is set to 0.33 by default, which means that if 3 consecutive highway segments detect the same peak on one side, still only 1 branch is likely to be performed.

Recall that $\pi(p, 0)$ indicates the population level at location p in the space. The branch probability of road r at the same location can be then defined as

$$\mathbb{P}_{branch}(r, p) = \begin{cases} \frac{e^{\pi(p,0)}-1}{e-1} & \in [0, 1] \quad \text{if } r.type == \textbf{\textit{street}} \\ 0.33 & \in [0, 1] \quad \text{if } r.type == \textbf{\textit{highway}} \end{cases}$$

4.2.3 Growth Rules

The road map generator contains an abstract interface for a generic growth rule. As described before, three different kinds of growth rules (see Figure 9) are included in the final implementation. Let π be the chosen population threshold, and consider the following rules:

1. **Basic (Organic) Rule** - this represents the basic pattern found in most towns and cities. Roads are generated in the general direction of population peaks and flow organically over the map. The general approach (as it is pretty much the same for streets and highways) is then:

   if $\rho_{straight} > \pi$ **then**
   go straight;
   end if
   if $\rho_{left} > \pi$ **then**
   branch left w.p. $\mathbb{P}_{branch}(r, p)$;
   end if
   if $\rho_{right} > \pi$ **then**
   branch right w.p. $\mathbb{P}_{branch}(r, p)$;
   end if=0

2. **New York (Grid) Rule** - this rule uses a grid like structure that can be found in New York city. Population density is still used, but roads are only allowed to make strict 90-degree turns and cannot curve. The logic is the same as for the *basic rule*, with the only difference that the straight angle is 0 degrees and all branch angles are equal to 90 degrees.

3. **Paris (Radial) Rule** - this rule enforces roads to not only follow population, but also rotate around a certain center point. This creates a road map pattern similar to that in Paris, France. The logic is the most complex one; a simplified version is provided here below.

   if $\rho_{straight} > \rho_{left}$ **and** $\rho_{straight} > \rho_{right}$ **then**
   % there is a peak in front, so
   branch left w.p. $\mathbb{P}_{branch}(r, p)$ to start curving;
   branch right w.p. $\mathbb{P}_{branch}(r, p)$ to start curving;
   else
   % there is a peak on one side, so
   if $\rho_{left} > \rho_{right}$ **then**
   % the center is on the left, so
   $\rho_{center} = \rho_{left}$;
   else
   % the center is on the right, so

$\rho_{center} = \rho_{right}$;
end if
curve in the direction of the center
if $\rho_{center} > \pi$ **then**
branch towards the center w.p. $\mathbb{P}_{branch}(r, p)$;
branch out of the center w.p. $\mathbb{P}_{branch}(r, p)$;
end if
end if

A growth rule can be tweaked on the following parameters:

- *Priority* - priority of a street/highway being added to the road network.
- *SegmentLength* - default length of the road segment.
- *PopulationThreshold* - min population value required to go straight.
- *DefaultBranchAngle* - default angle at which side-branching happens (default: 90)
- *MaxBranchAngle* - maximum angle by which the default branch angle can vary.
- *MaxStraightAngle* - maximum angle by which straight roads can vary.

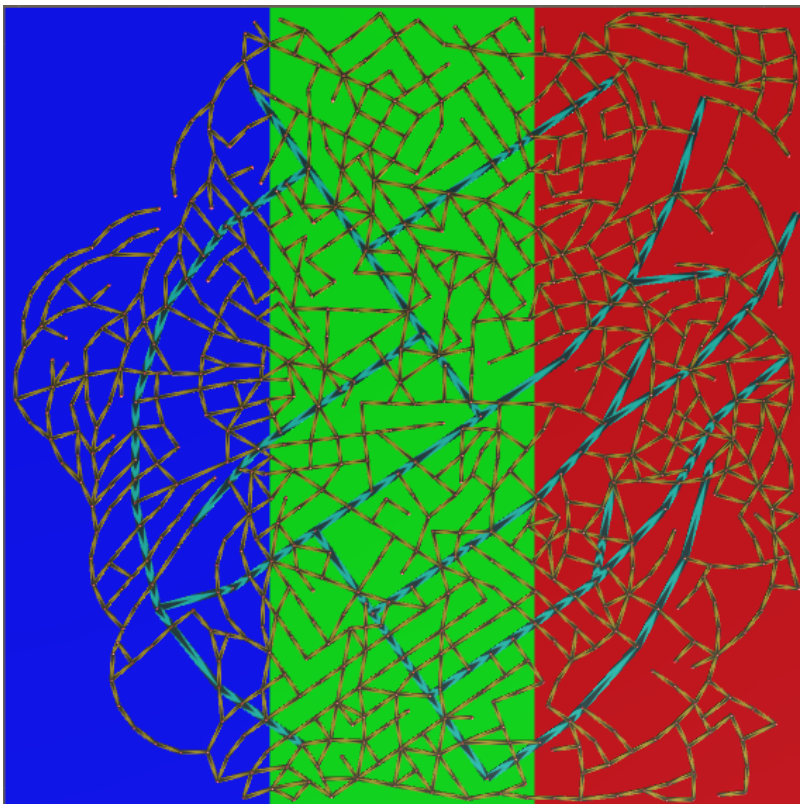


Figure 9: Effect of growth rules as generated by our algorithm. The blue, green and red areas represent the Paris, New York and Organic Rule respectively. The population peak for this map is in the center. [1]

4.3 Local Constraints

The roads that are generated by the global goals are in some sense only propositions. These road propositions still need to be evaluated on whether or not they are actually valid. In order to verify that a road can actually be placed, the local constraints are used. The local constraints consist of two components: the Position Legalizer and the Intersection Checker. Each of these will be described in Section 4.3.1 and Section 4.3.2 respectively.

4.3.1 Position Legalizer

The Position Legalizer is the most representative component of the local constraints section as it checks whether all the environmental constraints are respected or not. As the number and the nature of the constraints may vary depending on the assumptions and the complexity of the proposed solution, this component has been designed to be highly scalable in case of changes in or additions to the model.

With regards to the solution proposed here, a major assumption has been taken. This means that a road cannot be built over the ground (on an overpass, a skyway or a bridge)

or through the ground (like in the case of an underpass or a roadtunnel). According to these premises and for the time being, two controls have been implemented (note that more could be added as future work):

1. **Water**: as there cannot be underpasses nor overpassen, the proposed endpoint must be in a location where there is not water. This does however allow for bridges to be created, as long as the endpoints are not under water.

2. **Steepness**: as the road cannot lay on pillars nor go through the ground, the proposed endpoint must be located at a point where the terrain is at a reasonable height with respect to the height of the start point. Geometrically speaking, we require the slope of the line passing through the endpoints of the road to be within a certain limit. This value has been left as a customizable parameter. Let $P(x_p, y_p, z_p)$ and $Q(x_q, y_q, z_q)$ be the two endpoints of the given road in the 3D-space (where z indicates the height) and $s_{max} \in [0, 1]$ the maximum steepness (as a percentage) in input, then the position legalizer would perform this simple check:

$$\frac{|z_q - z_p|}{\sqrt{(x_q - x_p)^2 + (y_q - y_p)^2}} \leqslant s_{max}$$

In case at least one of these two conditions is not satisfied, the road goes through an adjustment process consisting of a simultaneous change of both the original length and angle (w.r.t. the original direction). Some additional thresholds have been introduced to control the maximal number of attempts or the maximal variation of length and angle, so that (for example) an adjusted road could not be shorter than half of the original road or rotated of more than α degrees with respect to the original road.

In the end, Position Legalizer returns a valid road according to the implemented environmental constraints. If the given road cannot be adjusted the system simply aborts its construction.

4.3.2 Intersection Checker

After the position of a road has been legalized, the intersection checker is responsible for checking if a road intersects with roads that are already placed. In order to do this, three cases are considered which can be seen in Figure 10. The first considers the case where the road to be placed crosses an already existing road. In this case the road is shortened such that an intersection is created. The second case considers the scenario where the endpoint of the road to be placed is close to an already existing endpoint. When this happens, the road endpoint is connected to the already existing endpoint. And the last case considers the scenario where the endpoint of the road to be placed is close to an existing road segment. When this happens the road is extended to generate an intersection. If any of these cases occur the road is adjusted, if none of these cases occur the road remains unchanged.

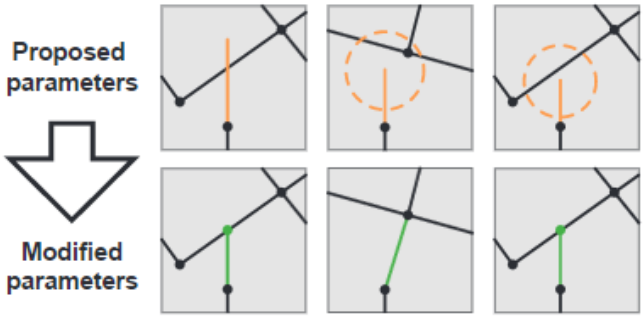


Figure 10: Intersection cases as found in [1]

4.4 Road Mesh Generation

After the roadmap generation algorithm is done, a network of nodes and edges has been mapped onto the terrain. However all of these edges are straight, so they do not really look like real roads yet. The final step in the roadmap generation algorithm then consists of converting this straight road network into a smooth and realistic road network.
Because the road network can become quite complex, the problem is simplified a bit by not procedurally generating intersections, which we deemed as future work for the application. The first step of the road mesh generation then becomes the identification of distinct roads. We basically wish to identify sequences of road segments which would form a realistic road. An illustration of this idea is shown in Figure 11. In order to identify these distinct roads, the following algorithm was implemented.

```
function IdentifyRoads(Edges, Nodes)
    roads = []
    while Edges.Count != 0 do
        roadPoints = []
        startEdge = Edges[0]
        remove startEdge from Edges
        add endpoints of startEdge to roadPoints
        successor = findNext(startEdge, startEdge.n2, Edges)
        predecessor = findNext(startEdge, startEdge.n1, Edges)
        while successor != null do
            newPoint = the endpoint of successor which was not in roadPoints yet
            add newPoint to roadPoints
            remove successor from Edges
            successor = findNext(successor, newPoint, Edges)
        end while
        while predecessor != null do
            newPoint = the endpoint of predecessor which was not in roadPoints yet
            add newPoint to roadPoints
```

```
            remove predecessor from Edges
            predecessor = FINDNEXT(predecessor, newPoint, Edges)
        end while
        add roadPoints to roads
    end while
    return roads
end function

function FINDNEXT(Road, EndPoint, Edges)
    if Two roads are connected to EndPoint then
        return the road connected to EndPoint which is not Road
    else
        nextRoad = the road connected to EndPoint which is still in Edges and makes
          the smallest angle with Road.
        if the angle between nextRoad and Road is still too large then
            return null
        else
            return nextRoad
        end if
    end if
end function
```

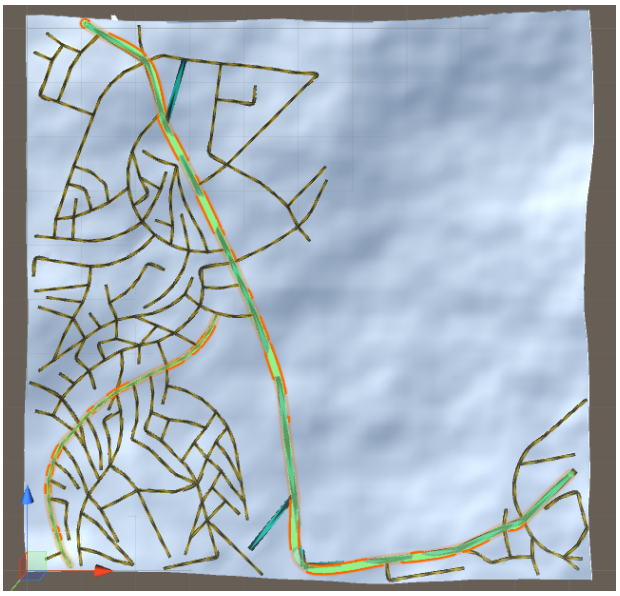


Figure 11: The highlighted parts show two examples of sequences of roads which were identified as unique roads.

After the individual roads have been identified, we basically have a number of lists of nodes. Here every list of points represents a unique road and the points represent markers along

which this road should go. To convert such a list of points (which still contains straight segments) into a smooth road mesh, Catmull-Rom Splines are used [7]. The points are given as an input and these are then used to generate a smooth curve (Figure 12a). Then to convert this curve into a mesh, the road mesh generation algorithm starts at one of the endpoints of the road and follows the curve. While doing this, at every unit step, based on the curve's direction, vertices, triangles and UVs for the mesh are generated. This idea is illustrated in Figure 13.

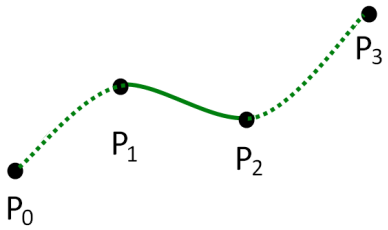


Figure 12a: Catmull-Rom Spline.

Figure 12b: Problem that can occur after generating the road mesh. The terrain can clip through the road.

Generating a road mesh like this already gives some nice results. However, there is still an issue to address. One can consider the scenario shown in Figure 12b. It is possible that due too the curvature of the terrain, some mesh clipping occurs. In order to solve this, the terrain height is changed below the roads such that the terrain is always below the road. In order to do this, for every coordinate in the terrain's heightmap, it is checked whether a road exists at that position. If a road exists at such a point, the heightmap is changed such that the terrain lies below the road.
Another functionality included in the roadmesh generation are curved roadends. When a road ends, it does not look good when it just suddenly stops. Hence, functionality was implemented such that these roadends have a nicely curved end. This can be seen in Figure 14. The figure also shows the triangles created in order to make the curve. Making these triangles is not too difficult, the real difficulty lies in the texture mapping. In order to make the texture go around like shown in Figure 14, every vertex on the curve was actually created twice such that for every triangle on the curve, unique UV values could be set. This allowed the proper texturing of the curve.
An example conversion from a simple roadmap to smooth road mesh can be seen in Figure 15.

5 Building Generation

This section describes the building generation component of the application. After a road map is generated, a graph data structure is made available to the building generator. In our implementation, two types of buildings are considered: skyscrapers and houses. Skyscrapers are tall buildings placed in highly populated areas and they cover complete blocks. Houses are smaller buildings placed in lower populated areas and multiple houses could

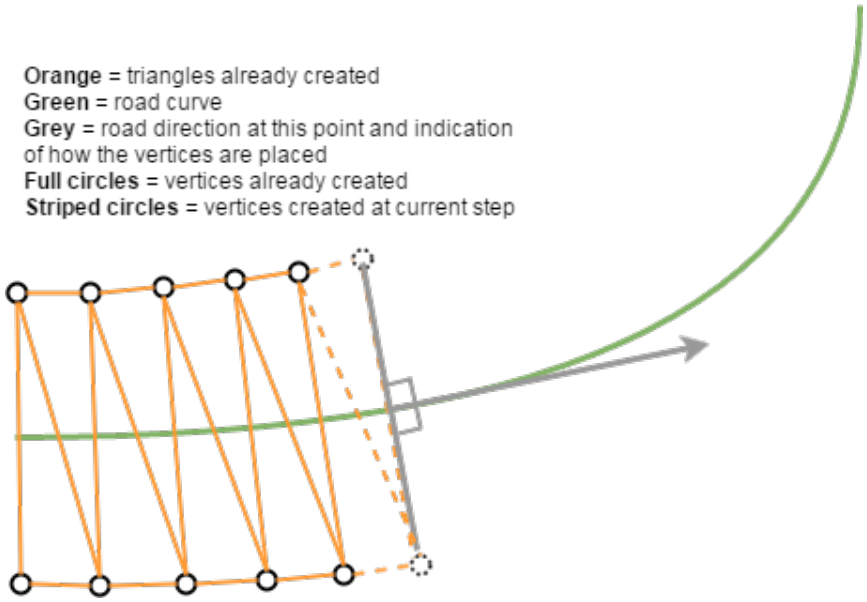


Figure 13: Topdown view on roadmesh generation, step in the process. Note that this figure was made by hand so vertex placement is not super accurate.

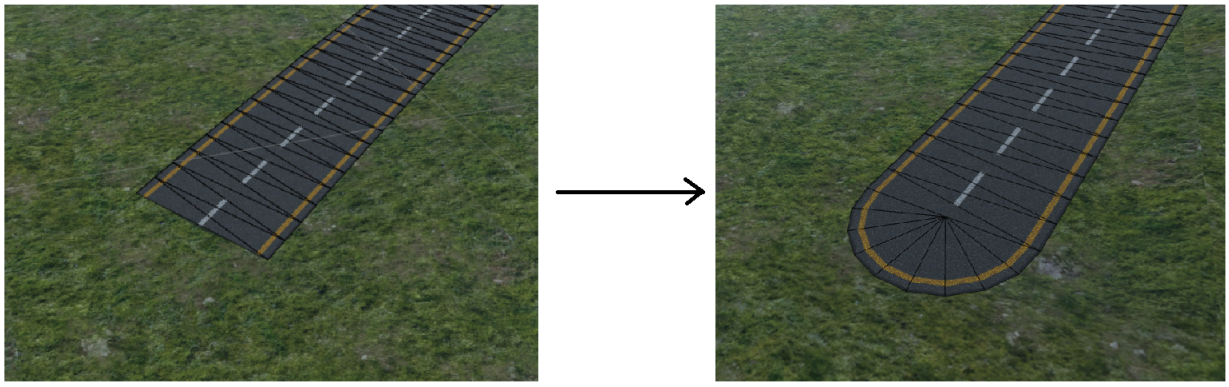


Figure 14: Illustrating how a smooth road endpoint is added to a road mesh.

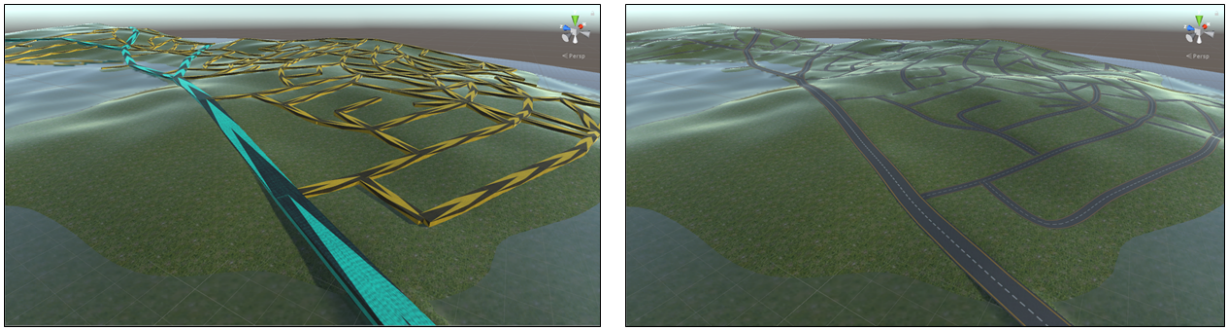


Figure 15: Conversion from a simple roadmap to a smooth road network

be placed in a block. Sections 5.1 and 5.2 below will describe the skyscraper and house generation respectively.

5.1 Skyscraper Generation

As mentioned before, skyscrapers cover complete blocks. So in order to start the generation of skyscrapers, blocks will need to be identified. Figure 16a shows some examples of blocks.

5.1.1 Block Generation

Using the road map graph, an algorithm is executed to find the polygons included in the road map. These will be called blocks. This algorithm performs the following steps:

1. Generate all tuples ($n1$, $n2$, $n3$) where $n1$, $n2$ and $n3$ are nodes, $(n1, n2)$ is an edge and $(n2, n3)$ is an edge, such that n2 connects both edges.

2. Sort these tuples on the angle between vectors $(n1, n2)$ and $(n2, n3)$. This is done such that tuples "going to the right" have smaller angle.

3. Now pick the first tuple and start traversing the graph by following the tuple's direction (i.e. go to $n3$). Check for all tuples (still in the tuple list) that have $n3$ as their middle node which tuple has the smallest angle.

4. Keep traversing the graph like this until we reach the starting tuple again. The nodes that were traversed form a polygon.

5. Remove the tuples that were traversed from the tuple list and go to step 3 again.

5.1.2 Block shrinking

After blocks have been generated, the next step is to generate actual skyscrapers on these blocks. Because the blocks found so far are based on the nodes of the roads, the blocks first need to be shrunk. In order to do this for a given block, the following steps are executed.

1. For every node in the polygon, compute the angle between the connected edges.

2. Then based on this angle, move the point along the vector between the connected edges towards the center. A large angle means small movement and a small angle means a lot of movement. Using some Pythogarian functions, this can be done with relative ease.

An illustration of this process is shown in Figure 16b

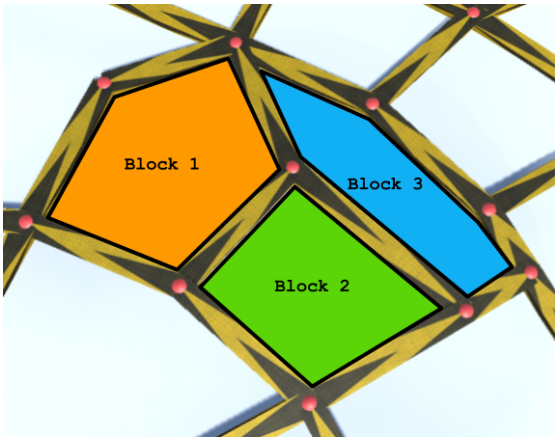


Figure 16a: Example of blocks generated from road graph.

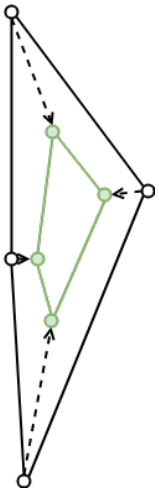


Figure 16b: Illustration of shrinking process. Black indicates old block and green the shrunk block. The striped arrows represent the vector between the connected edges. The length of this vector is related to the size of the angle between the connected edges.

5.1.3 Skyscraper Polygon Generation

Once all the blocks have been shrunk, the actual skyscrapers can be built. In order to generate a skyscraper, the shape and number of floors for the skyscraper needs to be specified. This is done by providing an array of 2D coordinates which represent a polygon outline. The first step in generating a skyscraper is generating its base. The base is used to make sure the building itself is always horizontal (Figure 17a). After that, for each edge in the specified polygon outline, a wall is procedurally generated. This is simply done by starting at the base and then generating a quad for every floor. And finally when all the walls are there, the roof will be constructed. This is done by using a triangulation algorithm.

5.1.4 Skyscraper Texture Generation and Window Placement

Before a skyscraper is procedurally generated, some random textures are picked. For every skyscraper part (base, floors, roof) there are three textures to choose from. Next to that, once a skyscraper has been generated, some windows can be added to the skyscraper. For performance reasons the windows are nothing more than a simple quad with a texture applied to it. Also the window type is picked randomly. A set of skyscrapers with random textures and windows can be seen in Figure 17b.

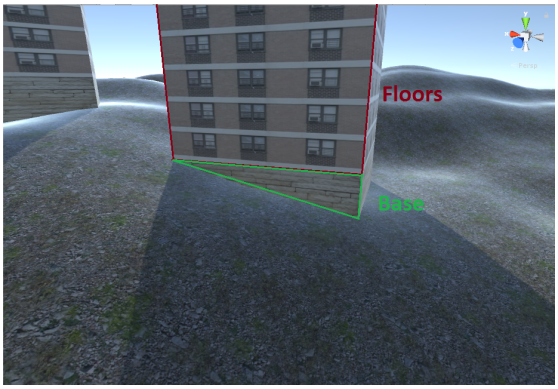


Figure 17a: Illustration of the building structure.

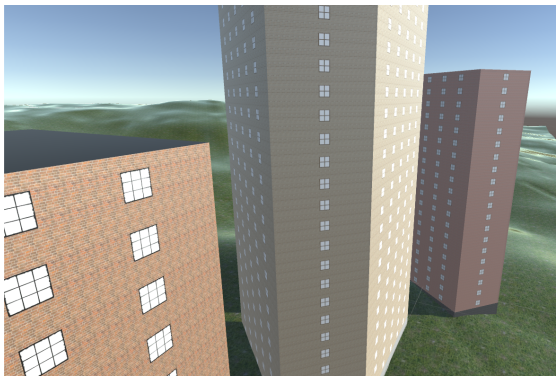


Figure 17b: Three skyscrapers, textures and windows are picked randomly per skyscraper.

5.2 House Generation

Unlike skyscrapers, houses are not necessarily placed in blocks. Houses can basically be placed anywhere, as long as they are connected to a road. The following sections describe the details of house generation.

5.2.1 House Construction

In order to build a house, its center, width, depth, direction and number of floors have to be specified. Based on these values, a complete house can be generated automatically.
Like skyscrapers, houses also have three parts: a base, floors, and a roof. The base and walls of a house are constructed in the same way as is done for skyscrapers. The roof is different however. Instead of a flat roof, houses have a pointy roof. Because of this, a house is always rectangular. To make the roofs a bit more interesting, their height is randomized within some bounds.
To construct the roof, the four highest corner points of the house are used as a reference. The main problem in the roof construction lies in proper texturing. Like was done for the road ends, vertices are duplicated in order to produce correct UV mapping.

5.2.2 Textures, Windows and a Doors

Like for skyscrapers, also for houses textures are picked randomly. For every house part (base, floors, roof) there are three textures to choose from. The same applies to windows, even if the way how they are placed is slightly different. More specifically, every wall is partitioned into sections, and on every section a window is randomly placed. Again, the window model is simply a quad which is picked randomly.
Finally, a house would not be a house without a door. Hence, based on the direction of the house, a door is placed on one of the walls. The door is placed at a random position and the door model is also randomized. Figure 18 shows three houses that were procedurally

generated.

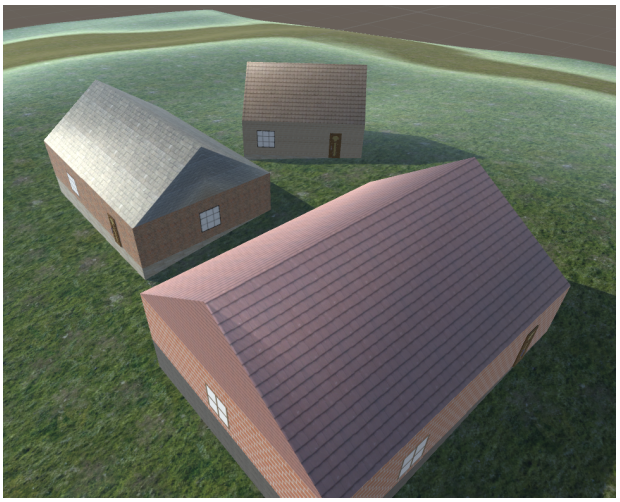


Figure 18: Three houses that were procedurally generated

5.2.3 House Placement

At this point, building a house is possible, but the location is still to be defined. The chosen approach simply tries (for every road in the network) to place a house on both sides of a road, such that the house is facing the road itself. Then each house should respect few constraints:

- it should not intersect with other houses (it can be simply checked using Unity colliders);
- it should not intersect with a skyscraper (also detectable by colliders);
- it should not intersect with a road, (also detectable by colliders);
- it should be placed at a position with suitable population density (specified by the population threshold in input).

If all constraints are satisfied, the house will be built; if not, its construction will be aborted.

5.3 Districts

Almost every city is divided into different districts. These differentiate from each other surely for the type and density of buildings, but these differences often relate to more complex factors like the population density and the road network: an industrial area is more likely to arise out of residential areas, preferably close to highways and are characterized by big lots with ample warehouses; a business district is often placed in the city center and is characterized by higher buildings like skyscrapers; finally, a residential area is generally right out of the city center and is characterized by houses and apartment buildings.

Keeping this in mind, different districts could be generated to increase the level of realism. However, a major challenge would be having actual models for different types of buildings (skyscraper, small house, warehouse, factory, etc.). We decided that district generation, for now, is out of scope. (see Section 8 for more information).

6 Results

The algorithm presented in the paper provides a lot of different methods of interpretation for the methods described. As the paper is quite short, it does not go into implementation details for the different subcomponents of the algorithm. Luckily there is a lot of additional research done on this topic, giving us information on the most efficient way to implement the components. The algorithm has a lot of different configurations, which limits its effectivity. Nevertheless, our algorithm runs in reasonable time not affecting the user's interaction with the system.

To show the capabilities of our algorithm, two actual cities have been re-generated using heightmaps, population and growth maps that are similar to the real ones. These two cities are New York (USA) and Krakow (Poland). The former has been chosen for its iconicity and its famous grid road pattern; the latter for itT marked Paris-style road pattern (actually more visible than in Paris itself) and a grid-style city center.he ressults can b e found in Figure 19 and Figure 20 respectively. Figure **??** gives an additional non-top-down look over the city of Krakow.

7 Discussion

The algorithm, although quite detailed, has a lot of room for improvement and limitations. For example, much more research can be done into the generation of buildings. Additionally, work can be put into making roads blend together more smoothly. Before the application can be used by actual game developers, these issues must certainly be addressed. We note that the algorithm has a considerable running time for some of the functionalities. This will make real-time city generation in applications not viable. Cities must therefore be generated before runtime and saved into the Unity scene. As cities can be quite large and contain many roads and buildings, more research can be done in storing the city's meshes in an efficient and fast way. Consequently, research can also be done in generation several levels of detail, enabling buildings to be rendered in a lower resolution if they have a greater distance to the scene camera. More improvements to the application will be discussed in the next section. At this point the final components of the tool are still in development. Because of this, there are no complete results yet. Once the tool is done, a complete discussion and evaluation will be provided here.

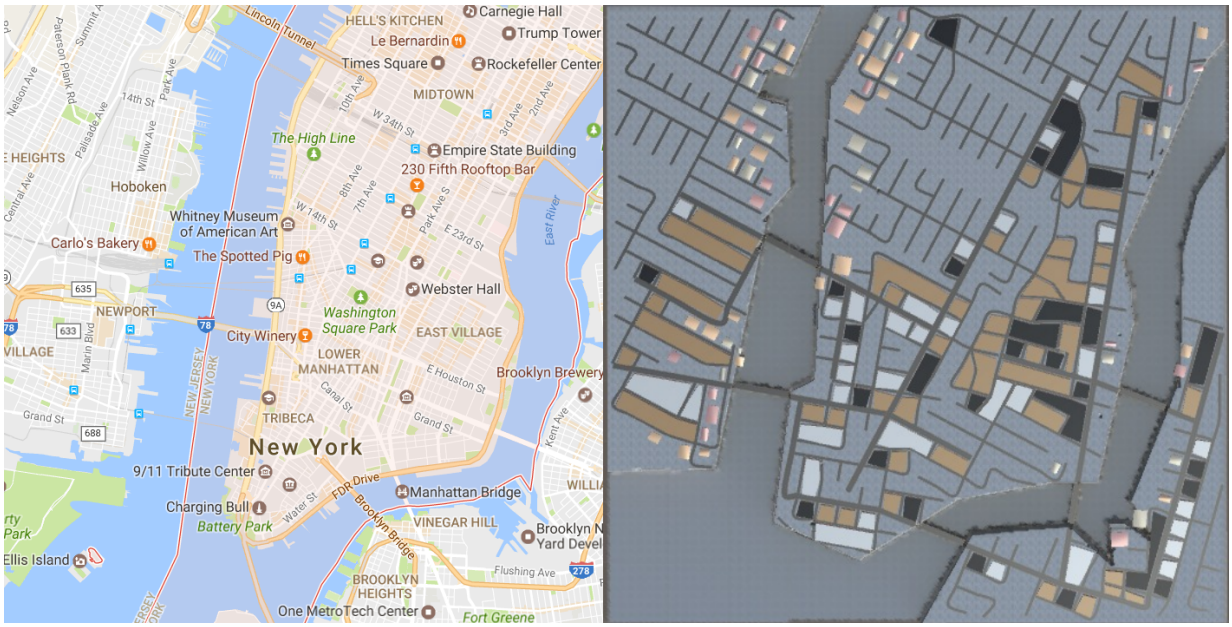


Figure 19: Results of the algorithm with height and population maps of Manhattan, New York. *New-York* has been clearly chosen as unique growth map. The algorithm manages to create a quite realistic result, building even bridges close to the real locations.

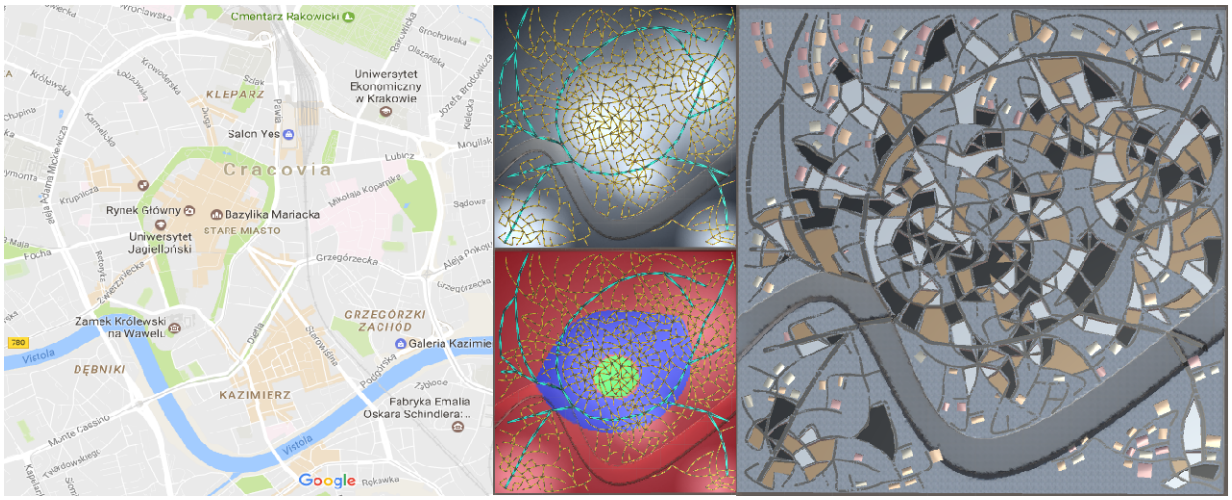


Figure 20: Results of the algorithm with height, growth and population maps of Krakow, Poland. The algorithm manages to create surprisingly realistic results, reaching all the important districts and building a highway around the main center.

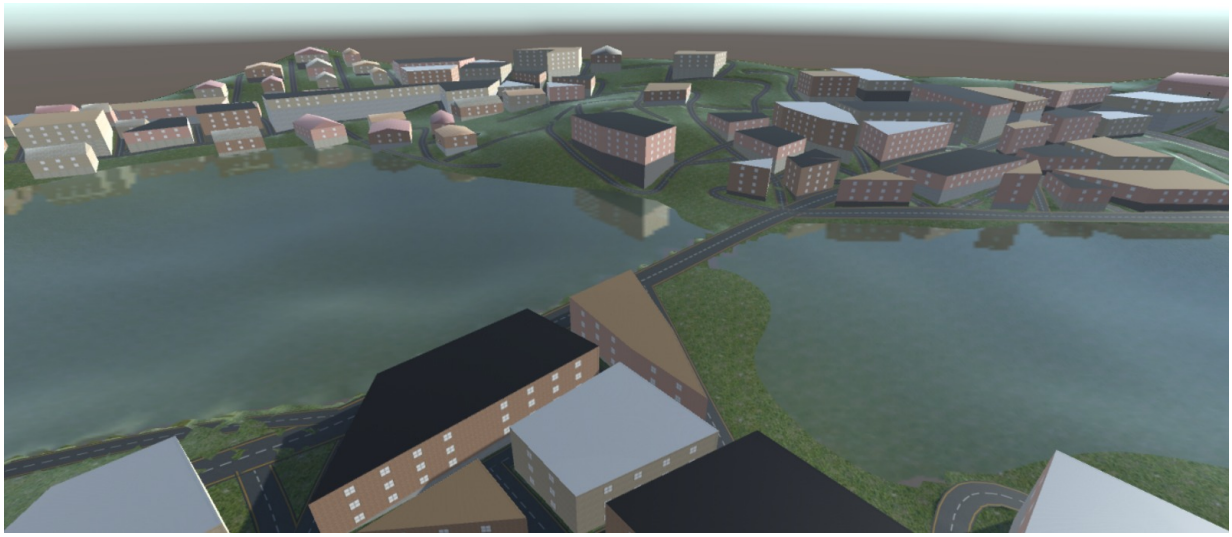


Figure 21: Result of city generation.

8 Future Work

8.1 Addition of Districts and Complex Roads

Generating a realistic city is a complex task and several assumptions and simplifications have been made because of time constraints. For this reason the future work aims at increasing the general feeling of reality. One possible improvement could be removing the assumptions made in Section 4.3 over the impossibility for roads to go through mountains or hills. Also, a wider selection of models for buildings could be implemented to increase the variety of the buildings and more complex rules used to determine the division of the map into districts according to levels of population, size of the blocks and other factors.

8.2 Improved Block and Lot Generation

The process of block and lot generation can be improved significantly. Blocks can sometimes be unrealistic in terms of shape and size. As suggested by a member of the Visualization group at Eindhoven University of Technology, additional post-processing on the roadmap graph can be done to overcome these issues. For example, small blocks can be merged together by removing edges from the road graph. Large blocks can then also be split by seperating them by another edge. As mentioned before, more attention can be put into dividing blocks into lots, such that one block can have multiple lots. This generates a much more realistic setting used in other city generators.

8.3 City Navigation

A big advantage of the chosen approach is that our roadmap is defined by a doubly-linked list graph structure. This enables pathfinding algorithms such as Dijkstra's Algorithm or A-star to be used to find shortest paths within the city. Using the shortest path, a moving camera can be created that follows the path from source to destination. This also enables virtual cars to drive around the city, following the road fragments using a pathfinding algorithm to determine their routes. This could provide an interesting addition increasing realism within a virtual city.

9 Conclusions

The chosen approach has proven to generate quite realistic results in terms of variety of roadmaps generated, applicable to any terrain. Our choice for customization options provides a lot of freedom for the user to change the generated city according to their preferences. The downside of the customisability of the application is that, with around 50 parameters to modify, the application becomes quite complex to use for novice users. When making the weigh-off between customisability and ease-of-use, we decided that for our purpose customisability was the main goal. In order to be usable by developers, the GUI of our application can be simplified and further documented such that entire cities can be generated by the click of a button. Further customisability concerning texturing is also a must-have for future use.

10 Bibliography & References

[**1**] Yoav I. H. Parish Pascal Mller, Procedural modeling of cities, Proceedings of the 28th annual conference on Computer graphics and interactive techniques, p.301-308, August 2001

[**2**] Lechner T., Watson B., Wilensky U. and Felsen M., Procedural City Modeling (2003)

[**3**] Groenewegen S. A., Smelik R.M., de Kraker K. J. and Bidarra R., Procedural City Layout Generation Based on Urban Land Use Models, EUROGRAPHICS, 2009

[**4**] G. Nishida, I. Garcia-Dorado, and D. Aliaga, Example-Driven Procedural Urban Roads, Computer Graphics Forum, Volume 34, Number 2 pp. 113, 2015

[**5**] Chen, G., Esch, G., Wonka, P., Mller, P., Zhang, E. 2008. Interactive Procedural Street Modeling. ACM Trans. Graph. 27, 3, Article 103, 2008

[**6**] Ken Perlin, An image synthesizer, Proceedings of the 12th annual conference on Computer graphics and interactive techniques, p.287-296, July 1985

[**7**] Catmull, E. and R. Rom, A Class of Local Interpolationg Splines, in Barnhill R.E. and R.F. Riesen-fled (eds.), Computer Aided Geometric Design, Academic Press, New York, 1974.

[**8**] L-Systems considered harmful, `http://nothings.org/gamedev/l_systems.html`, accessed on 14-03-2017