Authors' copy - Original version published in: Computers & Graphics, Volume 44, November 2014, p. 33 - 44.

Stateless Generation of Distributed Virtual Worlds

Jiri Danihelka, Lukas Kencl and Jiri Zara

Czech Technical University in Prague, Faculty of Electrical Engineering
{danihjir, kencl, zara}@fel.cvut.cz

Abstract

We present novel techniques for implementing possibly infinite on-demand generated 3D virtual worlds in distributed environments. Our approach can be useful in two scenarios: 1. A multiuser virtual world with mobile clients with sufficient CPU and GPU power but with limited network speed. This reflects current mobile phones, tablets and laptops in areas without a high-speed mobile connection or Wi-Fi connectivity. 2. Virtual world on-demand generation in a cloud environment that would be useful for scalable massive multiplayer games. If multiple independent generators create areas that are overlapping, our method ensures that the intersection of the areas will contain the same geometry for all of them. For this reason, we call our method *stateless generation*.

Keywords: graph algorithms, path problems, distributed graphics, computational geometry, urban modeling

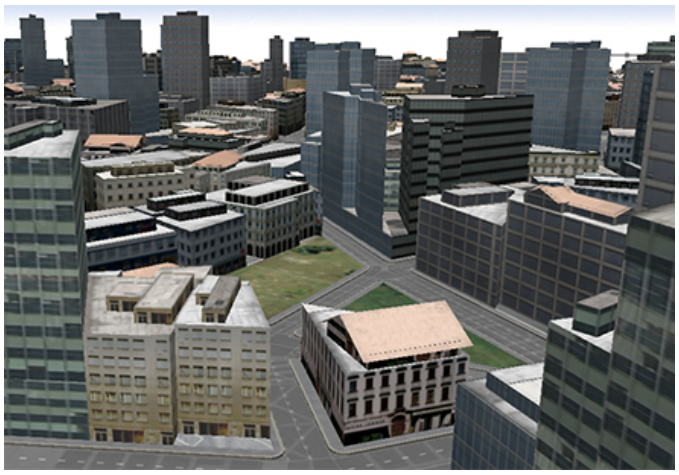

Figure 1: A stateless infinite city generated by our method

1. Introduction

Much attention is currently focused on multi-user virtual environments hosted in the cloud for both gaming and non-gaming purposes [1, 2, 3]. With the arrival of massive multiplayer online games, game creators have had to deal with limited server capacity in terms of world size or number of players [4], but virtual-world services must be scalable [5]. Current cloud computing technologies are able to provide additional resources on-demand, but virtual-world systems are rarely able to generate on-demand game content (to save memory for world parts not needed at the moment). Another problem is the limited network connectivity of mobile clients in cellular networks, which is often too slow for downloading the content generated on the server, thus having a highly negative impact on the emerging and ubiquitous mobile gaming.

Our method is innovative in that it eliminates the need to synchronize the static content that was procedurally-generated on multiple devices. This will allow virtual-world servers to dedicate additional machines from the cloud environment to parallel content generation, or even to generate content on client devices. Because the content is generated on-demand, the virtual world can be considered theoretically infinite.

We refer to our method as *stateless generation* because it allows for locally generating only the content of the world that is relevant to the viewing frustum of the clients, and this content is generated independently, without knowledge of the states of the other generators.

As one of the most difficult virtual environments to generate, our efforts focus on generating an urban landscape (see Figure 1). City environments are generally complex because they are structured and are both detailed and enormous. Procedural generation is a convenient tool for saving storage space and/or Internet bandwidth. When in the view frustum, buildings can be created on-demand from their lots (land parcels) using generating grammars.

Moreover, our method is general enough to be applicable to other types of structured landscapes (e.g., countryside, caves, labyrinths). Structured landscapes are generally more difficult to generate than unstructured landscapes (e.g., forests).

We provide a general purpose guideline for scalable algorithms generating virtual worlds that is applicable to most types of landscapes. We formalize the requirements and constraints such algorithms must fulfill. We then provide a novel algorithm for generating an infinite and scalable city-street layout, including a novel sub-algorithm for generating streets in a constrained environment, which could also be useful in traditional approaches to procedural city generation [6].

2. Related work

The most advanced approach for procedural building generation was published by Müller et al. [7] in 2006, improving upon the previous method by Wonka et al. [8] in 2003. The lot and street geometry can also be generated procedurally. The first such algorithm for finite cities was published by Parish and Müller [9].

In 2003, Geuter et al. [10, 11] presented an algorithm for the on-demand generation of infinite cities in a regular rectangular grid. In their approach, the street network has to be aligned with the main axis, and all building lots must have the same square shape and size (see Figure 2). The visible buildings are determined and procedurally generated according to the viewing frustum. Each building lot is assigned an integer number according to its coordinates using a hash function. This number is used as another seed for the pseudo-random building generation of that building lot. Some of these ideas are applied and extended in our approach.

A method for real-time generation of detailed procedural cities from GIS data was published by Cullen and O'Sullivan [12]. Their system uses a client-server approach, allowing multiple clients to generate any part of the city without requiring the full data-set. It creates the building geometry on-demand from the provided lot database and, in contrast to our work, does not address street and lot generation. Vanegas et al. [13] presented an interactive method for procedural generation of city parcels. They generate spatial configurations of parcels similar to real-world cities and support consistent lot locations relative to their containing blocks. Their approach generates parcels highly similar to those observed in real-word cities, but it mainly focuses on parcel layout and does not address all the phases of the city-generation process, unlike our method.

Aliaga et al. [14] presented a system for synthesizing urban landscapes by example. They proposed a random walk algorithm for obtaining parameters from existing cities that are later used in the generation process. Their system was somewhat capable of on-demand generation, but it was neither intended nor suitable for distributed environments because it required knowledge of all previously generated geometries for each future step.

On-demand world generation is highly related to texture synthesis algorithms. The main difference between these two approaches is the use of the generated results and whether the algorithm generates vector or raster output. Algorithms for texture synthesis usually use Voronoi diagrams [15] of randomly distributed points. One of the pioneering works in this area was published by Worley [16], who uses a function that complements Perlin fractal noise to produce textured surfaces resembling flagstone-like tiled areas, an organic crusty skin, crumpled paper, ice, rock, mountain ranges, and craters. Our algorithms are inspired by his function to determine the n^{th}-closest points that affect the structure of the texture at the currently generated area. We aim for a similar goal, but we use Delaunay triangulation instead. We also use techniques based on Voronoi diagrams to divide areas that are affected by different geometrical elements.

Figure 2: Previous approach in infinite-city rendering published by Greuter et al. [10, 11] showing street level view. Note the regular rectangular shape of the street network.

Liang et al. [17] presented another algorithm for synthesizing textures from an input sample in real-time, but they use a significantly different approach than ours and their results are not isotropic, which is important for stateless generation. Texture synthesis can also be used to generate street patterns [18, 19], but these works use different approaches that are difficult to adjust for the purposes of infinite cities. Lefebvre and Hoppe [20] presented an algorithm for parallel on-demand texture synthesis based on a neighborhood matching approach. Their scheme defines an infinite, deterministic, aperiodic texture from which rectangular views can be computed in real-time on a GPU. Another advance in infinite texture generation was made by Cohen et al. [21]. They utilized a small set of Wang Tiles to tile a plane non-periodically. Using a proper tile set, the texture can be extended on-demand. In 2007, Merrell [22] presented an algorithm for generating 3D buildings and cities from a set of 3D tiles. Merrell's later work [23] was focused on continuous city model synthesis. These techniques are, however, limited to structures aligned with the main axes.

To generate a realistic world structure, we must first analyze examples to acquire characteristics that are later used in procedural modeling. Important progress in inverse procedural modeling was made by Stava et al. [24]. They create parametric context-free L-systems that represent an input 2D model. Their approach is based on vector shape recognition, clustering in the transformation spaces and detecting structures as L-system rules. Elements and structures can be edited by changing the L-system parameters.

Our approach follows up on the above works, combining their benefits and removing some of their limitations. Unlike [10, 11, 22, 23], our building lots can have various sizes and shapes, streets can be arbitrarily oriented, and the street network is not periodic. Unlike [14], our approach adds capabilities for distributed environments as well as the ability to generate only content related to the view frustum of the client.

3. Stateless Generation Approach

We have laid down the following general requirements for our world generator:

1. The generated world is infinite and is not periodic.
2. Clients and/or servers are able to generate the static part of the world on-demand; they do not have to download

it from a single (common) place. They should download only one random generator seed (or hash function) for the whole world.

3. Generation of the world can start from any point. The client will generate only those parts that are relevant (e.g., visible) to it.

4. The generation process is deterministic (usually achieved using pseudo-random generators). The results are always identical, regardless of the starting point or the area relevant to the client.

If a generator fulfills the above requirements, we call it a *stateless generator*. This is because its results do not depend on the results (or inner states) of other generators working in parallel or on its previously generated content. The defined requirements have the following outcomes:

a. Generated parts of the world can be cached. When any part of the world is cleared from the cache, it can be re-generated again, and it will be exactly the same as the original.

b. When multiple clients and/or servers generate the world, the intersected areas are exactly the same, and the generated virtual worlds connect seamlessly, despite that they started generating at different starting points and have not communicated with each other.

We assume that the world is infinite in two dimensions only and consists of buildings or other objects on an infinite plane. The 3D model is created later from the generated layout on the plane. We provide both a general purpose guideline and a specific version for generating the infinite urban landscape. The specific algorithm follows the standard city-generation workflow proposed by Parish and Müller [9], initially designed for finite cities. The workflow starts with the city street network, and buildings are generated afterward (phases: a. street network, b. building blocks/lots, c. building geometry). We explain our approach to city street-network generation, while the other phases remain nearly the same as in the case of finite cities. When working with infinite structures, we assume that we will compute only those values that are needed due to the intersection with the view frustum.

3.1. Street-network characteristics

To achieve the desired city appearance, our algorithms use parameters that can either be provided by the user or measured from an example of a city map. We will denote these parameters as street-network characteristics.

When processing a map of an existing city, streets are replaced by straight segments between street junctions. These segments are called street segments. Street segments are also created during the generation process, and they are converted to streets in the final phase of the algorithm.

To acquire the street-network characteristics from a map, we use the random walk algorithm as described by Aliaga et al. [14] applied to city-map street segments. A detailed description of the street network characteristics and the corresponding terminology we adopt can also be found there. For our purposes, the street network characteristics contain the following parameters:

1. average length of a street segment and its variance

2. average unoriented angle between two consecutive street segments and its variance

3. average number of street junctions on a surface unit (e.g., a square kilometer)

4. average number of street segments on a surface unit

5. street tortuosity – the average ratio between street length and the corresponding street segment length

Our algorithms make an effort to ensure that the generated street network will have similar network characteristics to those of the original city map, but currently the resulting distribution exactly corresponds only in mean values.

4. General algorithm for stateless infinite worlds

This section describes general guidelines for designing algorithms for stateless infinite worlds. Later, we show how we apply this algorithm template to urban landscapes.

The algorithm input consists of items shared among generators (in the case of multiple generators): the generator parameters (e.g., street-network characteristics); the tessellation grid parameter d - as described below; and the hash function for the pseudo random generator. Other input parameters are individual per each generator: the position, orientation and field-of-view of the camera in the infinite world, with near and far clipping plane distance (this is usually called the *view frustum*).

The algorithm output is the geometry of objects within the view frustum. Note that because the algorithm works in a 2D plane, we work with view frustum projections to the plane rather than the corresponding 3D representation. The shape is not important for the algorithm; it works with any bounded 2D area.

Each step of the algorithm is now described in greater detail below:

Step 1 – Tessellation: Create a non-periodic tessellation of the infinite plane. The tessellation is only an auxiliary structure not visible in the final model. Subdividing the plane into a set of bounded areas (called tessellation fragments) allows us to transform the problem into sub-problems of generating content only in the areas intersected with the view frustum. In Section 5, we provide a novel sub-algorithm for the tessellation of an infinite plane into a non-periodical set of arbitrarily oriented triangles. We prove that this algorithm fulfills the requirements for stateless generation.

Step 2 – Generation of interfaces: Use a pseudo-random generator to define interfaces (objects that cross the fragment boundary) between the adjacent tessellation fragments. In Section 6, we provide a sub-algorithm that generates a street layout for an urban landscape, where the interfaces are the street-segments that intersect tessellation fragment boundaries.

Step 3 – Constrained generation: Generate the inner part of the fragments while preserving the constraints set by the interfaces. In Section 7, we provide another novel sub-algorithm for generating a street network in a constrained environment that solves this problem for street layouts.
Step 4 – Procedural generation: Use traditional procedural generation to finish the landscape. This is the only step performed in 3D.

5. Tessellation algorithm

This algorithm divides the infinite plane into a tessellation of triangular tessellation fragments. It provides only those fragments that are fully or partially within the client's view frustum.

The algorithm input parameters are the view frustum; a hash function; and the grid size parameter d. As its output, the tessellation fragments that intersect the frustum are provided. Note that the geometry of the fragments remains consistent in the case of multiple overlapping queries.

First, we present a theoretical approach that operates with infinite structures. Then, we provide practical instructions about how to correctly generate the content within the view frustum using limited computational resources. We achieve this by omitting data that do not impact the content of the view frustum. Note that, due to the connected structure of the world, even objects near but outside the view frustum can affect objects inside the view frustum; we have to take this into account. The approach is similar to the lazy-evaluation method used in functional programming languages (e.g., Haskell) to work with infinite data structures.

5.1. Theoretical approach

Theoretical step 1: Suppose we have a plane dedicated for generating the world. Create a regular infinite square grid on the plane. (The segment length of the grid should be a configurable generator parameter. Let us denote the length as d.) We denote the grid as the tessellation grid. The grid will divide the infinite plane space into squares. Each of these squares has an X and Y integer coordinate (positive or negative). Apply a hash function to each pair of coordinates to acquire a pseudo-random generator seed for each square (see Figure 3). We use universal hashing functions for integer vectors of size 2.
Theoretical step 2: Use the seed to generate a pseudo-random position inside the given square (with uniform distribution) and place a node on that position. We will refer to these nodes as tessellation nodes.
Theoretical step 3: Create a Delaunay triangulation using the tessellation nodes. All tessellation nodes are connected with triangulation edges to form an infinitely large tessellation. In Section 5.2, we prove that Delaunay triangulation always produces the same results, regardless of its starting area, and can be performed partially only using a finite subset of tessellation points for any bounded area. We use the triangulation technique described below to find the proper edges.

5.2. Triangulation

We must select a robust and unequivocal triangulation method that requires knowledge only of the local surroundings of the processed nodes, as an infinite number of nodes cannot fit into memory. Delaunay triangulation has these desirable properties for stateless generation.
Definition of Delaunay triangulation:
Three nodes form a triangle that belongs to Delaunay triangulation if and only if there are no other nodes inside its circumcircle.

We will now prove that Delaunay triangulation has properties of stateless generation. We will start with several lemmas.

Lemma 1: In a tessellation grid, any circle with a radius greater than $\sqrt{2}d$ has at least one tessellation node inside it.
Proof: Such a circle contains at least one whole grid square inside. Therefore, it also contains a tessellation node corresponding to that square (see Figure 4).
Lemma 2: In a tessellation grid with Delaunay triangulation performed on its tessellation nodes, no triangulation edge may be longer than $2\sqrt{2}d$.
Proof: According to the definition of Delaunay triangulation and lemma 1, every triangulation edge has to be part of a triangle with a circumcircle radius smaller than $\sqrt{2}d$. Edges greater than $2\sqrt{2}d$ will not fit into the circumcircles.
Lemma 3: If we remove some nodes from a Delaunay triangulation and perform a new Delaunay triangulation on the reduced set of nodes, all triangles from the original triangulation that were not using any of the removed nodes will be present in the new triangulation. (Some new triangles may emerge, but that is fine.)
Proof: It is an outcome of the definition of Delaunay triangulation. (No new nodes will appear inside the original circumcircles.)
Lemma 4: When we have Delaunay triangulation performed on tessellation nodes of a tessellation grid and a bounded area, removing the nodes that are farther from the area than $2\sqrt{2}d$ will have no effect on the triangles that intersect with the area.

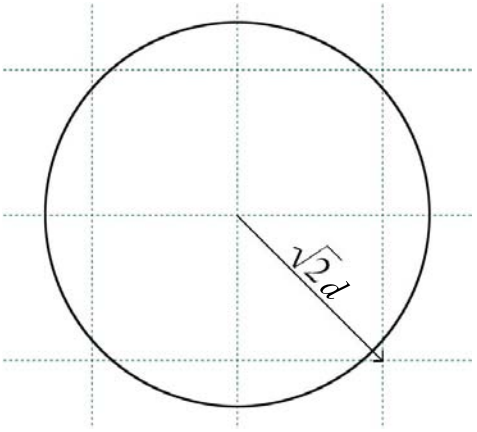

Figure 4: The circle has a radius slightly less than $\sqrt{2}d$ and does not contain any of the squares of the infinite grid. It is obvious that a circle with a radius greater than $\sqrt{2}d$ contains a whole square and its tessellation node.

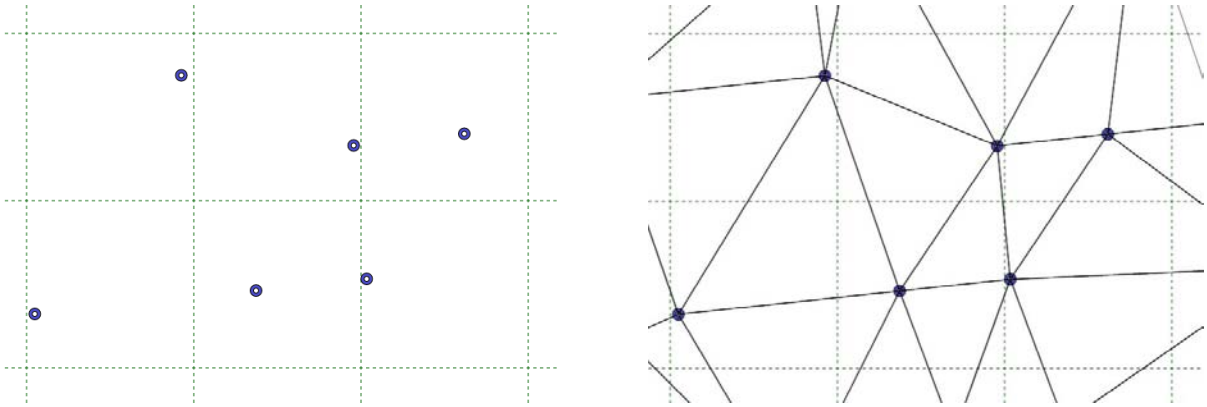

Figure 3: **Tessellation algorithm** Left: An infinite tesselation grid (step 1) with pseudo-randomly generated tessellation points (step 2); Right: Infinite Delaunay triangulation that forms tessellation fragments (step 3); These structures are only auxiliary for further phases and do not represent the final city geometry.

Proof: It is a direct outcome of lemma 2 and 3.
Theorem: The Delaunay triangulation performed on tessellation nodes of a tessellation grid fulfills the definition of stateless generation described in Section 3.
Proof:
We will follow the definition from Section 3:
1. The structure is infinite and is not periodic.
2. The triangulation can be generated by multiple generators independently.
3. Parts not relevant to a bounded area can be removed using lemma 4.
4. The process is deterministic. □

Delaunay triangulation is unique except for situations where there are 4 nodes on a circle. This, however, happens with negligible probability for randomly generated points. We use computation with particular high-precision real numbers, which substantially decreases the probability of such cases and provides the same results on different operating systems and CPUs.

5.3. Practical approach

This approach describes how to generate the part of the infinite tessellation that is within the view frustum.
Step 1: Take the view frustum and enlarge it by $2\sqrt{2}d$ in all directions. (Due to lemma 4 from the previous section, it is guaranteed that triangulation inside the view frustum will not be affected by content outside the enlarged view frustum.)
Step 2: Take all squares that intersect with the enlarged view frustum and generate their corresponding tessellation nodes. Filter out the tessellation nodes that are outside the extended frustum. We now have all the tessellation nodes that are within the extended view frustum.
Step 3: Create Delaunay triangulation on those nodes. Lemma 4 guarantees that we generate all triangles that intersect the view frustum correctly. Filter out triangles that are completely outside the original frustum (these may not be generated correctly). We have now acquired the part of the tessellation that is visible in the view frustum and conforms to the stateless-generation properties.
Complexity:
The most complex step of the algorithm is the triangulation. The well-known DeWall algorithm [25] creates a Delaunay triangulation for n nodes in $O(n \log\log n)$ time. However, we use the previous incremental algorithm for Delaunay triangulation [26], with time complexity $O(n \log n)$, which allows us to include additional nodes in the triangulation once the initial set is finished. This is a nice feature in cases when the user and his or her view frustum move and thus the visible area changes over time. Each additional node can be added in $O(\log n)$ time.

6. Generating the tessellation fragment interfaces

In the case of an urban landscape, the interfaces consist of street segments that cross the tessellation fragment boundary. These street segments form constraints that guarantee a continuous street network between the tessellation fragments.

The generation takes as its *Input* the processed tessellation fragment together with the tessellation fragments adjacent to it, plus the street-network characteristics. Its *Output* is the street segments that cross the boundary. The street segments do not cross each other and each of them intersects the target tessellation fragment and exactly one other tessellation fragment.

First, we must create a seed for the pseudo-random generator to generate random numbers consistently. We create one pseudo-random generator for each boundary edge. To do this, we take the coordinates of the grid squares in which the tessellation nodes were created (4 integers from two squares related to the edge endpoints) and apply a hash function to them to find the seed. We use the hashing function

$$h = [c_1(x_1 + x_2) + c_2(y_1 + y_2)] \mod c_3$$

where c_1, c_2, c_3 are random large prime number constants. This ensures that the results are the same when we start generating from the other tessellation fragment and that the transition of the street network is seamless.

For each of the boundary edges we use the Poisson distribution to determine the number of street segments that cross the edge. The distribution mean is set proportionally to the product of the boundary-edge length, of the average street-segment length and of the average number of street segments on a unit

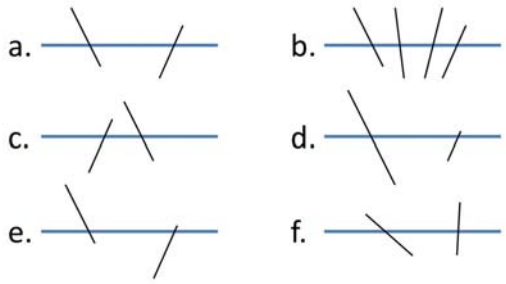

Figure 5: Variance in street segment interface generation – (a) reference case, (b) different numbers of street segments, (c) different crossing points, (d) different lengths, (e) different portion of street segments in the target fragment, (f) different angles

surface to reflect the original network characteristics. For each street segment, we generate a point on the boundary edge where the street segment crosses the edge (uniform distribution), the length of the street segment (normal distribution according to the average street segment-length and variance), the portion of the street segment that will be in the target fragment (uniform distribution) and the angle between the street segment and the edge (the distribution is proportional to the sine of the angle). See Figure 5 for examples.

Each of the generated street segments is then tested for the following conditions:

1. The street segment does not cross any previously generated street segment for this boundary edge.
2. The endpoints of the street segment are closer to its boundary edge than to any other boundary edge. (see Figure 6)

If the street segment does not fulfill the conditions above, we discard it and generate a new one instead.

7. Constrained street-generation algorithm

This algorithm generates the interior of the tessellation fragment with respect to the constraints set by interfaces that consist of street segments. Currently, the algorithm can be used only for a street network with no superimposed pattern. The algorithm will not produce rectangular street networks (New York, Chicago) or radial to center street networks (Paris). Those and other types of street network patterns are described in [9]. It is possible to create another stateless generation algorithm for each of these patterns, but it seems to be difficult to find a universal algorithm for all of them due to the already fixed positions of the tessellation fragment interfaces generated in the previous phase.

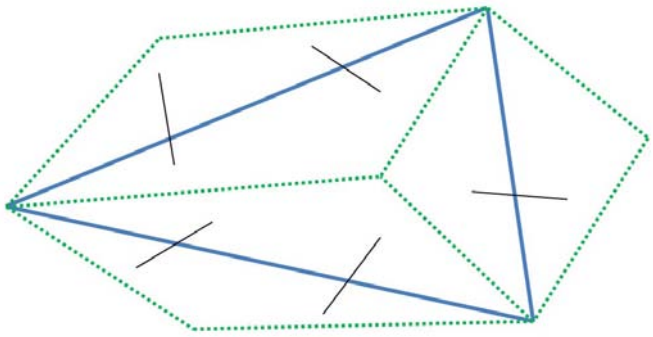

Figure 6: A tessellation fragment (blue) and its generated interface street segments (black); The endpoints of the street segments must be closer to its boundary edge, i.e., they must be within the dotted green boundary.

The algorithm *Input* parameters are the following: a convex shape (i.e., the tessellation fragment) with street segments crossing its boundaries (from the previous phase); and a pseudo-random generator (from the coordinates of the tessellation nodes of the triangle). The algorithm works generally for any convex shape, although in practice we use triangular shapes generated by the previous tessellation algorithm. Its *Output* is the generated street network inside the shape that takes the interfaces into account.

Each step of the algorithm is shown in Figure 7. First, we use the pseudo-random generator to place nodes that will be junction candidates. We use Poisson disk distribution for this task, and for simplification, the radius is equal to one-fourth the average street-segment length from the street-network characteristics. The end points of the interface street segments are also junction candidates. Next, we connect each pair of junction candidates with an edge (which we call a street-segment candidate) except for the following: (i) those that are too improbable according to the average street-segment length and its variance in the street-network characteristics (exceeding a predefined threshold) and (ii) those that would cross the interface street segments. We then add the interface street segments to the set of street-segment candidates. To generate the street network, we incrementally add the paths of street-segment candidates into the street network. We try to add paths that are in accordance with the street network characteristics. To do this step, we create an evaluation function to determine the appropriate paths to be added. Using this function, we transform our problem to finding the shortest path in a weighted graph.

7.1. Street-segment path evaluation

We now must define a set of evaluation functions f for street segments, angles and paths. For realistic street generation, we have to create functions that assign a low value to paths according to the street-network characteristics. For street segments ($e \in E$) and angles between two adjacent street segments (α), we have defined the evaluation functions as follows:

$$f(e) = \frac{1}{f_x(length(e))} \quad (1)$$

$$f(\alpha) = \frac{1}{f_x(\alpha)} \quad (2)$$

where f_x is the probability density from the street network characteristics (normal distribution based on the average length/angle and its variance)

For paths, we have defined the path evaluation function $f(P)$ as follows: Let $P = (e_1, e_2, ..., e_n)$ be a path, then its evaluation

is defined as:

$$f(P) = B(e_1)B(e_n)[f(e_1) + f(\alpha_{1,2}) + f(e_2) + f(\alpha_{2,3}) + \cdots + f(\alpha_{n-1,n}) + f(e_n)] \quad (3)$$

where $B(e)$ is a penalty for path-ending street segments based on the degree of the end nodes (i.e., junctions) of the path (the number of incident street segments already added to the street network). Note the difference between added street segments and street-segment candidates.

$B(e) =$ 10 for $degree = 0$ (i.e. dead end)
$B(e) =$ 1 if e is an interface street segment
$B(e) =$ $degree$ for $degree > 0$, not an interface

The purpose of the penalty function is to prefer paths that connect street-segment interfaces and to reduce the number of paths with dead ends. The values of the penalty function are empirical.

Our goal is to find a path in G with the minimum sum of evaluation values on its edges and angles. Because the common path-finding algorithms do not work for values on angles, we transform the original graph G to its dual form, called the line graph $L(G)$. This transforms angles to edges. After we find the evaluation for paths in $L(G)$ using a standard pathfinding algorithm, we perform a reverse transformation to obtain the path evaluations in graph G.

The line graph (also called the edge-to-vertex dual graph) $L(G)$ represents the adjacencies between the edges of G, and it has the following properties:

1. Each node of $L(G)$ represents an edge of G.
2. Two nodes of $L(G)$ have a common edge if and only if their corresponding edges in G share a common node.

Let us define the evaluation function f for the edges in the line graph:
Let $e^L_{a,b}$ be an edge in $L(G)$ corresponding to a pair of adjacent edges e_a and e_b in G.
Let $\alpha_{a,b}$ be an angle between the edges e_a and e_b
Let us define the evaluation function for the edge in the line graph $f(e^L_{a,b})$ as:

$$f(e^L_{a,b}) = \frac{1}{2}f(e_a) + f(\alpha_{a,b}) + \frac{1}{2}f(e_b) \quad (4)$$

Let $P^L = (e^L_{1,2}, e^L_{2,3}, \ldots, e^L_{n-1,n})$ be a path in the line graph; we define $f(P^L)$ as:

$$f(P^L) = f(e^L_{1,2}) + f(e^L_{2,3}) + \cdots + f(e^L_{n-1,n}) \quad (5)$$

Let us substitute $e^L_{a,b}$ using formula 4:

$$\begin{aligned} f(P^L) \quad = \quad & \left(\frac{1}{2}f(e_1) + f(\alpha_{1,2}) + \frac{1}{2}f(e_2)\right) + \\ & \left(\frac{1}{2}f(e_2) + f(\alpha_{2,3}) + \frac{1}{2}f(e_3)\right) + \cdots + \\ & \left(\frac{1}{2}f(e_{n-1}) + f(\alpha_{n-1,n}) + \frac{1}{2}f(e_n)\right) \end{aligned} \quad (6)$$

We simplify the expression to:

$$f(P^L) = \tfrac{1}{2}f(e_1) + f(\alpha_{1,2}) + f(e_2) + f(\alpha_{2,3}) + f(e_3) + \cdots + f(e_{n-1}) + f(\alpha_{n-1,n}) + \tfrac{1}{2}f(e_n)$$

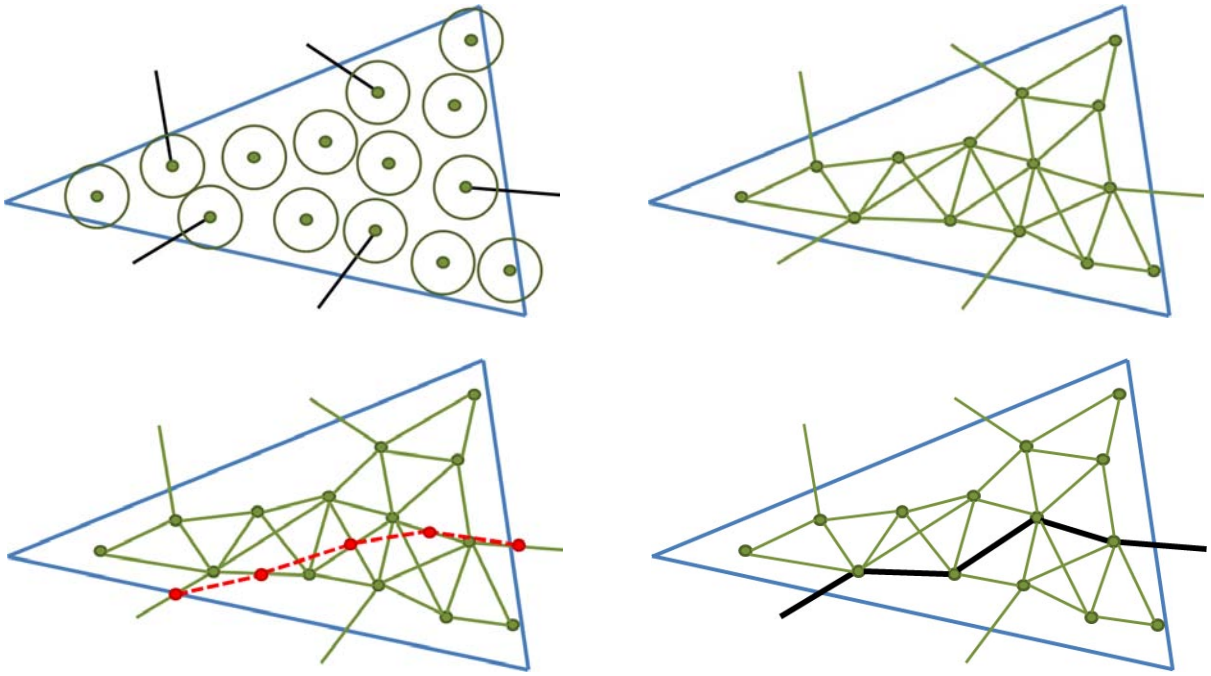

Figure 7: **Constrained street-generation algorithm** Top-left: A tessellation fragment (blue) with street network interfaces (black) during the phase of generating junction candidates using Poisson disks Top-right: Street network segment candidates graph (green) Bottom-left: The found dual path P^L (red, dashed) with minimum evaluation in the line graph. Bottom-right: Path P (black) added to the street network; crossing street-network candidates have been removed.

We use the substitution from formula 3:

$$f(P^L) = \frac{f(P)}{B(e_1)B(e_n)} - \frac{1}{2}\left(f(e_1) + f(e_n)\right) \tag{7}$$

We express $f(P)$ from the previous formula:

$$f(P) = \left(f(P^L) + \frac{f(e_1) + f(e_n)}{2}\right) B(e_1)B(e_n) \tag{8}$$

7.2. Minimum evaluation path-adding algorithm

1. From a given graph G, create its line graph $L(G)$.
2. Using the Floyd-Warshall algorithm, find the path with the smallest evaluation $f(P^L)$ between each pair of nodes in $L(G)$.
3. Using formula 8, compute $f(P)$ for paths dual to the paths from the previous step. (Note that dual paths have a minimal evaluation in G between the appropriate nodes as an outcome of formula 8. Also, note that $f(e_1)$, $f(e_n)$, $B(e_1)$ and $B(e_n)$ are constant for all paths between edges e_1 and e_n.)
4. Find the path in G with the minimum evaluation.
5. Add the path to the street network.
6. Remove all street segment candidates that cross the added path.
7. Repeat steps 1 – 6 until either the average number of street segments on a surface unit in the processed fragment is higher than the number from the street network characteristics, or no street candidates are left.
8. Add all remaining street network interfaces to the street network if they were not added by previous steps (this is quite rare because street network interfaces are preferred by the penalty function.)

The described path-adding algorithm presents an issue we must address. It is not guaranteed that the path in G with the minimum evaluation value will not cross itself, which is a situation we want to avoid. This happens quite rarely because self-crossing paths usually have a high evaluation value, but it is still possible. For this reason, we check the selected path for self-crossing. If self-crossing is found, we do not add the street segment that crosses a previously added street segment in step 5 and any subsequent segment of the processed path.

Complexity:

Because of the Poisson disk distribution of the junction candidates, the number of street-segment candidates is linear to the number of junction candidates $O(n)$. The number of edges and vertices of the line graph $L(G)$ is also $O(n)$. The Floyd-Warshall algorithm requires $O(n^3)$ time to compute the evaluation for all possible paths. The path evaluation has to be re-computed in each cycle, so the total time complexity is $O(n^4)$ in the worst case. The average case is faster because we add more than one street-segment candidate with an average path.

Optional post-processing:

When the process of generating street segments is finished, we generate the rest of the city using a combination of existing algorithms. As the first step, we can optionally convert the straight street segments into curved streets. This step is not needed when we work with short average street segments. We do this using the well-known midpoint displacements algorithm based on the measured tortuosity. To guarantee that the curved streets will not intersect, we create a Voronoi diagram for the original street segments and re-generate all displacement operations that are outside the bounds of the corresponding Voronoi cell until the displacement is within the cell.

8. Generating building lots and geometry

When the street network is complete, we create building blocks out of the areas bounded by streets. In the next step, we subdivide them into building lots, using an algorithm described by Parish and Müller [9]. Figure 8 shows different phases of the modeling process. The subdivision algorithm described by Aliaga et al. [14] would also be suitable.

To generate the building geometry from the corresponding lots, we use an approach based on L-systems[27] that was extended by Wonka et al. [8] and later significantly improved by Müller et al. [7]. This approach generates a building geometry using CGA (Computer Generated Architecture) grammar that could be created by a model designer or obtained from existing buildings using a semi-automatic process proposed in [28]. We use an existing grammar that is provided with the CityEngine [29] modeling software. One CGA grammar can generate many building variations.

9. Limitations and potential extension

This section discusses the limitations of our approach and offers suggestions on how they could be overcome. We also discuss how to combine our algorithms with previous related work. Our current implementation does not contain these techniques.

The street generation algorithm presented here does not currently support multiple street types (e.g., highways, main roads and narrow streets), but it can be extended to support them. First, we need to acquire separate street-network characteristics for each type of road. Then, we create street-interfaces for all types of streets. The constraint generation phase is performed multiple times for each street type, from the widest to the narrowest.

Our current implementation also does not take into account space-correlations of individual building types (e.g., industrial buildings are often placed together in existing cities, as are skyscrapers). This can be solved by creating an additional overlapping infinite quarter-type structure (e.g., a grid or a different pseudo-randomly generated Delaunay triangulation) that will provide building-type-probability parameters for the used CGA grammar. The quarter type would be determined by a hash function for each cell of the structure.

Currently, we are working with cities on a flat terrain only. However, our approach can be combined with existing techniques for procedural terrain generation, e.g. [30], to create more realistic infinite cities. The corresponding street-generation algorithm also has to be altered, e.g., to prefer streets on a flat terrain rather than on a bumpy one.

Procedurally generated architectures using our approach can be combined with a finite number of manually designed buildings. We suppose that the designed content is surrounded by a street loop. In such a case, we add all the endpoints of the street segments of the street loop to the pool of junction candidates when we perform the constrained street-generation algorithm of the corresponding Delaunay triangle. In the later phase, we choose the segments of the street loop first, and we do not perform an evaluation for them. We can proceed similarly when the designed content crosses the border of two or more Delaunay triangles.

By substituting the constrained generation algorithm, one can generate different types of worlds than cities, such as mazes, building interiors or electric circuits. Every new type of content will require certain minor modifications.

Moreover, we have identified certain weaknesses of the algorithm that we are unable to overcome at the moment. Our method does not provide good results when the input street-network characteristics have been obtained from an existing map with diverse types of street layouts, e.g., city center and surrounding suburbs. In such cases, the resulting street layout does not reflect either of the original structures. Another problem arises when distances between street crossings are relatively long compared to the tessellation fragments - this can occur particularly in the case of highways. In such cases, the constrained street-generation algorithm does not produce adequate results. We have also noticed performance problems in the case of high-density street layouts with large tessellation fragments due to the $O(n^4)$ complexity of the algorithm. We therefore recommend the use of smaller tessellation fragments for dense street layouts with short streets and greater ones for a sparse layout with long streets. Our method is also limited to simple CGA grammars that can be generated in real time and that provide a reasonably low number of polygons per building (up to 50). The sizes of the generated lots must also reflect the possibilities of the CGA grammar — some grammars can have a limit on the maximum or minimum lot sizes. Fortunately, most grammars can overcome this limitation by generating a simple standby geometry, usually an empty lot or a parking space.

10. Applications

We have found two main scenarios that perfectly take advantage of our approach:

The first scenario uses a distributed 3D world on multiple client devices with sufficient computation power but with limited network throughput. This reflects current mobile phones, tablets and laptops in areas without a high-speed mobile connection or Wi-Fi connectivity. The devices would use a simple server infrastructure to share the hash function and would then be able to generate the static content of the world on their own. They only need to synchronize dynamic changes and positions of users in the world.

The second scenario uses servers in the cloud environment for generating the world and providing the generated content to the connected clients. The stateless generation property allows us to dynamically add more instances of the server on-demand based on the number of clients connected and on the area that needs to be covered. The servers can generate content independently, and they do not have to synchronize their work. Nevertheless, having a common cache for already generated content would be beneficial to them. This scenario is useful for a demanding world-generation process and for clients with limited computational power (e.g., low-end mobile phones, streaming to TV). This can also be used in computer games to prevent players from cheating by preventing the client from accessing data it does not need to display. Content is generated in the cloud and is then sent to the client devices as vector geometry or is streamed as video. In the case of vector geometry, the client device caches content already received to save bandwidth for future queries.

11. Implementation

To verify that our approach is suitable on multiple platforms, we have developed mobile, laptop/tablet, and web browser applications that interactively generate infinite street networks

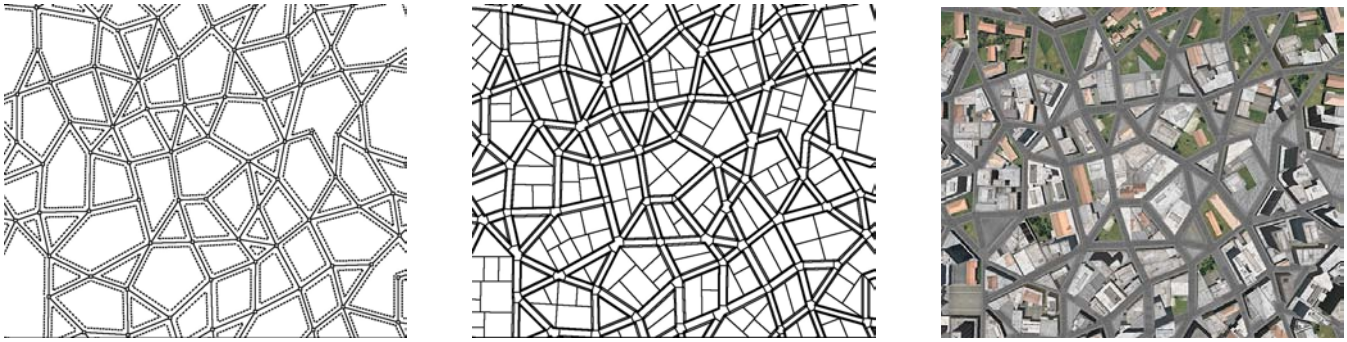

Figure 8: Left: Generated street network from top view; Middle: Added building lots; Right: Added building geometry

according to the generator parameters. Our implementation follows the first scenario from the previous section. We did not implement the second scenario.

The mobile application runs on the Windows Phone 8 operating system using WinPRT technology. The laptop/tablet application uses WinRT technology and can be used on Windows 8 laptops and tablets (e.g., Microsoft Surface), including the limited Windows RT operating system. The web application uses Silverlight 5 technology [31] (.Net equivalent of a Java applet). All our implementations are written in C# and share common parts of the code using C# Portable Libraries. The graphics are rendered using the cross-platform XNA library [32, 33].

We have considered using existing software for geometry generation from CGA grammars to avoid re-implementing them. However, none of the common procedural building modeling tools currently support on-demand generation controlled by 3rd party programs. We therefore created an automatic export module for the CityEngine for on-demand generation using its Jython scripting (Python based on Java Runtime), although plugins are not officially supported by this software. According to CityEngine customer support, future versions should contain official support for plugins. The plugin allows us to generate buildings on demand from their lot shapes. This approach enabled us to generate the city without re-implementing the CGA grammar interpreter. In real scenarios, it is not suitable to include CityEngine in the client application because of its licensing policy. CityEngine cannot be executed on mobile phones, so we run it remotely on a separate server in that case. For the requirements of real future computer games and simulations, we assume that the CGA grammar interpreter will be implemented as a sub-program of the client application. Details about CityEngine plugin development are described in [34].

Because different platforms provide different implementations of their standard pseudo-random generators, we provide our own unified implementation of the pseudo-random generators and the hash function that is based on Donald E. Knuth's subtractive random number generator algorithm [35].

12. Performance and measurements

We measured the performance of the application on a Sony Vaio S15 laptop (Windows 8; processor: Intel Core i7-3612QM Quad-core 2.1 GHz; 8 GB RAM; graphics card: Intel HD 4000; resolution: 1920 × 1080). First, we measured the time until the first frame is rendered. This requires that all buildings in the view frustum be generated beforehand. We compare our method with the previous method published by Greuter et al.[10]. Their approach creates many more streets and fewer buildings per surface unit. To compensate for this , we used different view frustum sizes for each of the methods to achieve roughly the same number of buildings in the view frustum. We performed the measurements for multiple frustum sizes that are in accordance with multiple numbers of generated buildings – see Figure 9.

Our method is slightly slower, as it generates more advanced streets, but both methods run at approximately the same speed.

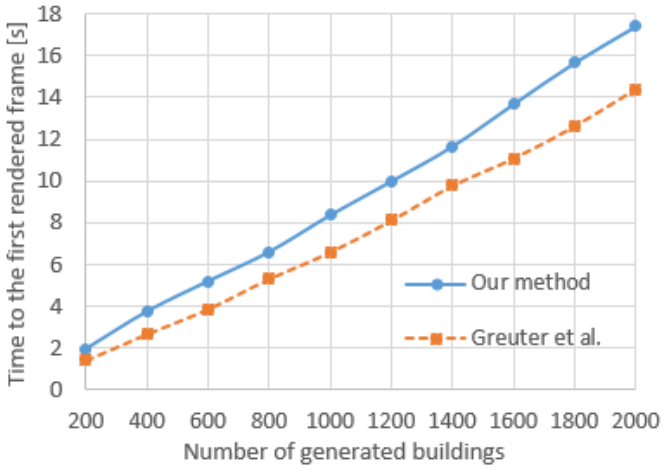

Figure 9: Comparison of the performance of methods for city-layout generation (average from 10 measurements with different seeds). Our method is more demanding.

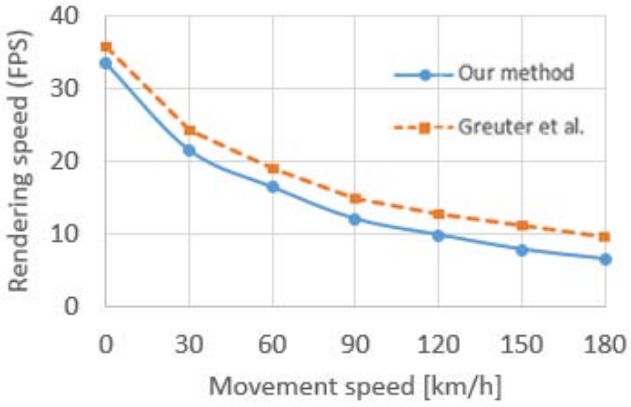

Figure 10: Rendering speed for a moving user; the content is generated on-demand (average from 10 measurements with different seeds). The experiment was performed with a view frustum that contains approximately 1000 simultaneously displayed buildings. The average building in the measurement has 35.4 textured polygons.

Next, we measured the number of frames per second (FPS) for a static camera and a moving camera. The moving view frustum case has to address on-demand generation of the street network and additional buildings. Figure 10 shows how a walking or running user fits into the real-time frame-rate rendering speed. The interactive frame-rate (above 5 FPS) is still maintained for a higher speed.

To prove that our approach can be used on mobile phones, we performed measurements on a Nokia Lumia 920 smartphone (Windows Phone 8, Qualcomm MSM8960 Snapdragon Dual-core 1.5 GHz processor, 1 GB RAM, Adreno 225 graphics card, resolution 1280 × 768). In this case, the mobile phone is used to render a geometry that is generated on a remote computer. The results are shown in Figure 12. The phone is capable of rendering up to 800 buildings in real time. In this test, we do not consider any delay caused by data transfer from a remote computer. These issues are discussed in [37].

Figure 11: Examples of street network varieties - Left: A dense street network based on triangular street shapes; Middle: A medium-dense network based on rectangular street shapes; Right: A low-density street network based on hexagonal shapes; Note that the results do not perfectly reflect the geometrical shapes of the original network.

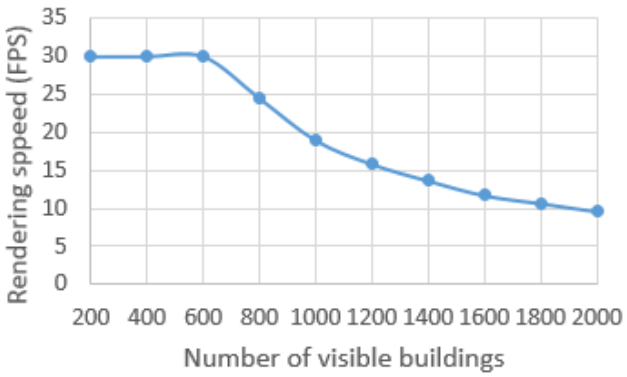

Figure 12: Rendering speed for different numbers of buildings on a Nokia Lumia 920. The phone refresh limit is 30 FPS.

13. Conclusion

We have developed an algorithm for generating a possibly infinite street network on-demand. The main advantage of the algorithm is that it can generate only the content that is visible to the client and that the generated content is consistent in the case of multiple clients. Our appearance of an infinite city looks more natural than that of the previous approach developed by Geuter et al. (compare the results in figures 1 and 2), although it does not perfectly simulate the patterns of the example of existing street network, as the path-selection heuristics do not necessarily result in a perfectly corresponding distribution of segment lengths - see examples in Figure 11. Additional examples of our method are shown in Figure 13. The algorithm can be used in real-time both on personal computers and on mobile phones. We have also defined requirements and general guidelines for stateless generation that can be used for other algorithms.

14. Future work

The use of the algorithm presented here is limited to static virtual-world content. It the system could be enhanced to handle both static and dynamic content, in which case synchronization among clients will be necessary.

Currently, our method does not guarantee an appropriate distribution of street length patterns and angles. The distributions only correspond in mean values. Although the result is already visually attractive, it may be possible to improve the algorithm to simulate the original street patterns more faithfully by reflecting this need more strongly, for example, in the characteristics of the Poisson-disk distribution or in the path selection heuristics.

Thus far, we have not implemented and evaluated the second scenario from section 10. We expect that this will be interesting from the viewpoints of scalability and synchronization.

Other generating algorithms could be adapted to fulfill stateless generation requirements and could be combined with our approach. This would be interesting especially with popular terrain-generating algorithms and engines. Müller et al. [38] have presented an algorithm for synthesizing building façades by example. It would be interesting to combine their approach with ours to automatically generate infinite cities with a structure that matches the maps of existing cities in terms of both building types and street structure.

On-demand real-time generated cities can benefit from techniques such as occlusion detection or level-of-detail of procedurally generated models to increase rendering performance. Previous post-processing tools and techniques must be altered to support models created on-demand. Important progress has been made in a concurrent work by Steinberger et al. [39, 40]. They use GPU for real-time shape-grammar-based generation and rendering of urban landscapes and utilize methods of visibility pruning and adaptive levels of detail to dynamically generate only the geometry needed to render the current view.

Acknowledgements

This research has been partially supported by the Technology Agency of the Czech Republic under research program TE01020415 (V3C - Visual Computing Competence Center) and by Microsoft Czech Republic.

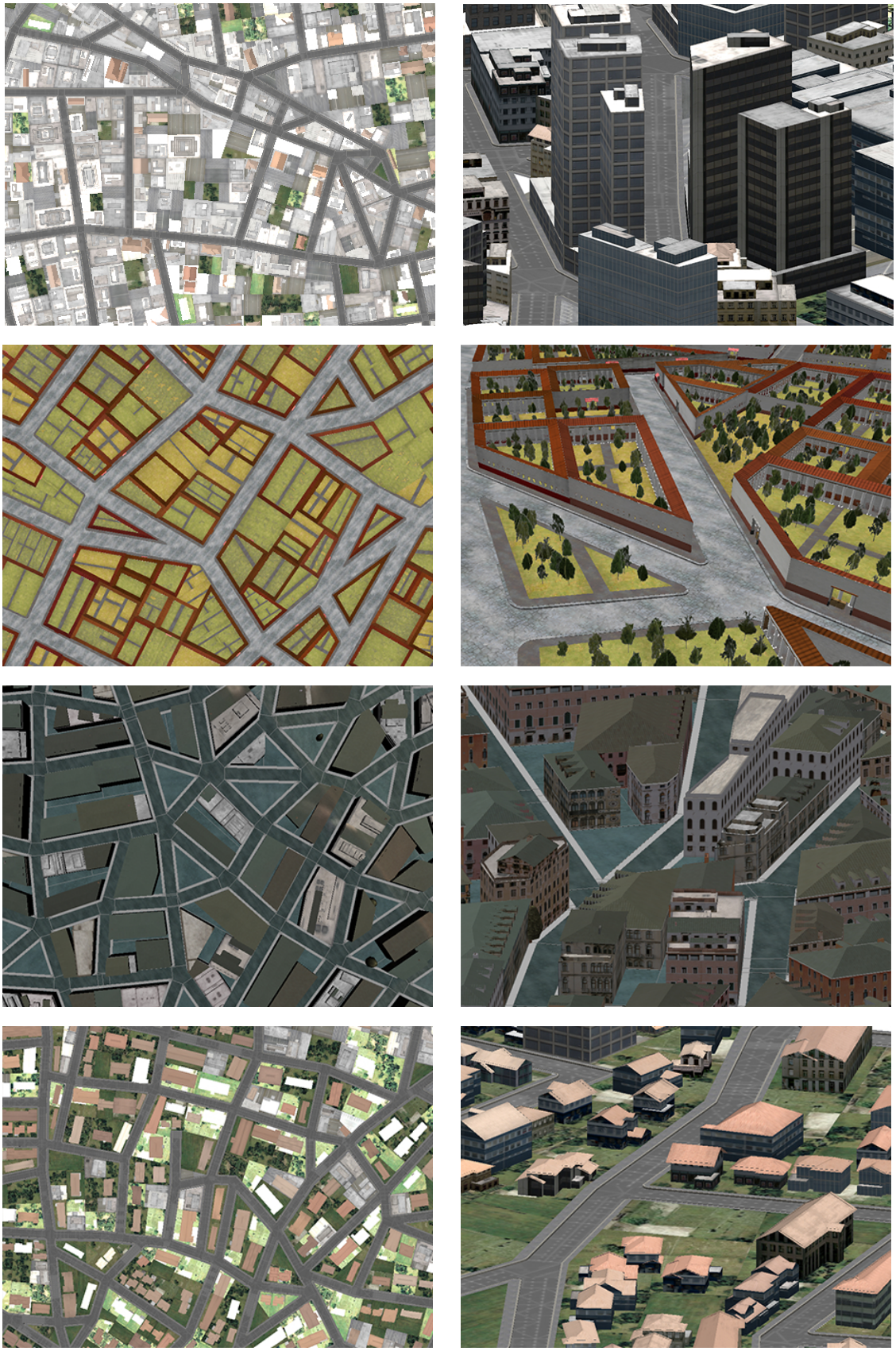

Figure 13: **Examples of our method** Top row: Street network generated from street-network characteristics of Berlin. Detailed view of the generated city; Following rows: Generated cities with alternative CGA grammars - ancient Pompeii [36], Venice and city suburbs

References

[1] Chun B, Maniatis P. Augmented smartphone applications through clone cloud execution. In: Proceedings of the 12th conference on Hot topics in operating systems. USENIX Association; 2009, p. 8–11.

[2] Iosup A, Lăscăteu A, Ţăpuş N. Cameo: Enabling social networks for massively multiplayer online games through continuous analytics and cloud computing. In: Proceedings of the 9th Annual Workshop on Network and Systems Support for Games. IEEE Press; 2010,.

[3] Najaran M, Krasic C. Scaling online games with adaptive interest management in the cloud. In: Proceedings of the 9th Annual Workshop on Network and Systems Support for Games. IEEE Press; 2010,.

[4] Chen K, Huang P, Huang C, Lei C. Game traffic analysis: An MMORPG perspective. In: Proceedings of the international workshop on Network and operating systems support for digital audio and video. ACM; 2005, p. 19–24.

[5] Waldo J. Scaling in games and virtual worlds. Communications of the ACM 2008;51(8):38–44.

[6] Vanegas C, Aliaga D, Wonka P, Müller P, Waddell P, Watson B. Modelling the appearance and behaviour of urban spaces. In: Computer Graphics Forum; vol. 29. Wiley Online Library; 2010, p. 25–42.

[7] Müller P, Wonka P, Haegler S, Ulmer A, Van Gool L. Procedural modeling of buildings; vol. 25. ACM; 2006.

[8] Wonka P, Wimmer M, Sillion F, Ribarsky W. Instant architecture; vol. 22. ACM; 2003.

[9] Parish Y, Müller P. Procedural modeling of cities. In: Proceedings of the 28th annual conference on Computer graphics and interactive techniques. ACM; 2001, p. 301–8.

[10] Greuter S, Parker J, Stewart N, Leach G. Real-time procedural generation of pseudo infinite cities. In: Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia. ACM; 2003, p. 87–95.

[11] Greuter S, Parker J, Stewart N, Leach G. Undiscovered worlds – Towards a framework for real-time procedural world generation. In: Fifth International Digital Arts and Culture Conference, Melbourne, Australia. 2003,.

[12] Cullen B, O'Sullivan C. A caching approach to real-time procedural generation of cities from gis data. Journal of WSCG 2011;19(3):119–26.

[13] Vanegas CA, Kelly T, Weber B, Halatsch J, Aliaga DG, Müller P. Procedural generation of parcels in urban modeling. In: Computer Graphics Forum; vol. 31. Wiley Online Library; 2012, p. 681–90.

[14] Aliaga D, Vanegas C, Beneš B. Interactive example-based urban layout synthesis. In: ACM Transactions on Graphics (TOG); vol. 27. 2008,.

[15] De Berg M, Van Kreveld M, Overmars M, Schwarzkopf OC. Computational geometry. Springer; 2000.

[16] Worley S. A cellular texture basis function. In: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques. ACM; 1996, p. 291–4.

[17] Liang L, Liu C, Xu YQ, Guo B, Shum HY. Real-time texture synthesis by patch-based sampling. ACM Transactions on Graphics (ToG) 2001;20(3):127–50.

[18] Glass KR, Morkel C, Bangay SD. Duplicating road patterns in south african informal settlements using procedural techniques. In: Proceedings of the 4th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa. ACM; 2006, p. 161–9.

[19] Sun J, Yu X, Baciu G, Green M. Template-based generation of road networks for virtual city modeling. In: Proceedings of the ACM symposium on Virtual reality software and technology. ACM; 2002, p. 33–40.

[20] Lefebvre S, Hoppe H. Parallel controllable texture synthesis. ACM Transactions on Graphics (TOG) 2005;24(3):777–86.

[21] Cohen M, Shade J, Hiller S, Deussen O. Wang tiles for image and texture generation. ACM Transactions on Graphics 2003;22(3):287–94.

[22] Merrell P. Example-based model synthesis. In: Proceedings of the 2007 symposium on Interactive 3D graphics and games. ACM; 2007, p. 105–12.

[23] Merrell P, Manocha D. Continuous model synthesis. In: ACM Transactions on Graphics (TOG). 2008,.

[24] Šťava O, Beneš B, Měch R, Aliaga D, Krištof P. Inverse procedural modeling by automatic generation of L-systems. In: Computer Graphics Forum; vol. 29. Wiley Online Library; 2010, p. 665–74.

[25] Cignoni P, Montani C, Scopigno R. Dewall: A fast divide and conquer delaunay triangulation algorithm in E^d. Computer-Aided Design 1998;30(5):333–41.

[26] Leach G. Improving worst-case optimal delaunay triangulation algorithms. In: 4th Canadian Conference on Computational Geometry. Citeseer; 1992, p. 340–6.

[27] Prusinkiewicz P, Lindenmayer A. The algorithmic beauty of plants. Springer; 1991.

[28] Aliaga D, Rosen P, Bekins D. Style grammars for interactive visualization of architecture. Visualization and Computer Graphics, IEEE Transactions on 2007;13(4):786–97.

[29] Esri . CityEngine – 3D modeling software for urban environments. 2008. http://www.esri.com/software/cityengine.

[30] Bevilacqua F, Pozzer CT, d'Ornellas MC. Charack: Tool for real-time generation of pseudo-infinite virtual worlds for 3D games. In: Proceedings of the 2009 VIII Brazilian Symposium on Games and Digital Entertainment. IEEE Computer Society. ISBN 978-0-7695-3963-8; 2009, p. 111–20.

[31] Microsoft . Silverlight. 2007. http://www.silverlight.net/.

[32] Petzold C. Microsoft XNA Framework Edition: Programming Windows Phone 7. Microsoft press; 2010.

[33] McClure WB, Blevins N, Croft IV JJ, et al. Cross Platform Android and iOS Mobile Development. Wrox; 2012.

[34] Sedlacek D, Danihelka J, Lukac M, Berka R, Zara J. Virtual cities in time and space (ViCiTiS). Tech. Rep.; Czech Technical University in Prague, FEE; 2012.

[35] Knuth DE. The art of computer programming 4th edition, volume 2, section 3.2. Addison-Wesley; 2006.

[36] Müller P, Vereenooghe T, Ulmer A, Van Gool L. Automatic reconstruction of Roman housing architecture. Recording, modeling and visualization of cultural heritage 2005;:287–98.

[37] Danihelka J, Kencl L. Collaborative 3D environments over Windows Azure. In: Proceedings of IEEE Seventh International Symposium on Service-Oriented System Engineering and Mobile Cloud. 2013, p. 472–7.

[38] Müller P, Zeng G, Wonka P, Van Gool L. Image-based procedural modeling of facades. ACM Transactions on Graphics 2007;26(3).

[39] Steinberger M, Kenzel M, Kainz B, Mueller J, Wonka P, Schmalstieg D. Parallel generation of architecture on the gpu. In: Computer Graphics Forum; vol. 33. 2014,.

[40] Steinberger M, Kenzel M, Kainz B, Wonka P, Schmalstieg D. On-the-fly generation and rendering of infinite cities on the gpu. In: Computer Graphics Forum; vol. 33. 2014,.