FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Procedural Modelling of Park Layouts

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Visual Computing

eingereicht von

Michael Vasiljevs, BSc.
Matrikelnummer 0727773

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: MSc. Martin Ilčík

Wien, 3 Mai, 2018

Michael Vasiljevs,	Michael Wimmer

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Procedural Modelling of Park Layouts

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Visual Computing

by

Michael Vasiljevs, BSc.
Registration Number 0727773

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: MSc. Martin Ilčík

Vienna, 3rd May, 2018

Michael Vasiljevs, Michael Wimmer

Erklärung zur Verfassung der Arbeit

Michael Vasiljevs, BSc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 3 Mai, 2018

Michael Vasiljevs,

Acknowledgements

I would like to thank Martin Ilčík who was largely responsible for assisting me in the development of the ideas described in this work, encouraging me to work on something unique. Likewise, I would like to thank Michael Wimmer for giving feedback in extremely quick times, making sure this thesis actually happens and showing an example of how to work effectively in the academic setting. I would like to thank Pascal Poublan for letting me use the vegetation models from his SketchUp extension in my ones. I would like to thank the colleagues at Esri and Paul Guerrero for giving tips and encouraging me to finish this work. I would like to thank my parents for the support and telling me to either finish or to quit! And last but not least I would like to thank random cats on the streets, who gave me the comfort when I was low-spirited.

Kurzfassung

Die prozedurale Modellierung in der Computergrafik automatisiert die Generierung von Inhalten. Dabei werden oft manuelle Methoden verwendet, wie bei Anwendungen wie Maya. Grammatikbasierte Methoden ermöglichen die Beschreibung von Objekten auf einer höheren Ebene, die Kodierung von Designentscheidungen in Rule-Dateien und die Erzeugung von unendlichen Variationen durch Ändern der Parameter. Methoden zur Synthese von Landschaften, Straßennetzwerken, Gebäuden und Vegetation wurden beschrieben. Im Kontext der Stadtgeneration kombiniert CityEngine einige dieser Techniken zu einer kommerziellen Lösung, mit der eine gesamte Stadt auf einmal generiert werden kann.
Im Kontext der Park-Synthese ist der Prozess in Layout-Generierung und Platzierung von Objekten unterteilt. Ein Parklayout wird manuell erstellt und in den reservierten Bereich eingefügt oder es wird eine Shape-Grammatik für die Gebäudesynthese verwendet. Im ersten Fall könnte eine Änderung des Designs oder der Bereiche zu erheblichen Modifikationen führen, die für den Benutzer erforderlich sind. Zurzeit ist die Schaffung von Parks und Grünflächen in einer Stadt eher begrenzt und konzentriert sich hauptsächlich auf die Vegetation.
Ziel der Arbeit war die Entwicklung einer Methode für die Parklayout- Synthese. Sie kann, kombiniert mit grundlegenden Platzierungsmethoden, verwendet werden, um Parkmodelle zu erstellen. Basierend auf der Analyse von realen Parks und 3D-Modellen haben wir Muster abgeleitet, die in die Regeln unserer neuartigen Grammatik übersetzt wurden. Insbesondere führten wir eine Regel für die Erzeugung gekrümmter Bereiche ein, die unseres Wissens auf der Ebene in grammatikalischen Methoden noch nicht behandelt wurde. Wir führen auch eine neuartige Möglichkeit ein, eine beliebige Teilmenge der Grenze zu indizieren und auf dieser Grundlage eine zusätzliche Insettierungsoperation bereitzustellen. In unserer Arbeit haben wir den Kontext von CityEngine als möglichen Anwendungsfall betrachtet.

Abstract

Procedural Modelling in Computer Graphics automates content generation, where commonly manual methods have been employed, as in using modelling applications like Maya. Grammar-based methods allow to describe creation of objects at a higher level, encoding design decisions in rule files and enabling generation of infinite variations by just altering the parameters. Methods for the synthesis of landscapes, street networks, buildings, and vegetation have been described. In the context of the city generation, CityEngine combines some such techniques into a commercial solution that can be used to generate the whole city at once.

In the context of park synthesis, the process is divided into layout generation and placement of objects in it. Typically, a park layout is either created manually and inserted into the reserved area, or a shape grammar designed for building synthesis is employed. In the first case, a change to the design or the surrounding regions could result in considerable modifications required of the user. At the present moment, generation of parks and green spaces in a city is rather limited and mainly focused on vegetation placement.

The aim of our work was to design a method for park layout synthesis, which when combined with basic placement methods could be used to create believable park models. Based on the observation of real-life parks and 3D models of parks, we have derived a number of patterns, which have been translated into the rules of our novel shape grammar. In particular, we introduce a rule for creating curved regions, which, to our knowledge, has not been addressed yet at this level in grammar-based methods. We also introduce a novel way to index arbitrary subset of the boundary and provide an additional insetting operation based on that. In our work we have considered the context of CityEngine as a possible use case.

Contents

Kurzfassung **ix**

Abstract **xi**

Contents **xiii**

List of Figures **xv**

List of Tables **xviii**

List of Algorithms **xix**

1 Introduction **1**
1.1 Motivation 1
1.2 System Overview 3

2 Analysis of Existing Parks **5**
2.1 About Parks 5
2.2 Parks in CityEngine 5
2.3 Park Layouts 7
2.4 Distribution of Objects 10

3 Related Work **11**
3.1 Methods for Model Synthesis 11
3.2 Grammars 12
3.3 Layout Generation 14
3.4 Object Placement 15

4 Layout Generation **17**
4.1 Definitions 17
4.2 Grammar for Park Synthesis 18
4.3 Grid Rule 18
4.4 Cells Rule 19
4.5 Rays Rule 23

4.6 Boundary Insetting, Peel Rule . . . 24
4.7 Placement Rules . . . 27
4.8 Push Extrusion . . . 29

5 Indexing and Selection **31**
5.1 Indexing . . . 31
5.2 Region Fitting . . . 38
5.3 Selection . . . 40

6 Placement **45**
6.1 Scatter Rule . . . 46

7 Implementation **49**
7.1 Layout Generation . . . 49
7.2 SketchUp Integration . . . 54

8 Results **57**
8.1 Layouts . . . 57
8.2 Performance . . . 62
8.3 Comparison to Other Work . . . 63

9 Conclusion **65**
9.1 Summary . . . 65
9.2 The Focus of the Tasks . . . 66
9.3 Future Work . . . 67

A Grammar Reference **69**
A.1 Parameter Types . . . 69
A.2 Partitioning Rules . . . 69
A.3 Layout Boundary-based Placement Rules . . . 74
A.4 Object Placement Rules . . . 76
A.5 Re-writing Rules . . . 77
A.6 Selection . . . 78
A.7 Selectors . . . 79

B Examples **83**
B.1 Grid Park Variant 1 . . . 83
B.2 Grid Park Variant 2 . . . 83
B.3 Cells Park Variant 1 . . . 84
B.4 Cells Park Variant 2 . . . 85
B.5 Rays Park . . . 86

List of Figures

1.1 Park Generation Schematics. The input polygon (left) is first divided by paths into zones in the layout generation step (middle). Then, vegetation and various park objects are placed in the object placement step (right). 3

2.1 CityEngine CGA region partitioning rules: (a) `split` along x axis and (b) `split` along x and then z axes, (c)`splitArea` along x axis, (d)`shapeL`, (e)`shapeU`, (f)`shapeO`, (g) `setback` and (h)`offset`. 6
2.2 CityEngine parks created using the CGA rules (courtesy of Esri). 7
2.3 CityEngine Garden Scene. 7
2.4 Left: gardens of the Château de Villandry. Right: parks from the game SimCity (2013) (courtesy of Steve Manx). 8
2.5 Grid-like parks 8
2.6 Rays and Grid structural patterns found in rather complex real-life parks like Schoenbrunn Gardens layout plan (left) and sattelite image (right) (courtesy of Google maps). 9
2.7 Examples of Ray-like park layouts with a single primary centre. 9
2.8 Free-form parks have curves instead of straight edges as boundaries. 10
2.9 Trees are often placed into small regions of a park, which could be round (left) or square (right) and are usually covered with mulch, courtesy of Buzzle.com and WHE. 10

3.1 Procedural content creation process flowchart. 12
3.2 Left: an example of L-Systems rules and derivation. Given two *rules* and an *axiom* shape, derivation allows generation of infinite number of sencentes in this manner. Right: a tree generated with an L-System (courtesy of Ted Wong, Bryn Mawr College). 13
3.3 Polygon Line split using two target points, Ulmer [Ulm05]. 14
3.4 Street graph generating methods could be applied to Park Modelling, Aliaga et al. [AVB08] 15
3.5 Park example using Edit Propagation, left: the initial layout, right: the result; Guerrero et al. [GJWW14]. 16
3.6 Plant Ecosystems, Deussen et al. [DHL+98]. 16

4.1 Grid path construction uses vertical (left) and horizontal (middle) path regions, which when combined, form a grid (right). 18
4.2 Left: desired Grid quarter ordering shown in blue, whereas the actual order of regions that result from geometric operations can be different (for example, in red). Right: even when a considerable chunk of a quarter is clipped away, there is a high likelihood that the point (green) at the centre of the unclipped version of the quarter is also found within the clipped quarter. 19
4.3 Free-form layouts. Left: a park between blocks of flats in Manhattan. Right: cells layout created by smoothing out Voronoi cells. 20
4.4 Early approaches for creating park layouts with a single junction. (a) Intersection of paths spawned by random positions along the pairs of opposite sides with a circular junction. (b) Based on Quarters created by Ellipses placed at the corners. (c) Based on Quarters created by Bezier curves. (d) Similar to (c), with a random permutation of control points. 20
4.5 Junctions in the Cells rule. Left: correctly created junction (the purple shape) from the three vertices that resulted from shifting three neighbouring cells. Middle: when two cell neighbour vertices are close enough to each other, they may merge, collapsing the section; right: correct placement of junctions after the section collapse. 21
4.6 Left: clipping before smoothing of boundary cells results in blobs; right: desired outcome. 22
4.7 kNN-based junction connection – intersection is visible. 24
4.8 Rays path thinout: (a) the dense triangulation-based paths without thinout (ratio 1), compare to (d) – the spanning tree of (a), i.e. a completely thinned out path network (ratio 0); (b) a thinout with ratio 0.5 and (c) 0.2. 25
4.9 Special treatment of the path geometry in the Rays rule. Left: we cannot simply create a union of paths sections and then subtract the junctions to get the clipped regions since often occurring section overlaps result in merges of a number of such input regions into the more complex polygons (dark blue), as well as some holes (green). Right: Sizes of a junction (blue region) and sections (green regions). Given a path width (d_s) and an angle between path sections (α), the junction diameter (d_j) should be larger than that of the pink region, $d_j > \frac{d_s/2}{sin(\alpha/2)}$. 26
4.10 Peel rule intruded shape extensions. Left: extensions at corners reaching two of the adjacent edges. Right: an extension between two intruded shapes that share a vertex with an angle greather than 180°. 26
4.11 Quarter hedges of a grid-like park layout appear to be 'inset' at a certain distance from the boundary. 27
4.12 Left: boundary selection in bright red on the initial region, for example a grid quarter. Middle: Peel rule generates light red region. Bright green selection using a special labelling (Sec. 5.1.2), that takes into the consideration the previous production. Right: Peel applied second time, resulting in a light green region region that is isolated from the boundary.) 27

4.13 Left: Insert rule with three new shape instances. Inserted shapes are clipped to the iregion subsequently (not shown). Right: Place rule with three new shape instances. 28

5.1 Left: default indexing follows the actual order of edges in the geometry data. Middle: a quad shape that has physical representation of 5-sided polygon Right: logical representation of the shape in the middle, a quad. 31
5.2 Inherent amplification of numeric indexing can be demonstrated on the Peel rule, where resulting number of indices will be $2n + 2$. 33
5.3 01-Indexing logical edge labelling on example application of the (left to right) Peel, Grid and Cells rules. 34
5.4 Alternative 01-Indexing mode on the example of the Peel rule. Compare to the left blue-shaded region on the Figure 5.3 left. 34
5.5 Example of the 01-indexing demonstrated by first applying the Grid rule (left), and then the Peel rule (right) on the **0**-subset of the boundary of each quarter. 34
5.6 Indexing continuity is broken by the inserted shape. Left: geometry surrounded by one index gets the same index. Middle: how should indexing be propagated when from both sides differ? Right: extending the index in the direction of orientation. 36
5.7 Example case for reordering of indices. Left: physical geometry ordering in iregion. Middle: geometry ordering after subtraction of, which could be a vertical path shape in a Grid production, at the middle of the iregion. In the bottom region incorrect orientation can be observed. Right: correct geometry ordering after reordering of indices. 36
5.8 Directions, represented by points, are assigned an index within the range [0..3] 37
5.9 Left: naïve Shrinking does not handle edge events. Middle: correct reindexing of shrunk polygon. Right: correct indices after multiple edge collapses at the indexing boundary. 38
5.10 Recursive Grid application, left: without hint shapes, may result in non-quad intermediate quarter shapes; right: works as expected with hint shapes . . . 39
5.11 The *unclipped quarter* in the example of a Grid production is used as the hint shape. 40
5.12 Numerical selection of LE-labelled polygons on the range [0.5..1.5]. 41
5.13 Edge orientation selection. Left: the set of initially selected edges (red) is often discontinuous. Right: a continuous range (red polyline) is computed by looking for the largest continuous *un-selected* range (amber polyline) and then subtracting from the boundary set. 42

6.1 Plants cover the ground region completely [DHL+98]. 45
6.2 Experimentation with object placement using non-procedural methods. The park layout is inspired by the SimCity Large Park (see Figure 2.4 right). . . 46
6.3 Process of sample generation using the Scatter rule. 47

7.1 Polygon creation from half-open cells. Left: half-open edge construction. Right: terminating edge (red) created between the vertices created by clipping of the two half-open edges of the same cell. 52
7.2 Polygon placement. Left: B creates a hole within A, middle: A is cut from the insertion point to the boundary, right: A is cut in half into $A1$ and $A2$ and B is inserted along the cut. 53
7.3 A derived shape may split the iregion which could be: left: An Intruded shape, right: or an Inserted shape. 53
7.4 Polygon insetting. Left: individually intruded edges (orange) connected junction polygons (brown). Right: ϵ-extension of a junction polygon. 54

8.1 Grid Layout variant 1 (Appendix B.1) – comparison of the generated park with the photograph of the real-life park that has inspired it. 57
8.2 Various axiom region sizes applied to Grid Park variant 1. Left: the default size is depicted on the Figure 8.1 left, compare to a large very region (middle) and a very small (right) region. 58
8.3 Grid Park variant 1: distortion applied to the quad axiom region. 58
8.4 Grid Park variant 1: results with two variants of more complex axiom regions. 58
8.5 Fitting methods applied on the Grid Park example1: (a) Axis Aligned Bounding Box (AABB), (b) Axis Aligned Corner fitting, (c) Oriented Bounding Box (OBB), (d) Oriented Corners fitting, (e) Angle Thresholding. 59
8.6 Grid Layout variant 2 (Appendix B.2), the model and the layout of the real-life park that has inspired it. 60
8.7 Shape Fitting plugin results: OBB (left) and Angle Thresholding (right). . . 60
8.8 Cells results. Top left: variant 1 (Appendix B.3), ithin a smoothed sub-region . Top right and bottom: variant 2 (Appendix B.4), the use of symmetry in cell samples in regions of various sizes. 61
8.9 Rays results (Appendix B.5), the side view (left) and the top view showing the path structure (right). 62

List of Tables

8.1 Timing of park generation using four different rules given in miliseconds. Rulefile for Grid park variant 1 was tested with (the second row) and without grass samples. Two square testing regions were used, one sized 500 (small) and another 2000 (large) units. 63

List of Algorithms

5.1 Calculation of the (logical) edge index based on the (logical) edge position within the region. As a reference, Figure 5.8 shows index for each direction of an edge that is represented by a point. 37
5.2 Combined shrinking and re-indexing. 38

CHAPTER 1

Introduction

This work deals with modelling within the realm of Computer Graphics (CG). CG is becoming increasingly utilised in various fields including, but not limited to entertainment, architecture, urban planning and industrial product design.

The technical aspect of CG image synthesis roughly involves two processes – modelling and rendering. Modelling deals with creation and placement of objects within a 3D scene, whereas rendering involves generation of images that are perceptible to the human visual system. It is entirely modelling that we would like to explore.

There are a number of ways to create, represent and store the objects for a 3D virtual world. We consider two of them – inputting the geometrical elements manually or using a program as a *tool*, i.e. writing a program to generate the geometry for you. The second part opens various possibilities for the automatic generation of various kinds of models on the computer. It is also called *procedural modelling* and is the area of interest of this thesis.

A common way to handle a 3D object in space is a *closed manifold surface* model, while other representations include *voxel* data and a *point cloud*. Moreover, we can also describe a surface model using polynomial equations or mathematical functions. However, the most standard way is to use a list of polygons that can be easily converted to triangles for rendering. We will remain with the polygonal representation of models, in particular, simple polygons with an arbitrary number of vertices, however the procedural system can naturally be extended to handle instancing of objects based on any higher level format that would make use of the modern hardware. However, not to lose the focus, we will leave this out of scope of this work.

1.1 Motivation

Some of the first models were created by manually entering the vertices based on measuring real-life model objects, for example, reproduction of the human hand at the

University of Utah in 1972 [Sit13]. Early CG sequences feature models that were often created from a combination of primitives. In fact, before modelling applications became commercially available, content creation was performed by programmers themselves, so model generation would have been performed with statements of a general-purpose programming language.

Development of increasingly affordable and capable graphics hardware enables modelling applications to provide interactive visualisation while performing complex operations in real time. Such a process allows for better productivity, a greater level of control and ability to manipulate larger amount of content at the same time. However, arguably, this also delays the adoption of procedural modelling pipelines – it is preferable, also for economic reasons, to improve production iteratively rather than switching to a radically new processes.

In the current generation of computer games, a player can navigate an entire city in one continuous gameplay session. The amount of content in an AAA title has risen considerably within the last two decades, and respectively, the teams responsible for producing it have increased. On the other hand, current techniques already allow generation of entire cities [Cit15] procedurally, so procedural modelling is expected to be employed to replace the laborious part of creating – in this case the set models for buildings and other objects – by hand.

The major advantage of procedural modelling is that once a design is specified, for example in grammar, it is possible to create a number of variations of it automatically. It is feasible to see an increasing rate of adoption of procedural content generation within the next decade.

1.1.1 CityEngine Framework

The framework of procedural city-generation in CityEngine [Cit15] has been the primary motivation for this thesis. It was the first, and still currently is the only solution of the kind that is capable of generating a whole city model from the ground up using entirely procedural techniques. As in a real city, modelling in CityEngine starts by partitioning a given city area by a street network. First, major roads and highways are placed along the landscape, partitioning it into polygonal faces called *blocks*. Then, blocks are split into smaller regions by local streets – *quarters*. Quarters are further subdivided into *lots*, where individual houses are placed. Building synthesis is performed using the CGA (CG architecture) grammar, provided by the user in the textual form – similar to a script. CGA is the core strength of CityEngine, and it can be used to produce other kinds of objects, including parks. Our interest in CGA extends only to the application of the grammar for generation of parks, and we will consider that in greater detail in Section 2.2. The framework within CityEngine, however, is also of interest since it can supply axiom shapes which become the starting polygons in our system.

Synthesis of buildings, streets, plants has been explored, but park generation has largely been unexplored in the literature, yet parks play an integral role of a modern city, and we wish to fill this gap by developing a method for procedural park synthesis.

1.2 System Overview

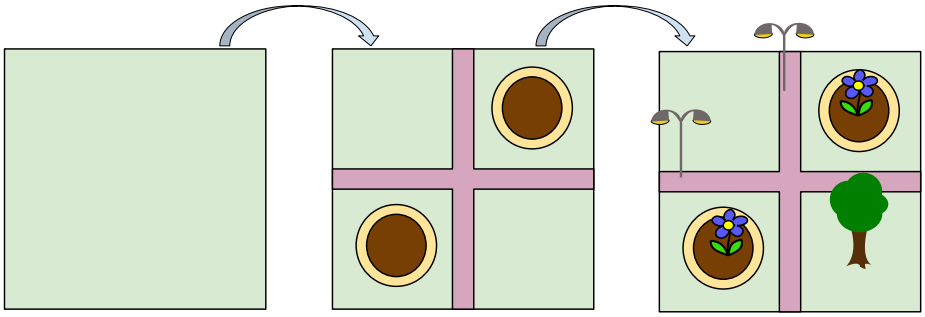


Figure 1.1: Park Generation Schematics. The input polygon (left) is first divided by paths into zones in the layout generation step (middle). Then, vegetation and various park objects are placed in the object placement step (right).

Following the idea of CityEngine of recursively partitioning a given area into the smaller parts, where eventually houses are placed, our concept of park generation, likewise, involves subdividing an input region into smaller regions. These 'subregions' correspond to particular park areas – paths, lawn patches, flower beds – where objects, like trees, plants, flower pots, are subsequently placed (see Figure 1.1). Hence, the park generator is divided into two major components working in succession: **layout generation** and **object placement**. The park generator was implemented as a SketchUp extension. The front-end, or the presentation layer of the extension, utilises SketchUp GUI facilities to allow the user to capture the input and to forward it to the standalone layout generator library. The library returns the subdivided regions of the layout and object positions. To allow a believable visualisation of the layout, regions are assigned colour and height. Object positions are used to instantiate externally created 3D models. The result is interactively rendered in the SketchUp viewport.

Layout generation is the core functionality of the system and is based on a CGA-inspired ruleset that targets park-specific partitioning. In addition to encoding observed park patterns into the rules, we introduce a novel way of partitioning based on indexing and selection of the park boundary.

CHAPTER 2

Analysis of Existing Parks

The majority of real-life park designs were observed to follow regular patterns, which include right angles, rectangular or triangular boundaries, circles – in both paths networks and also partitioning inside the connected non-walkable green areas we call *quarters*. Quarters are sometimes supplemented by more elaborate patterns (see Figure 2.4). Sometimes, especially in modern parks, designs are based on curved forms of roughly the same shape (see Figure 2.8).

Although, ideally, we want to generate parks that look like actual real-life parks, our secondary goal is to create parks similar to what was observed in computer games. We also try to follow the steps of forming an artificial park as close as possible (for instance, see Figure 2.3).

2.1 About Parks

The design, planting and maintenance of complex park structures like carpet parks are complex enough tasks in horticulture. Parks became public at the end of 18th century, and the time before then they were mostly used for aristocratic recreation. As a result, most of the design was following the tastes of the upper class [FW00]. It is interesting to note that some parks were often open to the public to improve the development of the lowest social classes.

2.2 Parks in CityEngine

A simple park could be created with CGA. Using CityEngine rules like `split`, `setback`, `offset`, `shape*` and `splitArea`, rectangular partitioning of the lot shape could be achieved (see Figure 2.1). On top of that, objects could be placed using the `scatter` rule to create a city park (see Figure 2.2).

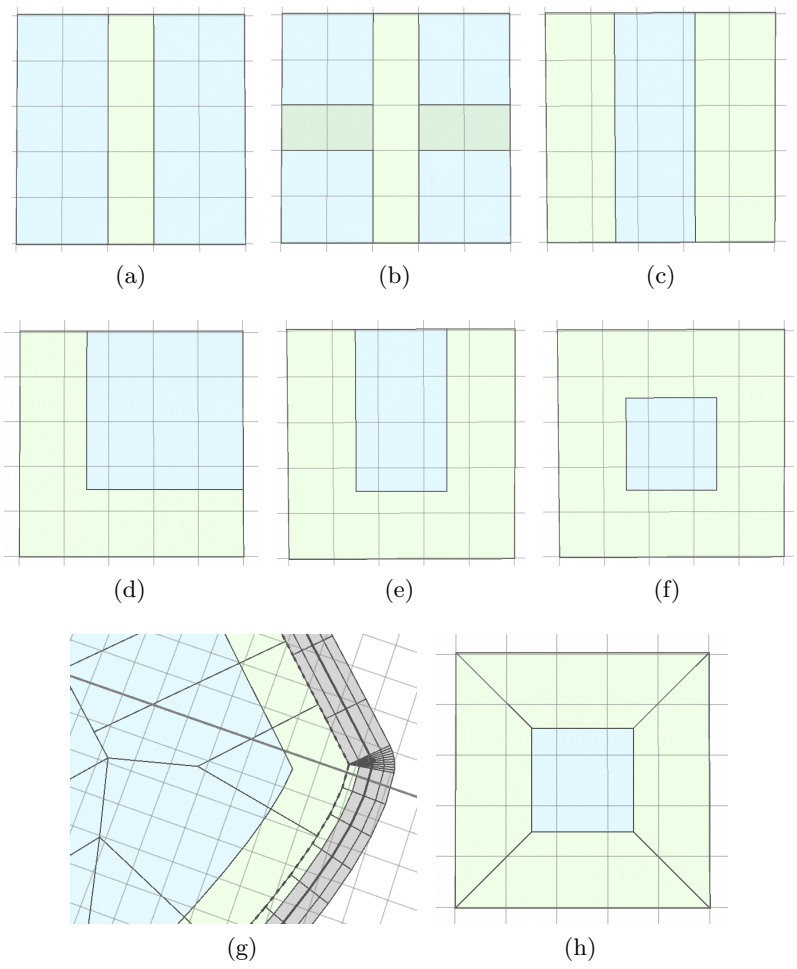


Figure 2.1: CityEngine CGA region partitioning rules: (a) `split` along x axis and (b) `split` along x and then z axes, (c)`splitArea` along x axis, (d)`shapeL`, (e)`shapeU`, (f)`shapeO`, (g) `setback` and (h)`offset`.

Figure 2.2: CityEngine parks created using the CGA rules (courtesy of Esri).

Figure 2.3: CityEngine Garden Scene.

Importing a manually generated layout, namely such created with a CAD or a Vector-drawing application, allows a CityEngine user to create a more interesting park, where the rules are used only for the object placement (see Figure 2.3).

2.3 Park Layouts

The overall schematic look of a park is determined by separation of park areas into paths and quarters, which we call a *structural pattern*. In most of the parks that we have observed, one of the three structural patterns has been noticed to either entirely shape the layout or take the dominant part of the design. The patterns were named as *Grid*, *Rays* and *Cells*, which are also eponymous to the names of the rules that implement them (description of which follows in Chapter 4).

2.3.1 Grid Pattern

A large number of park layouts, especially those developed in France following the Renaissance period, conform to, or, otherwise incorporate the Grid pattern into their designs (see Figure 2.4 left).

Figure 2.4: Left: gardens of the Château de Villandry. Right: parks from the game SimCity (2013) (courtesy of Steve Manx).

Figure 2.5: Grid-like parks

The foundation of the pattern is grid-like intersections of the two sets of paths, vertical an horizontal, placed at regular intervals next to each other. Paths split the park area into roughly rectangular quarter regions. The Grid pattern is applied to a quad region, ideally having right angles. However, more complex regions could also be accepted when using *shape fitting* (Sec. 5.2).

Grid-like park models are found in digital media, for instance, in city-building games (see Figure 2.4 right, ignoring the path curvature). Grid parks can be nested, or used as containers for smaller parks (see Figure 2.5).

2.3.2 Ray-like Partitioned Layouts

In addition to paths intersecting at right angles, a lot of the traditional parks can be observed to have paths arranged diagonally, often connecting the centre junction and the boundary corners (see Figure 2.7, Figure 2.6). To describe such designs, one can imagine paths as 'rays' being 'cast' from a junction towards a set of points on the boundary. Hence we describe such arrangement as the Rays pattern, although at the technical level

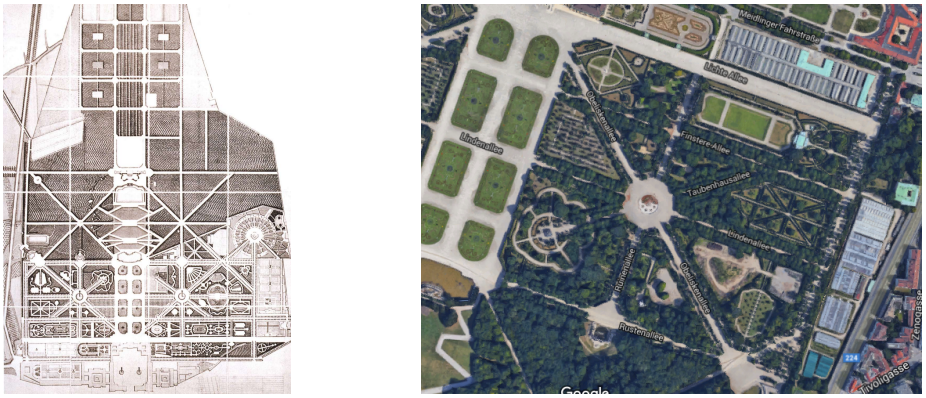


Figure 2.6: Rays and Grid structural patterns found in rather complex real-life parks like Schoenbrunn Gardens layout plan (left) and sattelite image (right) (courtesy of Google maps).

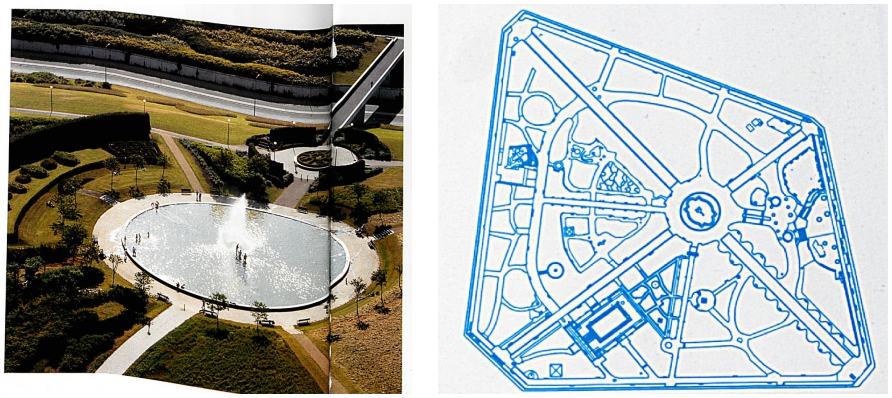


Figure 2.7: Examples of Ray-like park layouts with a single primary centre.

the rays are 'cast' the other way round. A single junction Grid layout can be represented with Rays where paths are cast at the right angles. Rays-based layouts very often have a circular centre, as can be seen in the Figure 2.7.

2.3.3 Curved Regions Layouts

Grid and Rays can model a large number of simple parks where normally, other then at junctions, straight lines dominate. However, plenty of parks, especially those of the more recent 'modern' designs, contain curved regions. Quarters, instead of having sharp angles, appear to be more blob-like (see Figure 2.8). The layout of the CityEngine garden scene example (see Figure 2.3), although manually created, is an example of this pattern.

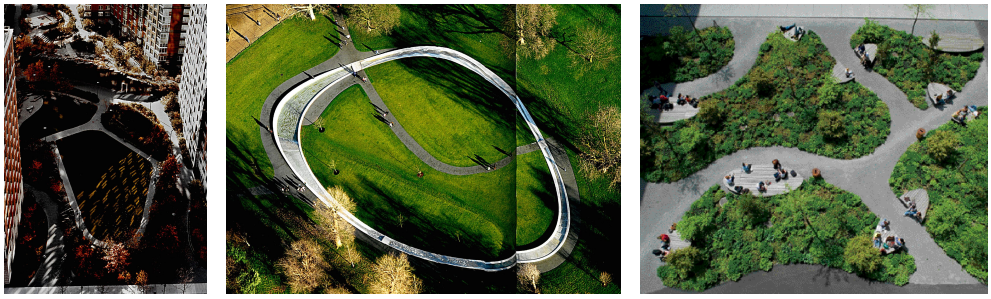


Figure 2.8: Free-form parks have curves instead of straight edges as boundaries.

Figure 2.9: Trees are often placed into small regions of a park, which could be round (left) or square (right) and are usually covered with mulch, courtesy of Buzzle.com and WHE.

2.4 Distribution of Objects

The look of the park is shaped by vegetation as well as some artificial objects like benches, fences, lampposts, growing pots and so on. Most often the plants are evenly distributed into the given region, whereas the desired aesthetic effect is achieved by sculpting and arranging the container regions, as is demonstrated in the garden scene in Figure 2.3. A single plant instance of a tree or bush may be observed at the centre of a small region, which is often a square or a circle, and a number of such regions could also be aligned along a line (see Figure 2.9). As a result, we want to put more emphasis into the placement of regions as a part of the partitioning process, as opposed to scattering, which had been the predominant instrument in the previous work. When extending our observation to the modelling of the artificial parks, we have noticed that in the garden example (Figure 2.3, the layout, however, is not visible), some regions are split into two, and a small tree region is inserted in the middle.

CHAPTER 3

Related Work

The most significant mention of grammars is the use of split grammars for building modelling [MWH+06] or the use of L-systems for road synthesis [PM01]. In this chapter, we describe how we can apply grammar methods, in particular, shape grammars into the context of park layout synthesis, supplemented by the means for scattering objects.

3.1 Methods for Model Synthesis

> "Only the automatic rule derivation enables large-scale procedural modelling." [VAW+10]

A **classical approach** to modelling involves manual manipulation of geometric elements, vertices and edges of a polygon or control points of a polynomial surface, using GUI. The 3D modelling applications, like Autodesk Maya, have been improving the work efficiency by enhancing the functionality for selection, manoeuvering and duplication of such elements.

To avoid repetitions or to reuse collections of the meaningful model parts, known as 'prefabs', the modelling tasks can be automated by using a general-purpose **imperative programming language** – for instance, Python scripting in Maya. The set of manipulated geometric elements are treated as the program state that is sequentially altered with geometric functions. A higher level system like GML offer more abstraction and emulate state with function sequences, but remain Turing complete [Hav05]. Leblanc at al. [LHP11] describe a way to generate the whole buildings including the facades, interiors, fixtures utilising Constructive Solid Geometry (CSG), storing the state in a *component tree*.

To simplify the process, instead of writing steps to generate a shape, we can create a *description* of the scene. **Formal grammars**, as exemplified by L-Systems and the CGA grammar, help us to achieve that. The description is encoded into a set of rules, each rule describing how to transform one element into another. Application of a rule is

comparable to a refinement step in performing modelling traditionally. We will describe grammars shortly in Section 3.2 in greater detail.

Rather than specifying design in grammar, more complex models could algorithmically be constructed by first analysing meaningful patterns from a group of simpler models – to retain the similar appearance while maintaining the continuity. This process is known as **example-based** synthesis, was described by Merrell [MSK10] and is roughly divided into two stages, analysis and synthesis. Such approach allows facilitating synthesis without the prior programming experience. An accomplished designer then can focus on, for example, modelling using the classical technique. Other methods, such as iWires [GSMCO09], use the analysis step but allow the user interactively alter the model. On the other hand Lipp et al. [LWW08] explored editing grammars visually by identifying *instance locators*. Synthesis of grammars based on sketching has also been explored [NGDGA+16].

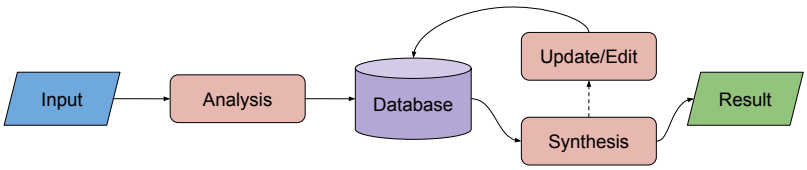


Figure 3.1: Procedural content creation process flowchart.

Considering all these methodologies, we can classify procedural modelling methods according to the diagram on the Figure 3.1, which contains analysis, synthesis and edit steps, and where intermediate results are stored in some form of a database. The primary focus of this thesis is on *synthesis* using a novel and a rather simple shape grammar targeting generation of park layouts, where the database is provided by a user in the form of a textual rule file. In this report, we omit the analysis and the editing steps, although region fitting (see Section 5.2), can be considered as a form of analysis. The editing part mostly remains detached from procedural generation and can either be performed manually on the input and output geometry, or considered a task for the future work.

3.2 Grammars

We have mentioned that the Grammar-based model synthesis is based on a description of a scene with rules. An application of a rule is called a *production* and the process consisting of multiple productions is called a *derivation*. A sequence of productions is usually required to model an object. A rule describes how to perform a transformation on an element. The elements can be *string symbols*, *shapes in space*, or *nodes of a graph*. The derivation process terminates when no further productions are possible or, in some cases like for example L-Systems, derivation needs to be explicitly terminated after a given number of productions. The resulting elements are then translated into geometry. Variations between two instances derived using the same rule sequences, when for example,

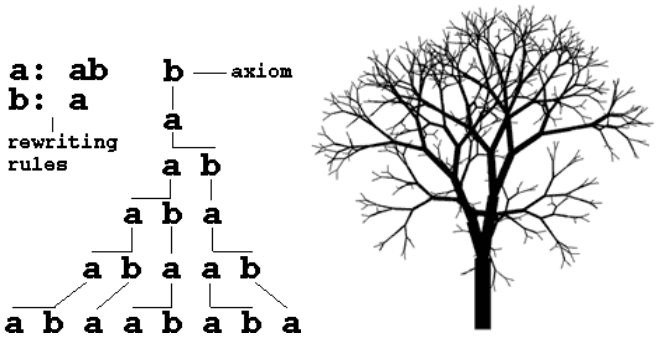


Figure 3.2: Left: an example of L-Systems rules and derivation. Given two *rules* and an *axiom* shape, derivation allows generation of infinite number of sencentes in this manner. Right: a tree generated with an L-System (courtesy of Ted Wong, Bryn Mawr College).

alternatives of the same design are demanded, can be achieved with parametrisation of rules by *random variables*. Such productions are called *stochastic*.

Procedural modelling by means of grammars was introduced by *L-Systems*, which operate by replacing symbols in a string, Figure 3.2. Starting with the **axiom**, the initial set of symbols (b in the Figure 3.2 left), we keep replacing all of the symbols in the string according the matching rule. Each rule contains a matching symbol and a list of symbols that replace this symbol (see Figure 3.2 left, in the first rule symbol a gets replaced into ab). All available symbols are listed in an **alphabet**. And at the end, we are left with the symbols that can be either converted to geometry or to geometric operations, which are called the **terminal** symbols.

Shape grammars operate on geometric shapes instead of symbols. The production context of a shape cabe be important, which is then *context sensitive*. Hence, unlike L-systems, parallel derivation often cannot occur. In their work Wonka et al. [WWSR03] derived a two-grammar system to building models, one of which is used for the production of symbols, and the other for replacement of these symbols with design-guided geometry. A straightforward and effective system has been designed by Müller and colleagues [MWH$^+$06], which has seen been successfully employed in the industry in the CityEngine [Cit15] application, where designs are encoded into the grammar. The CGA was enhanced further by a number of papers, for example, Krecklau et al. [KPK10] introduced better design encapsulation, and described custom geometric operations at terminal symbols, for instance, curved surfaces. Ilcik et al. [IMAW15] described mixing rules using layers within the context of facades. Schwarz and Müller [SM15] introduce CGA++, where shapes can be accessed at any step of the derivation process, which is also enchanced with the *events*. Besides modelling, grammars can be applied to other fields like animation [IFPW10].

Representing rules with nodes allows for abstraction into a *graph* form. Such structure allows better visualisation of the rule data, and also manipulation with graph-based algorithms. Such representation is more intuitive and is also accessible to a broader user group, without requiring the more technical knowledge of scripting. Patow [Pat12] applies

graph algorithms for verification, and graph rewriting for optimisation, for example, to remove repetitions. For further abstraction, sub-graphs components can be abstracted into the operator nodes. Silva et al. [SMBC13] extend the work of Patow, introducing cycles into the graphs, which increases node density. Furthermore, *semantic components* are introduced, which are functional components relating to, for instance, the architectural objects.

3.3 Layout Generation

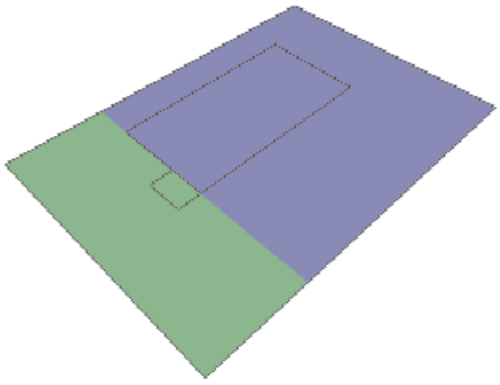


Figure 3.3: Polygon Line split using two target points, Ulmer [Ulm05].

Ulmer [Ulm05] facilitates simple partitioning of a target polygon based on a line split and rectangle insertion along the split line (see Figure 3.3). CityEngine utilises partitioning at three different levels: at the global level by the street network, splitting of a block into parcels, and by using the CGA. Since a rough park layout consists of a network of paths that partitions a region, where quarters are reduced to lawn areas, the initial work on CityEngine by Parish and Müller [PM01] on the creation of street graphs could be applied to the park modelling. This process is based on an *extended L-System* and could be applied to generate certain classes of parks where narrow road-like paths predominate. Moreover, the *globalGoals* function could direct generation to target a particular path layout pattern. Methods splitting a block into parcels, for instance, a recursive subdivision of the largest near-parallel edges, as also described in the original CityEngine paper [PM01], was extended by Vanegas et al. [VKW+12], employing hierarchical OBB-based and Skeleton-based splitting. In the context of parks, parcel splitting methods could be employed for partitioning of flower-beds, for example, as well as fitting of a simpler polygon into a more complex block in the latter case. The CGA rules allow more sophisticated partitioning, for instance, arbitrary splitting along a local axis and polygon offsetting, which can be used to create both paths and various non-walkable regions of a quarter.

Example-based methods could also be adapted to synthesise park layouts. Yang [YWVW13] uses elements of example-based synthesis and to select a block splitting patterns from a template database into the regions between major roads. Based on layout examples extracted from images, Aliaga et al. [AVB08], uses operations *join*, *expand* and *blend* to synthesise new layouts, where the join operation of creating new segments is based on probabilistic random walk decided by angle and rotation distribution. One of the results from the Aliaga's work may resemble a complex park layout (see Figure 3.4).

Lipp [LSWW11] uses *graph cut* to merge multiple layouts coming from the CityEngine. This and the layer-based method [IMAW15] could be used to merge multiple park patterns into one. Sub-park rectangular regions could be merged at entrances by solving a set of linear equations, as described in the example of game levels by Ma et al. [MVLS14].

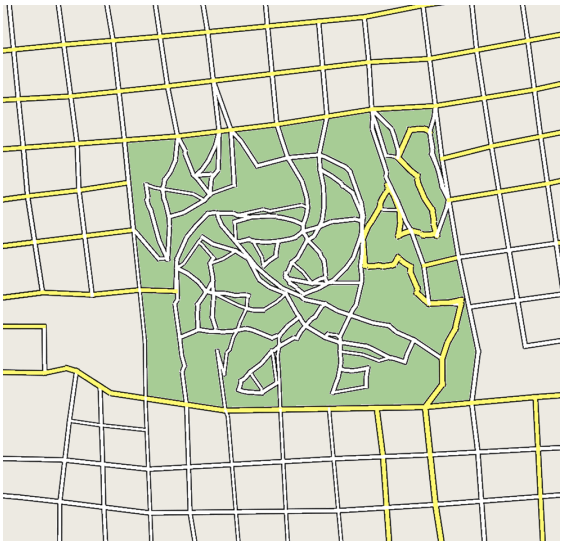


Figure 3.4: Street graph generating methods could be applied to Park Modelling, Aliaga et al. [AVB08]

3.4 Object Placement

Ulmer [Ulm05] developed a grammar targeting vegetation placement into three types of objects: avenues, inner plots, and whole parks, including distributions and jittered placement along lines and circles. Deussen et al. [DHL+98] simulated a vegetation ecosystem, taking into account plant lifespans by pipelining Floyd-Steinberg algorithm with Voronoi Diagram to create initial seed samples (see Figure 3.6). These were later self thinned-out when simulating the growth process. Merrell et al. [MSL+11] tries to encode design decisions based on functional and visual criteria, accessibility to people in the context of furniture placement within the room layouts, which could be translated into the placement of park furniture. Guerrero et al. [GJWW14] considers example-based synthesis driven by an interactive user input in the context of object placement, based on

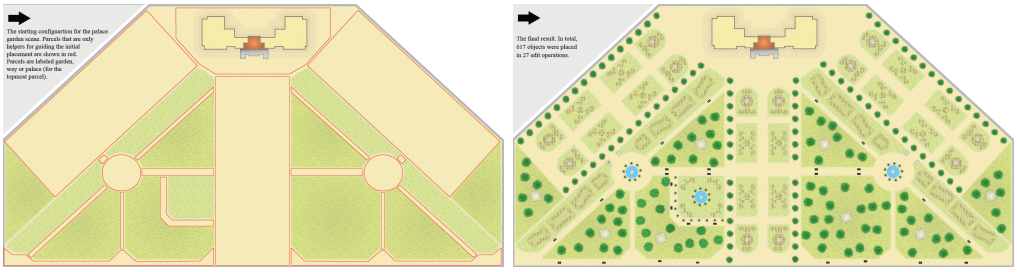


Figure 3.5: Park example using Edit Propagation, left: the initial layout, right: the result; Guerrero et al. [GJWW14].

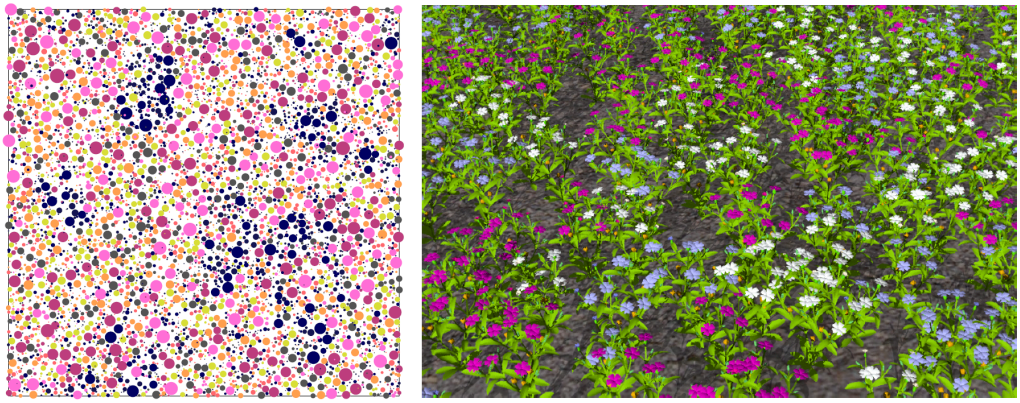


Figure 3.6: Plant Ecosystems, Deussen et al. [DHL+98].

geometric relationships or 'poses' between the placed instance and the region containing it, to replicate the same placement in new regions. An example placement propagation within a manually created park layout was given in see Figure 3.5.

Sampling is used for rendering, and surface remeshing, but is also ideally suited for 'natural' distribution of objects on a surface. Roughly speaking, sampling is a conversion of a continuous distribution into a discrete and Cook [Coo86] mentions that merely randomly 'throwing a dart' will sometimes produce two samples too close to each other, and others would be too far apart, in other words, sampling a range of frequencies or white noise. By sampling with a higher frequency or blue noise, only small jitter, or variations of distances between samples, is allowed. Poisson distribution is based on placing samples closely but at a minimum *radius* from each other. Such distribution suits the purpose well since it was also observed in nature, for instance in the retinal photoreceptors of animals' eyes. The *jittered grid* method approximates Poisson distribution and is considerably more efficient over the 'naïve' removal of collisions. Recently, more efficient Poisson sampling methods have been described, for instance, a one based on grid data structures [Bri07].

CHAPTER 4

Layout Generation

The overview (Sec. 1.2) mentions the separation of the park generation into the library back-end and the host application front-end. The content of this chapter is limited to the primary component of the back-end, the **layout generator**. The input of the layout generator includes the rules file and the axiom region. It is optionally complemented with a list of attributes provided by a user to override the ones stored in the rule file. The output is a dense subdivision of the axiom region into subregions. The axiom region needs to be a quad or mapped to a quad (see Section 5.2), and it geometrically limits all of the layout synthesis.

4.1 Definitions

In the context of this text the terms *shape*, *region* and *polygon* refer to the same geometric object – a simple planar polygon without holes, vertices of which are in $\mathbb{R}^2$ – with the following differences. Shape, being part of the production system, besides geometry also contains indexing, attributes and any additional meta-shapes (e.g. *hint shapes*, Sec. 5.2.2). A region may refer to both a polygon and a shape but may be used outside of the production context. The input region in a context of a single rule application we call the **iregion**.

The structural rules partition the input shape entirely, producing two types of regions: **path**s – where one can walk upon, and **quarter**s – non-walkable regions, which are disjoint. Path regions are furthermore divided into **junction**s and **section**s. A junction can be replaced with a **new shape**, for instance, a primitive, forming a **custom junction**. Sections join at junctions, and the two, in the case of the structural partitioning, can be loosely thought of as nodes and edges of a graph. A section connects the boundary at an **entrance**. A region type is identified by the `type` attribute and the rules are not barred from overriding it.

4.2 Grammar for Park Synthesis

Our grammar is similar to the CGA with a much-simplified syntax (see Appendix A for grammar reference). It does not allow multiple statements in a production and is mostly not context sensitive. The rule statement consists of the rule name and **parameters**, followed by a set of **selector block**s. A selector block consists of a number **selector statement**s. During the derivation a single selector block corresponds to a set of shapes with a single *label*. Variation is achieved by **selector**s that when matched, choose the label for the derived shape. These could be *stochastic*, *index-based*, or a *border* selector. The latter is matched when the derived shape touches the iregion boundary. Besides insertion of primitives, we allow insertion of *smoothed* and/or *shrunk* iregion into itself.

The grammar is supplemented with *global* attributes and *shape* attributes. Shape attributes are propagated to all descendant shapes. The attributes serve a similar function as in CityEngine. Besides being accessible during production, attributes are retained in the terminal shapes and could be used as semantic tagging of regions.

The system has the following rules for structural partitioning that results in quarters: Grid, Cells and Rays. For boundary-based partitioning, Peel rule can be used. For object placement Insert and Place rule can be used.

4.3 Grid Rule

Although it is possible to partition a park into a grid-like structure of paths using two consecutive applications of the CGA *split* rule along x and y axes, the **Grid** rule aims to achieve the whole operation within a single production, incorporating handling of custom junctions. The most basic example of this can be seen in the Figure 4.4a. The two sets of perpendicular path polygons form *junction*s when intersected (see Figure 4.1). Each of these polygons is extended beyond the boundary for proper partitioning of non-standard iregions – such cases are described Section 5.2. *Path sections* are produced by subtracting the junction regions from the union of the paths (shown in red in the Figure 1.10c). *Quarter* regions are created by subtracting the union of paths from the input region.

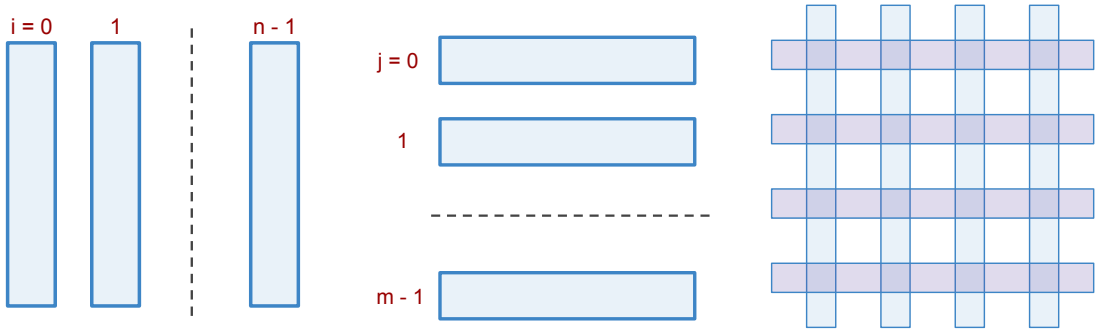


Figure 4.1: Grid path construction uses vertical (left) and horizontal (middle) path regions, which when combined, form a grid (right).

4.3.1 Symmetry

Symmetry can be observed to be an integral part of grid-based designs. We allow only 2-by-2 symmetry of regions, in other words mirroring of content in the lower left quarter region against the local y and x axes in the three other quarters. When more than two quarters are specified, symmetry is simply extended using modulo-2 arithmetics. Symmetry is achieved by setting the appropriate orientation and offset of the quarter indexing (See Chapter 5, and in particular, Section 5.1.1).

4.3.2 Quarter Indices

Rule selectors allow targeting a particular Grid quarter with i and j indices along the x and y axes correspondingly. However, the geometry library functions do not return regions arranged in a grid pattern, and we assume the ordering is non-deterministic (see Figure 4.2 left). Quarter indices are obtained by performing a *spatial query* at the *centre* of the region (see Figure 4.2 right)using an R-tree [Gut84]. This results in $O(n \log n)$ complexity, however, we are aware that a linear lookup method is possible since limits of each quarter are well defined. Sometimes a part containing the centre point is clipped away, for instance by the iregion boundary. In such a case the quarter is terminated, resulting in grass-only content.

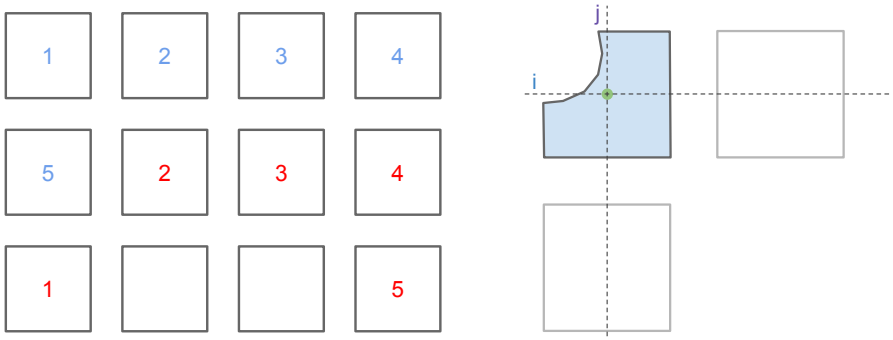


Figure 4.2: Left: desired Grid quarter ordering shown in blue, whereas the actual order of regions that result from geometric operations can be different (for example, in red). Right: even when a considerable chunk of a quarter is clipped away, there is a high likelihood that the point (green) at the centre of the unclipped version of the quarter is also found within the clipped quarter.

4.4 Cells Rule

The **Cells** rule aims to generate park and garden designs, which are dominated by regions with curved boundaries (see Figure. 4.3 left).

Early on, we have experimented with the generation of curved regions in single-junction parks by using ellipse functions (see Figure 4.4b), and Bezier curves (see Figure 4.4c and

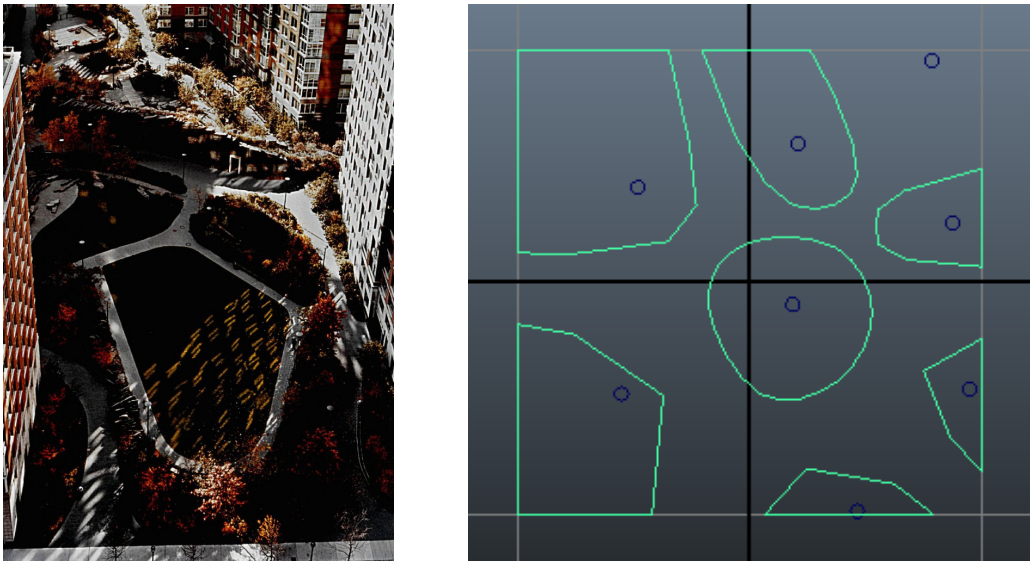
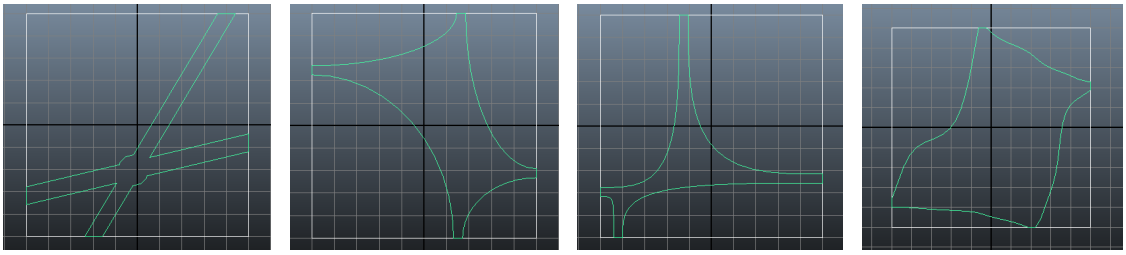


Figure 4.3: Free-form layouts. Left: a park between blocks of flats in Manhattan. Right: cells layout created by smoothing out Voronoi cells.



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

Figure 4.4: Early approaches for creating park layouts with a single junction. (a) Intersection of paths spawned by random positions along the pairs of opposite sides with a circular junction. (b) Based on Quarters created by Ellipses placed at the corners. (c) Based on Quarters created by Bezier curves. (d) Similar to (c), with a random permutation of control points.

see Figure 4.4d). The core problem, however, is to find a solution for multiple junctions, while avoiding development of the grid pattern. In the end, Voronoi region partitioning and subdivision-based smoothing have been selected, because of the simplicity and close observed resemblance to some of the real-life designs (see Figure 4.3 right). Moreover, Voronoi Diagrams had been employed in other layout generating algorithms, for instance, those used in games (e.g. [Gam]).

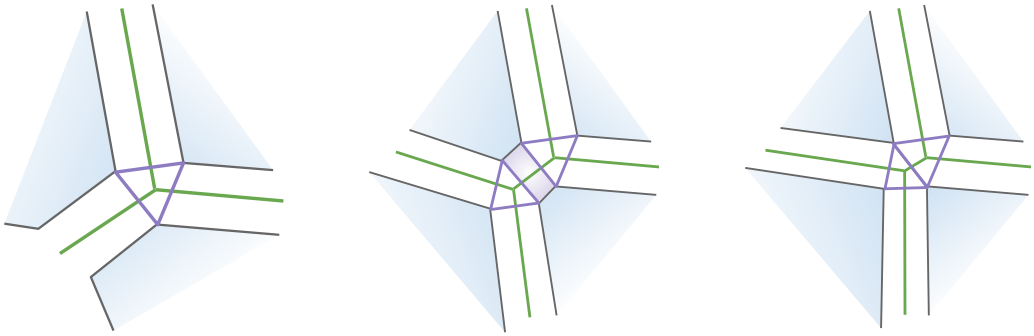


Figure 4.5: Junctions in the Cells rule. Left: correctly created junction (the purple shape) from the three vertices that resulted from shifting three neighbouring cells. Middle: when two cell neighbour vertices are close enough to each other, they may merge, collapsing the section; right: correct placement of junctions after the section collapse.

4.4.1 Cells Partitioning

Layout generation starts with the generation of 2D Poisson point samples (see Chapter 6). The Voronoi diagram is then generated by using these samples as cell vertices. Each resulting cell is shrunk by moving edges away from the cell boundary (see Section 4.4.2) and then smoothed. The resulting regions form park quarters. To create 'blocky' quarters with sharp corners, smoothing can be disabled. The space between the quarters is taken by the path regions, as described next.

Path Creation Algorithm

1. Shrink each cell.

2. Each shifted edge of the shrunk quarter is paired with the *twin edge* of the neighbouring quarter to create a path section.

3. Since a vertex is shared by a number of cells (usually three), shifted versions of the same vertex from each cell are used to construct the junction (see Figure 4.5 left).

4.4.2 Cell Shrinking and Grassfire Transform

A naïve polygon shrinking algorithm moves each edge "inside" along the normal by a given offset value t. The new intersections for each edge pair create new vertex positions. Shrinking a polygon in such a way, however, may result in self-intersection (see Figure 5.9 left). This occurs when an **edge event** which is an *edge collapse* are not handled correctly, in a process known as the *wavefront propagation* [AA96] and sometimes referred to as the *grassfire transform*. When this happens, the two neighbouring edges 'close in' from both sides, and should be merged into a single vertex. General methods for reliably solving

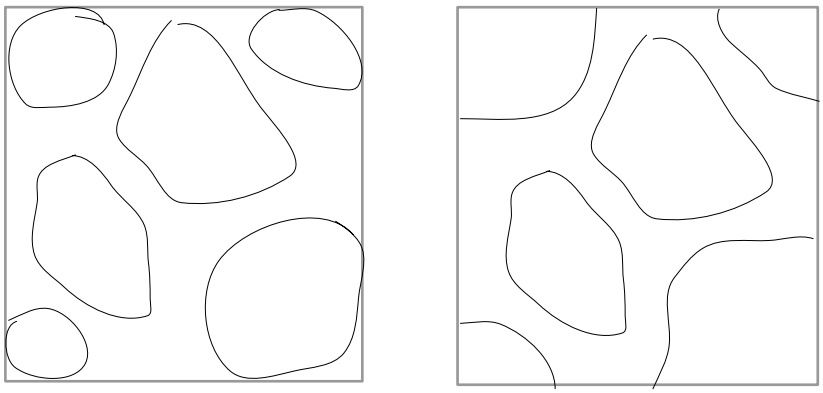


Figure 4.6: Left: clipping before smoothing of boundary cells results in blobs; right: desired outcome.

this offsetting problem exist, but are rather involved, for instance, are based on tracking collapse times of triangles in a priority queue (such as by Palfrader and Held [PH15]).

We derive a more simple method that is limited to handling edge events in a convex polygon, which also tracks the *existing indices* in the process – it is considered in Section 5.1.6. Since a number of edges may have collapsed, the respective path sections are removed and the adjacent junctions are connected (see Figure 4.5 right, compare to an almost collapsing section Figure 4.5 middle).

4.4.3 Smoothing

Since we deal with discrete geometry, a curve is approximated by a polyline with high geometric detail. A polyline and a polygon can be smoothed, getting nearer to a curve, where the number of edges rises and the sharpness of the angles is reduced. Since new geometry is created, to preserve topology a group of edges can be abstracted under a *logical edge* (see Section 5.1), that represented a single edge in the original polygon. Preservation of topology is important, which is needed for subsequent construction of path sections and junctions. The smoothing algorithm itself consists of simply mixing each vertex with its neighbour's average, $v' = mix(v, \frac{v_{prev}+v_{next}}{2}, weight_v)$, where the weight determines the sharpness of the result. Usually, a number of passes are performed to achieve the desired results. We perform an adaptive smoothing, applying the algorithm repeatedly until the magnitude of the *longest* edge is below a given threshold that is determined by an attribute.

Since the outer cells of the Voronoi diagram are half-open, they need to be clipped before smoothing. Clipping at the boundary results in blobs (see Figure 4.6 left, while the look of the Figure 4.6 right is more desirable). We resort to using a region slightly larger than the iregion to clip the outer cells before smoothing.

4.4.4 Symmetry

Sample distribution can be guided using symmetry against either x or y or both axes, and this allows to create symmetrical regions. However, since Voronoi Diagram generation and the subsequent smoothing is applied on the clipped samples instead of the quarters, symmetry may not be perfectly reflected at such a boundary that is not symmetrical itself.

4.5 Rays Rule

Rays rule implements the Rays pattern described in Section 2.3.2, where path placement is compared to casting a ray, although the construction process differs a bit.

Partitioning starts with the creation of the path network. Path creation consists of placement of junctions followed by the creation of sections between these junctions and the boundary. In Rays rule paths are classified into two types, the entrance sections – those that touch the boundary, and the inner sections – those connecting the junctions.

4.5.1 Junction Placement

Junction positions are generated by sampling the iregion Poisson distribution (Sec. 3.4). This is contrasted to the Cells rule, where samples are used to generate quarters. In addition to the Poisson radius, offset from the boundary is also specified, which determines how far the envelope, containing all of the junctions, is from the boundary.

4.5.2 Outer Sections

Entrances are locations on the iregion boundary, and need to be specified using the discrete selection (see Section 5.3.2). Each entrance is connected to the nearest junction. R-tree is used to accelerate the search.

4.5.3 Inner Sections

Larger parks with the observed Rays pattern often have multiple junctions connected with sections in some way. Junctions and inner sections can be viewed as nodes and edges of a *connected graph*, which should be *planar* since we do not consider bridges. Initially, we used a sequential merging solution to connect junctions in such a graph by successively attaching a randomly selected one from the set of kNN-queried neighbour junctions, keeping track of already connected ones. This graph, which is also a spanning tree, however, may contain intersections between sections, meaning the graph will no longer be planar (see Figure 4.7).

Instead, a more robust construction using Delaunay triangulation is performed, which is the dual of the Voronoi Diagram. Simply using the triangulation graph, where the connections are maximal, results in a park looking rather artificial, mesh-like (see Figure 8.9). To improve this, we compute spanning tree using a standard algorithm

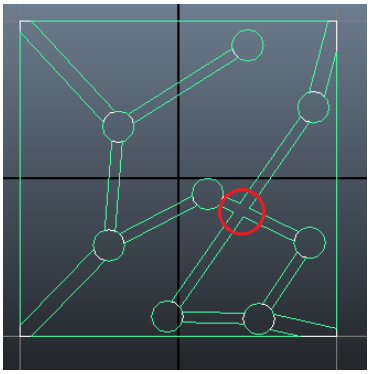


Figure 4.7: kNN-based junction connection – intersection is visible.

(Kruskal's) and then randomly mix in few edges that are found in the maximal graph but not in this tree (see Figure 4.8). The number of additional edges is specified by an attribute in percent. The sequential merging of sections' regions remains the same.

4.5.4 Junctions

The Rays rule employs circular junctions, which not only occur in real parks but are particularly suitable since path sections meet at arbitrary angles. We need to consider overlaps between path sections to avoid geometric errors (see Figure 4.9 left). Avoiding overlaps places a number of design limits with respect to the path width, junction radius and the angle between path sections (see Figure 4.9 right). However, to avoid excessively large junctions, we found that limiting the angle is impractical since sharp angles occur quite often. To resolve overlaps, even when sections touch each other at junctions, we clip and accumulate them into an initially empty *set of disjoint paths*, $S_d = \emptyset$. Before a new section s_i is added, the existing set is subtracted from it, $s'_i = s_i - S_d$; $S_d = S_d \cup s'_i$. Please note that geometric difference operator '$-$' is used (see Section 7.1.3) instead of set '$\setminus$' operator, since the path set is formed from a set of separate regions. An extension to custom junctions (see Section 4.1) is a trivial task.

4.6 Boundary Insetting, Peel Rule

Rules described up until now partition a larger iregion into a number of smaller tile-like regions. Such a region, a quarter, can be further partitioned at the more local level by 'insetting' a part of the boundary into the region. Unlike *Setback* and collection of corner-snapping *Shape** rules of CityEngine, we would like to inset an *arbitrary subset of the boundary* that is not necessary snapped to a vertex, for example, observe the hedges of each park quarter in the Figure 4.11. Such inner offsetting using an arbitrary boundary subset allows a more varied design and can be appealing to an eye.

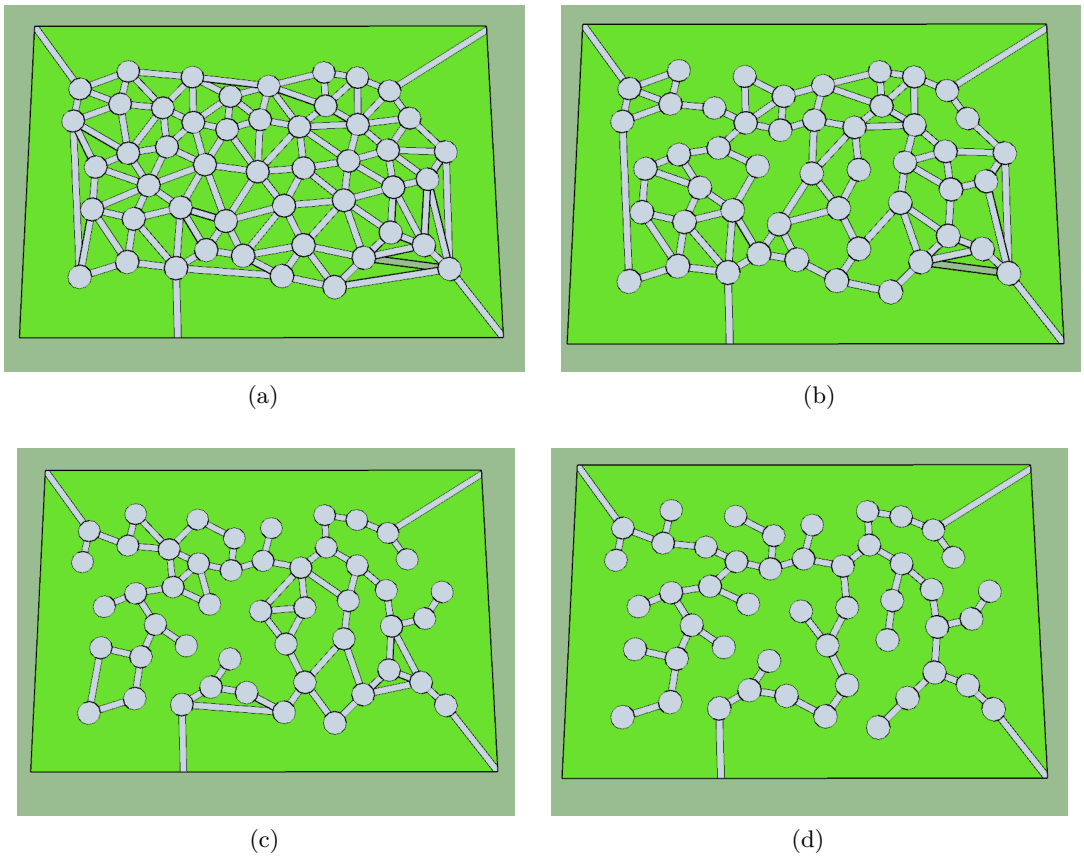


Figure 4.8: Rays path thinout: (a) the dense triangulation-based paths without thinout (ratio 1), compare to (d) – the spanning tree of (a), i.e. a completely thinned out path network (ratio 0); (b) a thinout with ratio 0.5 and (c) 0.2.

4.6.1 Peel Rule

Peel rule is used to offset a selected subset of the boundary inwards, creating an **intruded** shape (the light red shape in Figure 4.12 middle), which cuts into the iregion. The remainder shape we call the **substrate** (the light blue shape in Figure 4.12 middle). The boundary subset is supplied as a list of *boundary selection* intervals (Sec. 5.3). Multiple shapes could result from a production.

Shape Extension

For a closer correspondence to the real-world designs, the intruded shape has been extended in the following two cases. First, when boundary selection snaps at vertices and an angle to an edge adjacent to the selection is less than 45°, the intruded shape is extended to meet such an edge, as shown in the Figure 4.10, left. Second, when two

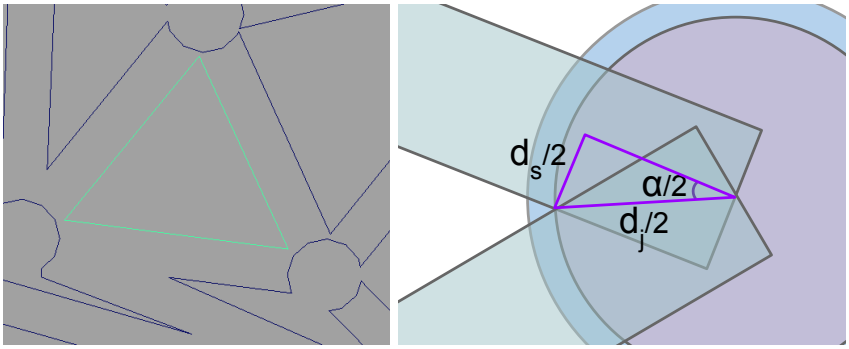


Figure 4.9: Special treatment of the path geometry in the Rays rule. Left: we cannot simply create a union of paths sections and then subtract the junctions to get the clipped regions since often occurring section overlaps result in merges of a number of such input regions into the more complex polygons (dark blue), as well as some holes (green). Right: Sizes of a junction (blue region) and sections (green regions). Given a path width (d_s) and an angle between path sections (α), the junction diameter (d_j) should be larger than that of the pink region, $d_j > \frac{d_s/2}{sin(\alpha/2)}$.

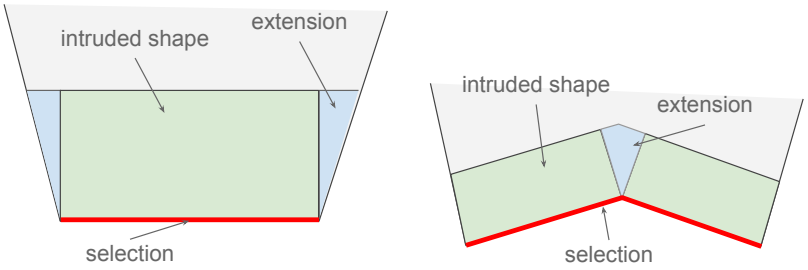


Figure 4.10: Peel rule intruded shape extensions. Left: extensions at corners reaching two of the adjacent edges. Right: an extension between two intruded shapes that share a vertex with an angle greather than 180°.

intruded shapes meet at a vertex that forms an inner angle that is greater than 180°, a completing polygon is inserted (see Figure 4.10 right).

4.6.2 Inset-based Partitioning

We can use two or more applications of Peel to insert a shape. With the help of labelling (refer to Sec. 5.1.2) we can locate the new boundary between the intruded and the substrate shapes. For example, this can represent bush hedges (as shown in Figure 4.11) in the park quarter (see Figure 4.12). Moreover, Peel operations based on such propagated labelling can be combined to achieve a certain design, where the original boundary selection is used as a guide.

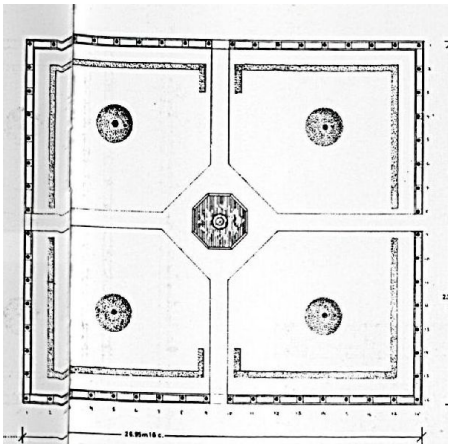


Figure 4.11: Quarter hedges of a grid-like park layout appear to be 'inset' at a certain distance from the boundary.

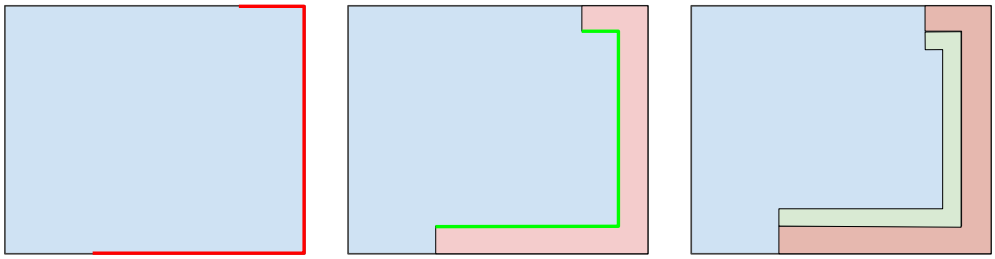


Figure 4.12: Left: boundary selection in bright red on the initial region, for example a grid quarter. Middle: Peel rule generates light red region. Bright green selection using a special labelling (Sec. 5.1.2), that takes into the consideration the previous production. Right: Peel applied second time, resulting in a light green region region that is isolated from the boundary.)

4.7 Placement Rules

Placement rules insert new shapes into the iregion. New shapes resulting from a Place or an Insert rule production are treated as *new entities*, while in comparison, in custom junctions, new shapes *take the place* of the default shapes that would still exist otherwise.

4.7.1 Insert Operator

All new shapes are specified with the **insert** operator. The insert operator is invoked within the selector block and it is only evaluated if the rule *accepts* a new shape in the given context. For instance, a placement rule *requires* a new shape, and respectively, the insert operator, while in the junction block of the Grid a new shape is *optional*.

There are three basic *built-in shapes* that could be inserted - **square**, **circle** and

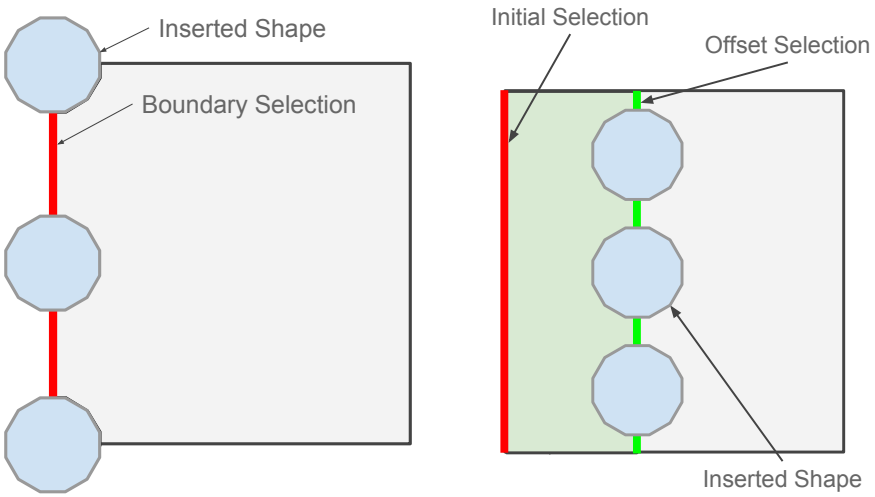


Figure 4.13: Left: Insert rule with three new shape instances. Inserted shapes are clipped to the iregion subsequently (not shown). Right: Place rule with three new shape instances.

rhombus. These are the shapes often observed in real-life park layout. The shape operator takes two, x and y, parameters. When $x \neq y$, square and circle shapes become *rectangle* and an *ellipse* respectively.

4.7.2 Insert Rule

Insert rule is used to insert new shapes along the iregion boundary. Two arguments are given, the boundary subset selection (see Section 5.3.1), and the sample discrete distribution. The boundary subset is mapped into the [0..1] range, which is then used to distribute the samples over (see Section 5.3.2). A shape is inserted with its centre placed at the sample position (see Figure 4.13 left). New shapes are then clipped against the original input shape and the input shaped is clipped against all of the new shapes.

4.7.3 Place Rule

Whereas in the Insert rule new shapes are placed directly on the boundary, Place rule inserts shapes in the middle of the input shape. Hence, an additional boundary *offset* parameter is given. To accomplish such process, first, the Peel rule is applied to the selected boundary subset. This results in an intruded shape with the propagated selection – the green *offset selection* as shown in the Figure 4.13 right (propagation is the same process as illustrated in the Figure 4.12 which is described in Section 5.1.2). Insert rule is subsequently applied to the offset selection, which is then situated entirely within the initial input shape. It is important to mention that both the intruded and the substrate shapes are clipped against the new shapes.

Centre placement was considered as a separate step. To make the insertion more robust *for all cases*, we avoid having to deduce the correct Peel and Insert parameters. Instead, we split the input shape along the line which passes through the centroid, and then a *single* new shape is inserted *at the centroid* of the input shape.

4.8 Push Extrusion

Regions could be extruded upwards into the 3D shapes by setting the *elevation* attribute. Regions with the *type* attribute values of "bushes" or "border" have implicit *elevation* values set, which still can be overridden. The extrusion, or a *push* operation, is performed by the front-end. The order of *push*es is important. Given two neighbouring regions with different *elevation* values and materials, when the lower is pushed first, the higher region side-face gets assigned the material of the lower region at the common edge boundary. To solve this, we group all regions by height, storing them in a Ruby dictionary, and push the regions starting from the highest in descending order.

CHAPTER 5

Indexing and Selection

Selection (Sec. 5.3) is used by rules to target a part of the iregion boundary. It is passed to a rule as one of the arguments. **Indexing** (Sec. 5.1) describes how the boundary is labelled in a way that is meaningful to the selection step. Indexing is propagated at each derivation step. Grid rule requires a special **region fitting** (Sec. 5.2) step – fitting a complex polygon to a simpler one, to enable correct path placement, and can be considered as a sub-task of indexing.

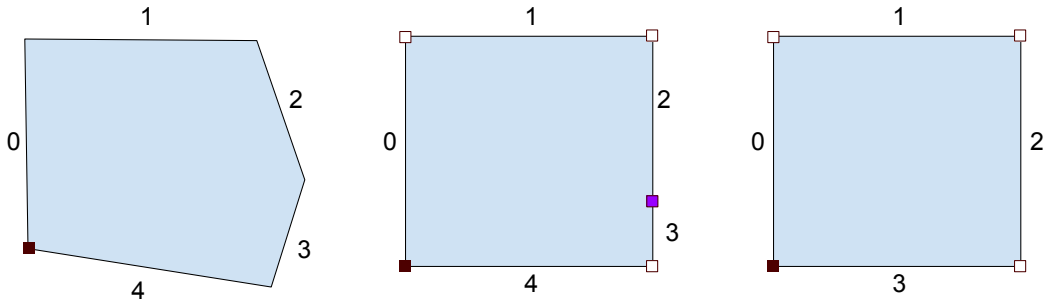


Figure 5.1: Left: default indexing follows the actual order of edges in the geometry data. Middle: a quad shape that has physical representation of 5-sided polygon Right: logical representation of the shape in the middle, a quad.

5.1 Indexing

After introducing the indexing subsystem, we describe how to index an empty region using *detection* (Sec. 5.1.3). **Indexing propagation** fills gaps in indexing when an existing shape becomes fragmented (Sec. 5.1.4). A rule can also use indexing internally for its means, as described in Section 5.1.6.

Indexing is numeric labelling of edges of the shape polygon. In a default context, it coincides with the 'physical' sequence of edges in the data structure or the file. Indexing starts with 0 and ends with $n-1$ ($n = \|E\| = \|V\|$) (see Figure 5.1 left). We use clockwise (CW) enumeration direction to follow the convention of Boost Geometry library (Sec. 7.1.3), without the loss of generality to the counterclockwise direction. The main problem with such one-to-one indexing is that it is tightly coupled with the geometry. For example, more complex polygon (see Figure 5.1 middle) should be represented by a quad, because the extra complexity in geometry that does not alter the perception of the polygon and should be either ignored or discarded (see Figure 5.1 right, the violet vertex from the centre image is not be counted). This detail is important in the field of procedural modelling for the preservation of design. When a modification to the iregion is performed, for instance, an external change to the street or block level (Sec. 1.1.1), changes to the final look of the model should be minimised. The same applies to the intermediate shapes within our system.

5.1.1 Logical Edges

We were often concerned only what a shape looks like roughly. The detail of the actual geometry may be unnecessarily high, for instance when trying to approximate a curve. We introduce the **logical edge** (LE), $LE = \{E_i, E_{i+1}, ...E_j\}$, an abstraction of physical geometry to logical geometry that can be selected and manipulated by a rule. Henceforth, physical edges of a polygon are grouped into LEs, which attempt to abstract fine geometric detail into a set of meaningful structures. All indexing in our system is based on logical edges. Logical edges allow to label groups of edges in a polygon arbitrarily. In practice, however, the concept naturally fulfils the role of numeric labelling, where labels are given by an LE index. Moreover, indices may be skipped, i.e. holes created, using empty LEs where $i = j$. An example of this can be found after an edge collapse in Section 5.1.6.

5.1.2 Polygon Labelling and Problems

After review of various labelling representation we decided on 'flat' numerical labelling, or indexing. By that we mean an association of a label with a single integer value. Such representation has a few advantages:

Generic with respect to the production context. Labelling that applies to all partitioning rules, while preserving the same semantics where possible, is prefered. Namely it should work equally well with the Peel or the Grid rules. As an alternative we considered using various categories of semantic labelling. This could be based on proximity, for instance, closeness to the 'path' type regions. Or it could be dependent on the current production context, such as label names `peeled` or `inner_quarter`, and so on. This is obviously problematic when different rules are mixed together, not to mention the difficulty of implementing such a system.

Orientation-independent. In contrast, CityEngine uses semantic labelling based on directions allowing, for instance, to select the North-facing edges or the 'front' of

the polygon within the local transformation context. This is useful for building modelling. Yet, even if the transformation contexts are perfectly synchronised, this is not as useful for modelling of a park, where an element often follows the boundary for an arbitrary extent.

Allowing numeric selection. Indexing allows selection using continuous numeric ranges. For instance, $[0, 0.5]$ refers to the first half of the shape's first LE.

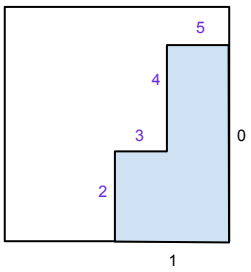


Figure 5.2: Inherent amplification of numeric indexing can be demonstrated on the Peel rule, where resulting number of indices will be $2n + 2$.

The above mentioned points have prompted us to decide for numeric labelling based on logical edges. However, even then, naïve enumeration of LEs has led to the explosion of the index sets – in practice, this has turned out to be somewhat chaotic after a number of productions. This problem can be exemplified by applying the Peel rule on more than one LE, where its output shapes have higher geometric complexity (see Figure 5.2), which is in turn accumulated with successive applications within the same locality. One solution would have been to use sub-indexing. However, this would complicate the system even further. As a result, a simpler indexing system was devised and is described next, which we use by default.

01-Indexing

We define a 'binary' partition of the boundary into two LEs, inner and outer, the **'01'-indexing**. The **outer** logical edge labelled with **0**, containing edges spanned by vertices that are found on the boundary of the parent shape. The **inner** LE is labelled with **1** and contains all the other edges (see Figure 5.3). The result is an indexable range $\in \{0, 1\}$, which allows for a continuous selection range of $[0, 2)$, as discussed later in this Chapter.

An alternative mode of 01-indexing, triggered by an attribute, defines *outer* edges as those that *contain* the boundary vertex of a parent shape Figure 5.4. Such alternative is useful in a context when a set of edges needs to be selected that does not touch the boundary, for instance in a peel rule.

This 'sufficiently minimal' indexing mode is particularly suitable for the context of park layouts since the majority of partitioning is observed to be relative to the boundary

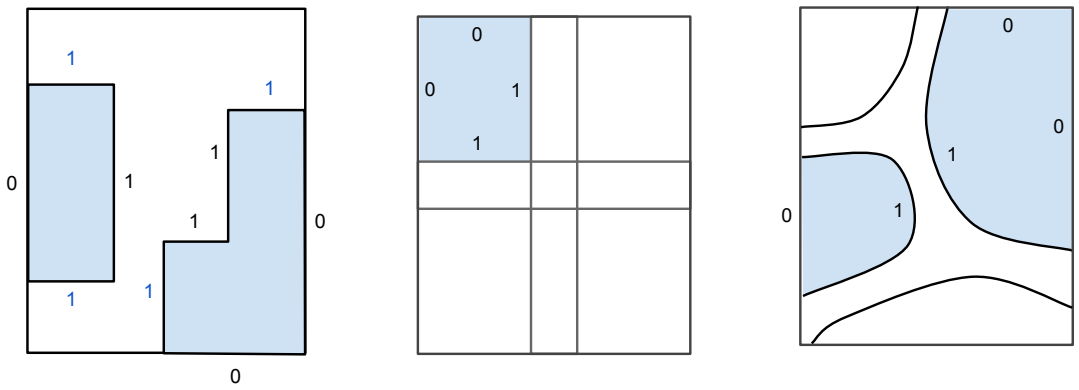


Figure 5.3: 01-Indexing logical edge labelling on example application of the (left to right) Peel, Grid and Cells rules.

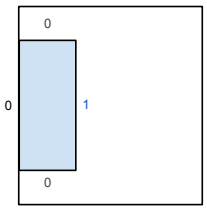


Figure 5.4: Alternative 01-Indexing mode on the example of the Peel rule. Compare to the left blue-shaded region on the Figure 5.3 left.

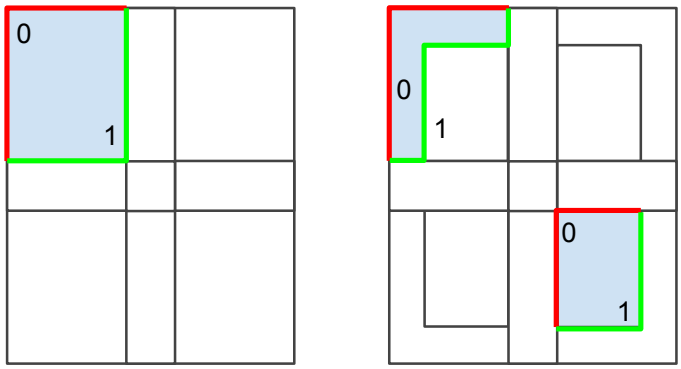


Figure 5.5: Example of the 01-indexing demonstrated by first applying the Grid rule (left), and then the Peel rule (right) on the **0**-subset of the boundary of each quarter.

of the container (the parent shape). It delivers enough expressiveness and is quite simple at the same time. Anything smaller than that, for example, the unary indexing would allow selection range of $[0, 1)$, is not systematic enough to provide ordered results since

in park designs it is often important to distinguish between inner and outer boundaries. Indexing of higher arity, for instance, 4-nary indexing, already accumulates unnecessary complication with subsequent rule application.

All rules that alter geometry in our system produce 01-indexed shapes. Internally, a rule may use arbitrary labelling, as long as the resulting shapes are 01-indexed. Offset-partitioning shapes propagate (see an example on the Figure 5.5) instead of synthesising new indexing. The exception is the newly inserted shapes, as described in Section 4.7, where the inserted shape does not share any context with the target shapes, so the propagation of the existing labelling is irrelevant. The axiom shape does not have to be 01-indexed. The user is responsible for indexing the axiom shape (see Section 5.1.3 below), which can be arbitrary and is, of course, reflected in the productions. This is useful when one wishes to determine the layout of the initial Grid production by supplying a shape with four indices (see the end of the Section 5.2.2).

Important to note that our system does completely implement indexing because the *path* shapes remain default indexed.

5.1.3 Indexing Detection

A user may wish to partition the axiom shape into k number of logical edges. For automation of this, we resort to using geometry simplification techniques, which must occur on the axiom shape as a separate sub-step. Moreover, the resulting k vertices can be used used to construct a fitting polygon (Sec. 5.2). Previous work in this direction has been explored by authors such as Leu and Chen [LC88]. They came up with a solution, also described in Boxer at al. [BCMRC93], that looks at the set of 2 or 3 consecutive edges, and if any vertices within this set differ by a significant enough distance from the 'chord', formed from the endpoints this set.

We choose to use a simpler metric based on angle of edges incident to a vertex, although we have also considered growing-based and adaptive methods. The sharper the angle, the more prominent is the vertex and hence should provide a better fit when included. To simplify the polygon we select k vertices with the best metric, i.e. largest angle, and from LEs between them. We call this method **Angle Thresholding**. Selected vertices are sorted in the order of occurrence in the original polygon, in order to minimise the possibility of self-intersections.

5.1.4 Indexing Propagation

01-Indexing consists of two continuous ranges, and when it cannot be applied, **propagation** of parent indexing must occur on the derived shape(s). For example, when the Insert rule is applied, new shapes 'cut away' parts of the input, splicing the boundary, Figure 5.6. If the new geometry is surrounded by the same index, it is assigned this index as well. The problem arises when the new geometry is surrounded by edges with different indices after the insertion. Such boundary cases are labelled by extending the label that occurs first in the indexing sequence, until the geometry with the other label is met. Such simple strategy is preferable to covering edges from both endpoints that

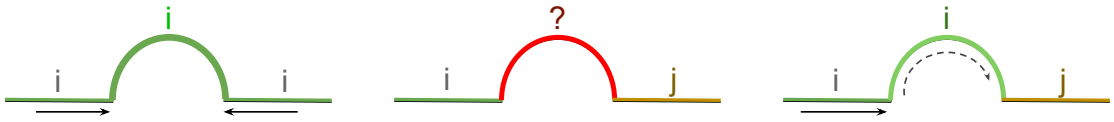


Figure 5.6: Indexing continuity is broken by the inserted shape. Left: geometry surrounded by one index gets the same index. Middle: how should indexing be propagated when from both sides differ? Right: extending the index in the direction of orientation.

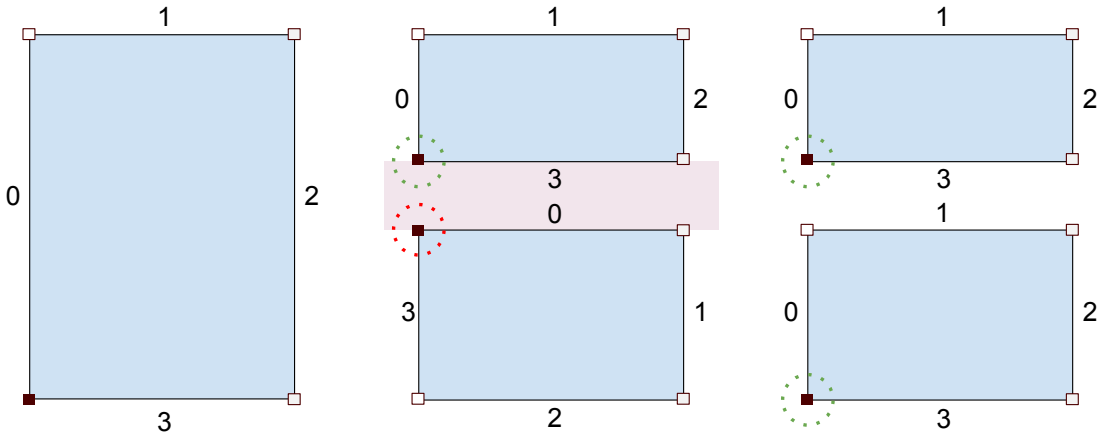


Figure 5.7: Example case for reordering of indices. Left: physical geometry ordering in iregion. Middle: geometry ordering after subtraction of, which could be a vertical path shape in a Grid production, at the middle of the iregion. In the bottom region incorrect orientation can be observed. Right: correct geometry ordering after reordering of indices.

meet in the middle of the new geometry at some prominent vertex (as mentioned in Section 5.1.3) since that would result in fragmentation.

5.1.5 Reordering of Indices for Grid Quarters

Grid rule is dependent on the order of logical edges in the iregion for the orientation of resulting shapes. Symmetry is also determined by the logical edge ordering (Sec. 4.3.1). However, the geometric library does not guarantee the same ordering of elements in the output, in other words, the ordering of results is non-deterministic (see Figure 5.7 left and middle). Hence, iregion indices order needs to be synced to the output subregions, where, in our example, the first LE is at the West of the region (see Figure 5.7 right). Based on their positions within the region (logical) edges can be correctly re-adjusted within the [0..3] indexing set by using the Algorithm 5.1. Afterwards the symmetry can be enforced by further modification to the indexing.

Algorithm 5.1: Calculation of the (logical) edge index based on the (logical) edge position within the region. As a reference, Figure 5.8 shows index for each direction of an edge that is represented by a point.

```
   // (logical) edge direction based on its (endpoint)
      vertices
1  d_i ← v_{i+1} − v_i
   // compute angle.  Arctan2 arguments are inverted.
2  α ← atan2(d_i.y, d_i.x)
   // scale to [-2.0, 2.0]
3  α ← α * 2.0/π
   // round and convert to [0, 3] range
4  index = modulus(round(α + 4), 4)
```

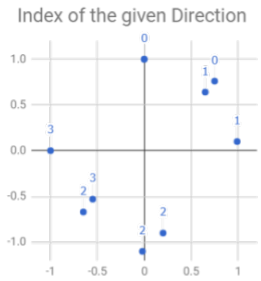


Figure 5.8: Directions, represented by points, are assigned an index within the range [0..3]

5.1.6 Index Mapping: Shrinking Case Study

The Cells rule uses *grassfire*-based shrinking to create sections and junctions (Sec. 4.4.2) and uses indexing internally to preserve geometric relationships between quarters and path regions. When edge events occur, correct re-indexing or index propagation must be performed to ensure the validity of the regions. We perform naïve shrinking and then test for self-intersection to check if any edge events need to be resolved. The goal is to merge multiple adjacent edge events, which we call *event clusters*, at once into a single 'merged' vertex. We observed that such cluster will result in a single intersection between non-neighbouring edges that enclose it. Since we know the indices of the intersected edges, we simply propagate them to the new edges formed at the intersection. The Algorithm 5.2 combines the two tasks, merging of clusters and re-indexing. When new correctly shrunk polygon is constructed, either a non-event vertex or the cluster intersection is used for each new vertex. The **index map** is created based on the cluster pair.

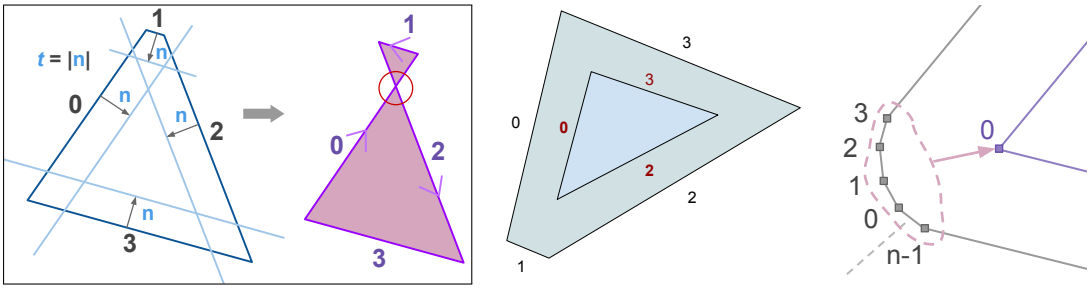


Figure 5.9: Left: naïve Shrinking does not handle edge events. Middle: correct reindexing of shrunk polygon. Right: correct indices after multiple edge collapses at the indexing boundary.

Algorithm 5.2: Combined shrinking and re-indexing.

Input: Self-intersecting naïvely shrunk polygon p
Result: Correctly shrunk polygon p', index map m

```
1  begin compute the event cluster index set
       Result: a list N of cluster events, storing surrounding cluster indices
2      I_e ← R-tree-indexing(Edges(p));
3      foreach e in Edges(p) do
4          e_ε ← shorten e by ε at both ends;
5          E_i ← find intersections of e_ε in I_e;
6          N ← N ∪ E_i;
7      end
8  end
9  foreach v in Vertices(shrunkPoly) do
10     if Index of v within a cluster n ∈ N then add only-once intersection from n as
         a vertex to p';
11     else add v to p';
12     update indexing of m accordingly;
13 end
```

5.2 Region Fitting

The sole interest in this process arose because the Grid rule requires a quad region as an input, otherwise, we cannot reliably construct a grid layout. A problem can occur when the Grid rule is applied twice or more times consecutively, which can be caused by the clipping to a custom junction or the iregion boundary. Hence, given an arbitrary polygon, we need to perform **shape** or **region fitting** to match or fit it to a quad with minimal distortion for a given use case. In our implementation the axiom region will always require to be fitted by a user.

Such fitting may be performed by *wrapping another shape* around iregion, a rectangle in our case, as discussed shortly in Section 5.2.1. For shapes that are 'almost-quad'-like

we can form a fitting polygon from *logical edge endpoints* of the iregion with *detected indices* (Sec. 5.1.3). In special case of nested Grid productions, the rule is able to create a 'shortcut' by using *unclipped quad region* for such purposes (Sec. 5.2.2).

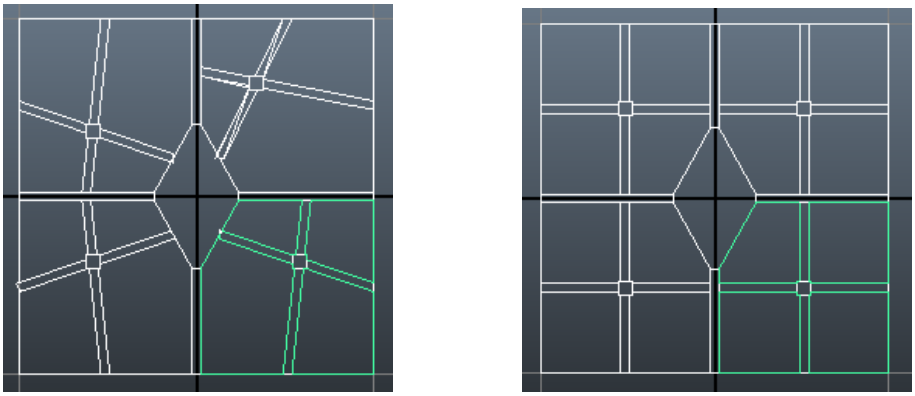


Figure 5.10: Recursive Grid application, left: without hint shapes, may result in non-quad intermediate quarter shapes; right: works as expected with hint shapes

5.2.1 Bounding Boxes based Fitting

City blocks as well as parks often resemble a rectangle. Moreover, paths of the grid-like parks usually meet at the right angles. We were motivated by the work of Vanegas et al. [VKW+12] who used the minimal oriented rectangle to find split-lines in recursive partitioning of a region, and some affinity to the Grid splitting using paths strips can be found. We tried using Axis Aligned Bounding Box (AABB) and Oriented Bounding Box (OBB) to enclose the entire input polygon. AABB is the smallest rectangle containing the input that is aligned to x and y coordinate axes. It is trivially computed using the minimum and the maximum of the shape coordinates in the Euclidian plane. The OBB is obtained by rotating the input polygon and looking for the minimal AABB. The implementation that we used, of the rotating callipers algorithm, probes each edge of the convex hull of the shape. It was first described by Shamos [Sha78]. The OBB is the minimal area rectangle that contains the entire input polygon. The AABB does not work so well for diagonally aligned shapes, while the OBB is independent of the shape orientation and provides better fitting.

We introduce the **box corners** method, which casts rays from the centre of a *bounding box* to each of the corners, and the resulting *intersections with the shape edges* make up the fitting quad vertices. We tried box corners fitting with both AABB and OBB.

5.2.2 Hint Shapes

Besides using the Bounding Box as a reference of paths construction in a Grid production, even a more appropriate way to fix this would be to add a **hint shape** quad, which is the *quarter region*, before the rule clips it against the neighbouring junctions and the

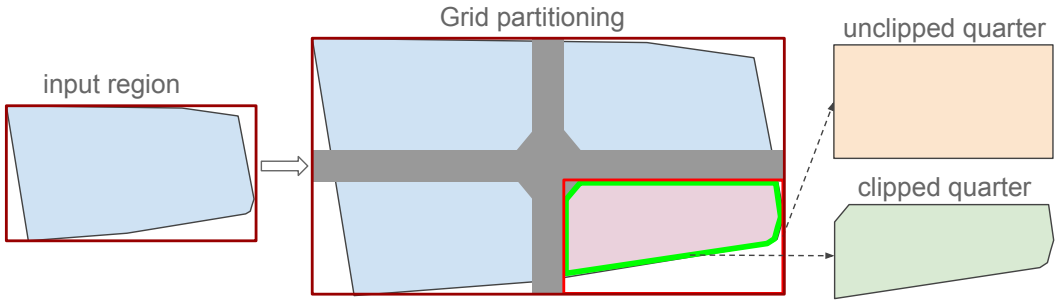


Figure 5.11: The *unclipped quarter* in the example of a Grid production is used as the hint shape.

iregion during the same production step (see Figure 5.11). Figure 5.10 shows an example generated with the same rule-set with and without the hint shapes.

Generally, this operation is rule-specific, and a given rule is trusted to do the best decision with all the necessary information at hand. Internally, a rule, if possible, should provide a hint shape, which should be the best fit according to the rule's judgement, since it is aware how the region is distorted. This is preferable to other region fitting methods, which may or may not produce desired results since they act *without the knowledge of the production context*. The process of adding hint shapes can be thought of as a higher level "semantic indexing". The hint shape is stored as a child shape within the given region.

Hint shapes were only employed for the Grid rule and are not propagated by any other rules, in other words, it is currently only useful for nested Grid productions. Unaware of the prior production history, the Grid rule searches for the quad to be used as the reference for path construction, in the following order: a) hint shape, if available b) logical edges, if $k = 4$ (which coincides with the physical polygon when $n = k = 4$) c) Oriented minimum Bounding Box (OBB). Further work is required to explore the usefulness of hint shapes for other rules.

5.3 Selection

The goal of selection is to specify a subset of the boundary to be passed to a rule. Selection can be of two types – we either want to select continuous intervals or discrete values. For instance, the Peel rule operates on continuous selection, while if we want to insert objects along the boundary, we use the discrete type. Selection of both types can be specified in various units.

5.3.1 Selection of Ranges

A polygon boundary indexed with the logical edges can be mapped to the interval $[0, k]$. For example, $[0, 1]$ refers to the first logical edge. The continuous unit of selection is

called a **range**, $[a, b] \in \mathbb{R}^2$, where $a < b$, $I_{min} \leq a$ and $b \leq I_{max}$. The master range, $[I_{min}, I_{max}]$, is dependent on the selection mode.

Every selection mode is always evaluated into the physical range $[I_{min}, I_{max}] = [0, n]$ internally. Physical ranges, however, cannot be supplied by the user because they are too volatile with respect to the changes in geometry. Doing so would also cause confusion of whether to address the boundary by the length or the number of elements.

Numerical Selection

A range in the plain numerical form references the logical edge extent (Sec. 5.1.1). For example, the value $[0.5, 1.5]$ selects a continuous range that includes the second half of LE_0 and the first half of LE_1 (see Figure 5.12).

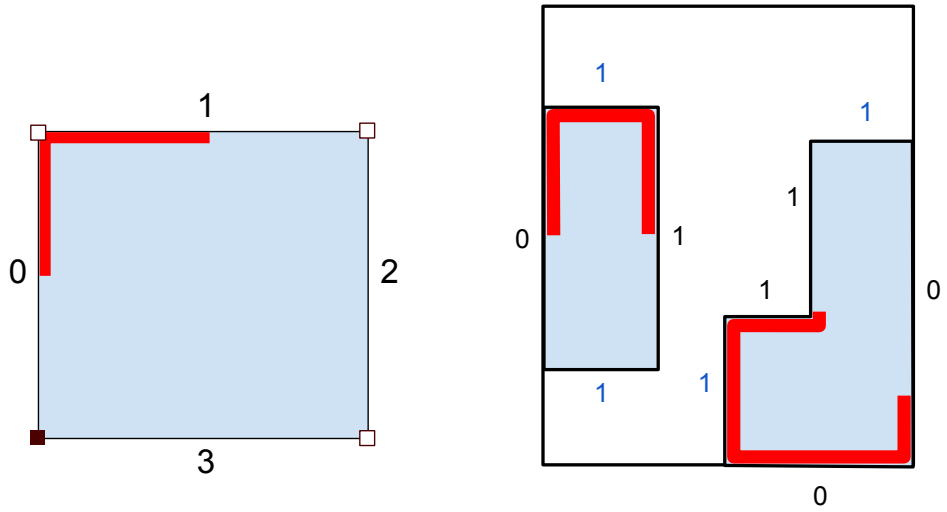


Figure 5.12: Numerical selection of LE-labelled polygons on the range [0.5..1.5].

Edge Orientation Selection

The **edge orientation** selection mode looks for a sequence of edges that are oriented within the angles specified in the master range $[\theta_{min}, \theta_{max}]$ in degrees. Angle applies only to the most local transformation context, in other words the history of rotations is taken into the account. A naïve version of the algorithm simply returns an edge list that meets the normal orientation condition (see Figure 5.13 left). However, the goal is to obtain a single *continuous* edge range. Although a growing algorithm has been tested, trying out on a number of shapes it turned out that *complement* of the largest continuous range *not* in the selection produces the desired results (see Figure 5.13 right).

5.3.2 Selection of Samples

Discrete selection is another name for directed boundary sampling.

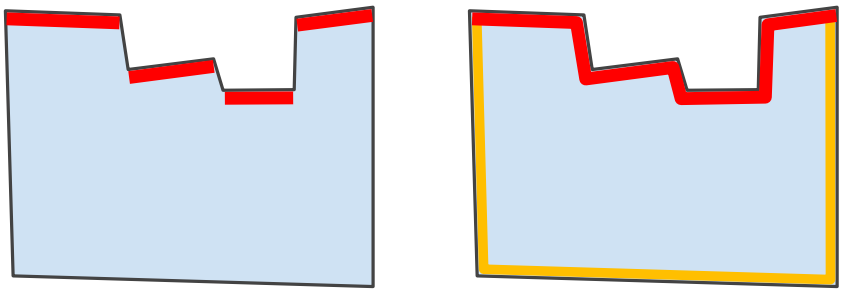


Figure 5.13: Edge orientation selection. Left: the set of initially selected edges (red) is often discontinuous. Right: a continuous range (red polyline) is computed by looking for the largest continuous *un-selected* range (amber polyline) and then subtracting from the boundary set.

Boundary Locations

Boundary locations are used to place one or multiple objects along the *subset* of the boundary represented by a polyline. It is specified as an expression that is evaluated as to list of locations along the given polyline within the range $[0, 1]$. An expression can be the following:

1. A list of real values within the range $[0, 1]$
2. A distribution of line-samples, specified by *start* and, optionally, *end* offsets, and frequency of items. The frequency of items is given by the distance between them or the number of items.
3. A vertex-counting value. For instance $[1, 3]$ would select the second and the fourth vertex. It is also possible to specify a position between the vertices, where $[0.8, 1.2]$ refers to two positions $1/5$ of an edge length away from the second vertex in the two of its adjacent edges.

Shape Placement Discrete Selection

As the name suggests, this kind of selection is used by the Place rule. Shapes placement occurs along a subset of the boundary, which we parametrise to the interval $[0, 1]$. Positions can be either explicitly given, e.g. [0.1, 0.5, 0.9], or encoded using a 'line placed' formula. The formula consists of the start offset, distribution parameters and, optionally, the end offset. If no end offset is specified, it is equal to the start offset. To clarify, the *end offset* is counted from the *end* of the range. Distribution is specified with either the number of samples or the distance between the samples. The distance is given in the parametric space of the context within the range $[0, 1]$. The distance is the desired distance, since we cannot set the exact distance, unless the length is a multiple of it. The distance is used to compute the number of samples, we fit as many samples as we

can, plus if the remainder is greater than $d/2$, we add another sample, making the actual distance smaller. Alternatively, if it is less than $d/2$ nothing is added making the actual distance slightly larger than desired.

Entrance Placement Discrete Selection

The Rays rule uses a slightly different discrete selection syntax to specify the boundary points that are used as positions for the park entrances. Samples are simply specified as a list of parameters. Parameters can either be mapped to the whole length of the boundary, or correspond to the physical indexing. The latter is the only exception to the rule of referencing the actual geometry, because in the Rays rule it is often desirable to specify vertex positions directly, especially with respect to the initial axiom shape.

CHAPTER 6

Placement

Figure 6.1: Plants cover the ground region completely [DHL+98].

The look of the rendered park is made up out of a composition of the objects placed in the final regions (those that correspond to the terminal shapes). Most of the visible entities are vegetation, including trees, grasses, and plants of other sizes and varieties. In a city park, it is also common to find artificial objects like benches, lamp-posts and so on. When emulating a real-life park, the vegetation should cover the regions completely, as demonstrated in the Figure 6.1, since the topsoil layer is usually not visible behind the dense cover of grasses. Because sampling of individual grass threads greatly impedes the performance of both the generation and the visualisation (though tile-based sampling would improve performance a bit), we limit to an illustrative representation of the lawn made of a dark-green coloured region.

To supplement the layout generation process, the core functionality of placement consists of uniform scattering of samples within a 2D region. We tried jittered grid-based sampling (see Figure 6.2), but ended up employing the Poisson distribution (Sec. 3.4), due the more 'natural' appearance of its samples, using the Bridson method [Bri07]. In

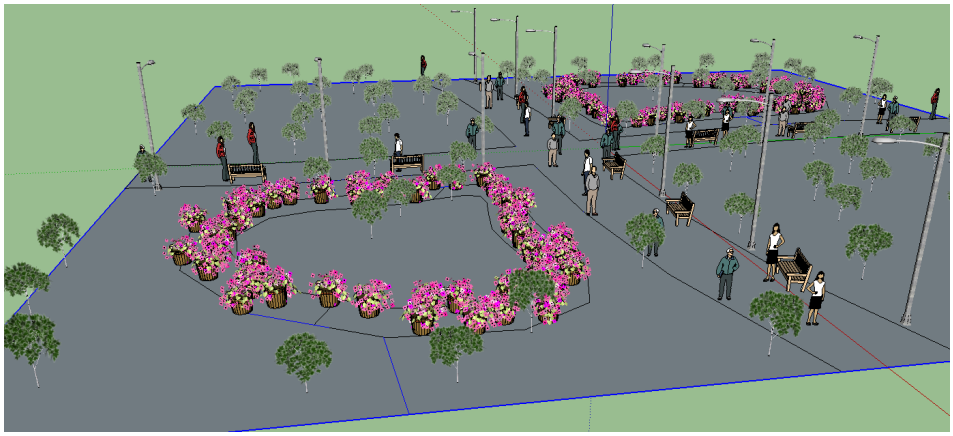


Figure 6.2: Experimentation with object placement using non-procedural methods. The park layout is inspired by the SimCity Large Park (see Figure 2.4 right).

addition, since having a tree in the middle of a region is such a frequent detail in parks (see Figure 2.9), we included an option for placing an object in the centre, corresponding to the centre of mass, a centroid, of the polygon. Since indexing of paths has not been pursued, except for the centre placement, we omit placement within the paths.

6.1 Scatter Rule

Placement of objects was implemented in the **Scatter** rule, which accepts either a keyword `CENTRE` or the distribution parameters. These parameters can either include a **distribution radius** (as described in Section 3.4) or a desired **number of samples** to be distributed, followed by an optional **offset** from the border. Offset is the same as the radius, if it is not given.

To compute the samples, first, the offset shape is computed using the offset parameter t, Figure 6.3. Then, the AABB rectangle is obtained where the samples are actually placed into. Finally, the samples outside of the offset polygon are clipped.

6.1.1 Guessing Radius from the Number of Samples

As mentioned in Section 3.4, the radius specifies the minimum distance between any two samples. Very often, a user wishes to distribute a fixed number of plants within the region. We are currently not aware of a solution that would reliably deliver a distribution radius given the number of samples. Our solution results in rather sparse distributions, which uses the formula $\frac{1}{2}\sqrt{w \cdot h / N_s}$ for getting the radius given a number of samples N_s and the dimensions (width and height) of the AABB. In other words, the radius corresponds to the circle that is inscribed in the square that has the average sample area of the AABB region. A better method would take into account the areas in the corners of such square that is not occupied by the circle.

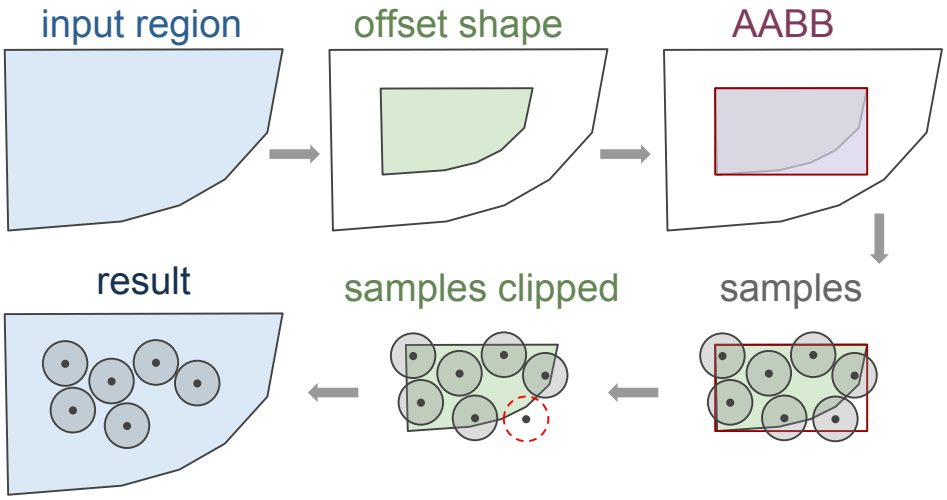


Figure 6.3: Process of sample generation using the Scatter rule.

When symmetry is used (see Section 4.4.4) for junction sampling in Cells and Rays rules, the sampling is performed within the subregions and is simply mirrored.

CHAPTER 7

Implementation

The implementation consists of three parts which can loosely be connected to the MVC pattern [KP88]: The SketchUp UI or the *View*, the main *Controller* Ruby script and a Layout Generator loader, the functionality of which can be attributed to a *Model*. The View and the Controller form the front-end, and the LG library – the back-end. The task of the front-end is to accept the user input and to forward it to the back-end. The axiom region is specified by selecting the face using the host application viewpoint UI. The C++ codebase uses Boost libraries, which are largely header-based, resulting in fewer link dependencies. Using Boost versions of STL containers for data structures helped to solve the incompatibilities between different versions of compiler toolsets.

7.1 Layout Generation

The Layout Generator, i.e. the back-end, is written in C++ and built as a shared library, designed to be independently loaded by any client (i.e. the calling code). The library accepts input geometry, a rule file, supplemented by a list of additional attributes. Park synthesis consists of three steps: rule file *parsing*, optionally the parameter or *attribute update*, and *generation*. Furthermore, the latter involves a combination of *indexing*, *partitioning* and *sampling* sub-steps. During the parsing step, rules (Sec. 7.1.1) and attributes (Sec. 7.1.2) are loaded from a text file into the two separate data structures. If an error occurs at this step, the generation may not proceed, and the client has to report this error to the user (for instance as a UI message) so that the rule file may be corrected and re-parsed. The generation step uses the rule data structure to perform productions, referencing the attributes in the process. Upon termination, the generator returns a list of attributed labelled shapes and sample positions. Once the rule file is parsed, attribute update and generation (with an updated set of parameters) steps can occur an arbitrary number of times. For instance, the interleaved calls to these two operations would be required to allow an interactive visualisation – if one wanted to change the path width in real time, for instance.

7.1.1 Rule Parsing

We have simplified the CGA grammar considerably in order to derive our shape grammar, for instance, we allow only a single rule application in a statement.

Implementation of a recursive descent parser in the Boost *Spirit* library was used for reading in the rules from the textual form. Using C++ operator overloading, Spirit allows to closely follow the EBNF (Extended Backus-Naur Form) syntax in code.

Since a rule statement in the rule file maps to a single rule application, such is stored as an **operation** object. The rule data structure is a dictionary of operations, indexed by a string – the match label. The **flat_map** implementation of a hash table from the Boost *Container* library was used as a dictionary, which is a variant of the *unordered_map* of STL. From the performance point of view, the hash table has many advantages over the tree-based implementations, when cache-based memory architectures are taken into the account. Besides the technical aspects of the modern hardware architectures (which is outside of the scope of this text), the dictionary has an advantage of having on average constant lookup time. Since insertion is conducted only once – during parsing, the lookup time occurring during production has a decisive factor in performance.

7.1.2 Attributes

An Attribute can be evaluated to a **numeric** or a **string** value. Similarly to rules, attributes are also stored in a hash table, which is justified by high access and update frequency in comparison to insertion. It is implemented as a hierarchical tagged union type based on the Boost **variant**, which allows to easily add modifiers and attach additional sub-types. Using modifiers, a *numeric* value can be tagged as **relative** and **stochastic**. A stochastic value consists of a **distribution** type modifier and two subtypes of a *numeric* type, **lower**, and **upper**, denoting the *distribution range*. The *string* value allows to reference another attribute by storing its name, which results in the final value resolving at runtime.

A *relative* attribute stores 100% as the value 1, but values of over 100% are possible. The purpose of a relative attribute is to scale distances and regions with relation to, in most cases, the axiom region by using the 1/4 of its perimeter as the **magnitude**, or the 100% value. In an ideal case we wish to use a *side length* of a square shape as the magnitude during the design, or, alternatively the *mean quad edge length*. Our presumption is that the user will avoid supplying a 'winding' boundary polygon with a high *perimeter*/*area* ratio, to avoid a possible content upscaling.

7.1.3 Geometric Operations and Partitioning

Boost *Geometry* library is employed for the 2D Boolean operations. Although the library by default largely uses the rational number arithmetic, numerical errors for some operations are common, for instance, the Boolean *union* on touching polygons produces invalid results. We use ϵ-areas, for instance extending the relevant subset of the boundary, to mitigate the numerical errors.

Partitioning into Regions

While working on the partitioning system, we observed that algebraic operations often do not directly translate into the implementation, and we give a brief account on this. Given two regions a and b, in the *implementation context* we write the *union* of them as $a + b$, the *difference* as $a - b$. We used brackets to enforce the order of the operations applied, for instance in $a + (b - c)$ difference occurs before the union.

For all three structural region partitioning methods (Grid, Cells, and Rays) the input region is subdivided into three types of regions: quarters, path junctions and path sections, as already described in Section 2.3. We discovered that for the Grid and Cells rules, in order to avoid numerical geometric errors, the optimal order of operations is when the path sections and junctions are computed first. The quarters ($\mathbf{R_q}$) are subsequently created by subtracting the paths (junctions $\mathbf{R_j}$, sections $\mathbf{R_s}$) from the iregion ($\mathbf{R_i}$):

$$\mathbf{R_q} = \mathbf{R_i} - (\mathbf{R_j} + \mathbf{R_s})$$

For the Cells rule, since quarters (that are derived from cells) are initially known, and the path sections, as opposed to the junctions, are rather trivially computed – we have attempted the following order of operations:

$$\mathbf{R_j} = \mathbf{R_i} - (\mathbf{R_q} + \mathbf{R_s})$$

However, the above stated and other attempted arrangements of boolean operations with the Boost Geometry library result in non-manifold complex polygons either as an intermediate product or the end result, and this makes the result is unusable in both cases.

Starting with the version 1.5.8, Boost Geometry supports *multipolygons*, allowing polygons to be grouped for operations. For instance, a union of paths in Grid layout can be achieved with a union of two groups containing vertical and horizontal strips. Before that, we used the incremental path union – starting by combining two perpendicular path regions nearest to a boundary (for example the leftmost horizontal and uppermost vertical) and proceeding with other regions away from the boundary. We also know that a more efficient solution recursively merges polygon pairs is possible, but multipolygon-capable operations alleviate the need for that.

Grid Partitioning

A hole will occur when a custom junction ($\mathbf{R_{jc}}$) does not contain the default junction ($\mathbf{R_{jd}}$) entirely, $\mathbf{R_{jd}} - \mathbf{R_{jc}} \neq \varnothing$, during a Grid rule production (Sec. 4.3). To avoid these holes custom and default junctions are *union*ed, $\mathbf{R'_j} = \mathbf{R_{jc}} + \mathbf{R_{jd}}$. Moreover, to avoid neighbour overlaps in custom junction, each one is limited by a quad $\mathbf{R_{jmax}}$ that extends slightly less than the halfway to the next junction, $\mathbf{R''_j} = \mathbf{R'_j} - \mathbf{R_{jmax}}$.

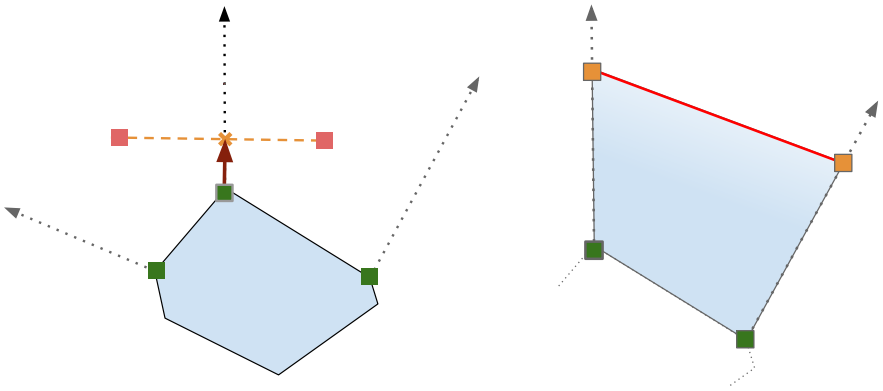


Figure 7.1: Polygon creation from half-open cells. Left: half-open edge construction. Right: terminating edge (red) created between the vertices created by clipping of the two half-open edges of the same cell.

Cells Partitioning

Boost Polygon library is used to generate Voronoi diagrams (VDs). The library uses the *half-edge* [McG] data structure to store geometry data, with vertices oriented Counter-Clockwise (CCW), so conversion and mapping to CW orientation of Boost Geometry have been performed. We take advantage of the linear access time to neighbouring edges and vertices in the half-edge based container during the creation of path regions.

It is important to note that outer cells are *half-open* since a Voronoi diagram spans the whole plane and a single such cell has two *half-infinite edges – rays*, each of which has two cells that share it. Moreover, the *direction* of the ray is not given, however, a library example uses direction as the vector from the *source vertex* to the *midpoint* of the line segment spanned by the centres [Syd12] of the the ray's cells (see Figure 7.1 left). Half-open regions are converted into a polygon by cutting the ray. The endpoints of clipped rays are joined with a 'terminating' edge (see Figure 7.1 right).

Cell shrinking and, optionally, smoothing results in two opposite or twin edges moving away from each other and forming a path section. Each pair of edges is only accessed once by marking the already visited edges. Similarly, a path junction is formed by the vertices that are originally at the same position within the neighbouring cells are that are subsequently shifted away from each other in a shrunk and/or smoothed region.

Rays Partitioning

The R-tree data structure is used to find the nearest junction for each entrance. Path sections can intersect near junctions for a sharp enough angle and we create a set of disjoint clipped path sections around a junction, as described in Section 4.5.4.

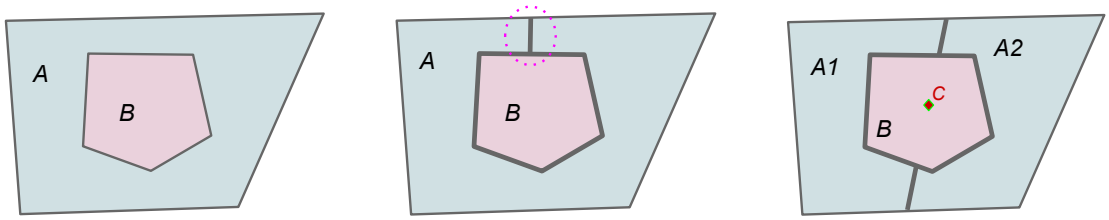


Figure 7.2: Polygon placement. Left: B creates a hole within A, middle: A is cut from the insertion point to the boundary, right: A is cut in half into $A1$ and $A2$ and B is inserted along the cut.

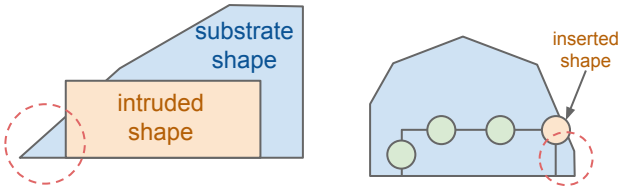


Figure 7.3: A derived shape may split the iregion which could be: left: An Intruded shape, right: or an Inserted shape.

7.1.4 Polygon Placement

Placement of a polygon within another, a usually larger polygon is implemented in Insert and Place rules. Overlapping two polygons means a certain area of the park is occupied twice, which may result in overlapping of placed objects. This must be avoided.

Subtracting one from another, as shown in the Figure 7.2 left, when B is entirely surrounded by A, is possible when polygons with holes are permitted. We decided to avoid holes since this considerably complicates all of the algorithms, including the object placement.

As a result, A has to be cut to allow insertion of B. Making a cut half the way through A and then subtracting B, Figure 7.2 middle, would have been possible, but the Geometry library lacks the support for weakly simple polygons, moreover, because such a polygon is not *injective*, it cannot be used in algorithms employing ϵ-areas.

A better solution, exemplified with a Garden Scene (see Figure 2.3), is to split A into two sub-polygons, $A1$ and $A2$, and then subtract the B from both of them, Figure 7.2 right. This insertion method requires that B touches the boundary of both $A1$ and $A2$. This precondition is guaranteed if the split line of A goes through the centre of B, point C (and centre of B is in B, which we assume to be in most cases). The Place rule satisfies this, since B is inserted at the split line. In the case of the Insert rule, the problem does not occur, since instead of the split line we use the boundary. The case where polygons are concave may result in a polygon split (Sec. 7.1.5).

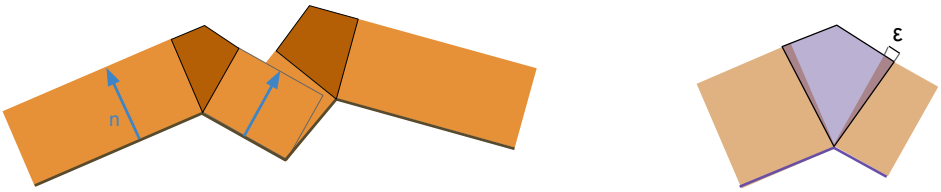


Figure 7.4: Polygon insetting. Left: individually intruded edges (orange) connected junction polygons (brown). Right: ϵ-extension of a junction polygon.

7.1.5 Special Cases for Region Placement

Normally, the Peel and Place rules should produce only a single substrate shape. However, an intruded (Figure 7.3 left) or inserted (Figure 7.3 right) region can fragment the iregion, when it *touches the boundary* at another location other than where it is inserted. In such circumstance we can either *clip* the shape – allowing the fragmentation to occur, or to *discard* the shape, terminating the production for the local branch of the 'offending' shape.

7.1.6 Border Subset Intrusion

Boundary subset intrusion is creation of the polygon between a subset of a boundary, represented by a polyline, and the inset (offset with a negative t, see also Section 4.4.2) instance of it. It is generated by intruding each edge of this polyline and then joining them. Joining polygons are inserted at angles greater or equal to 180° (see Figure 7.4 left). Joining polygons are extended by ϵ-value along the inset boundary (see Figure 7.4 right).

A *logical edge index tracking* version of intrude was implemented. However, this was not used in the final implementation since 01-indexing requires only to discern between the boundary and non-boundary cases (Sec. 5.1.2).

7.1.7 Polygon Fixing

Extra vertices may appear in the results of a geometry operation. These are either an 'unessential' vertex that lies within an ϵ-distance to the edge spanned by its neighbours (the violet vertex in the Figure 5.1 middle), or a vertex within an ϵ-distance to another vertex. These are often caused by numerical errors. Since such rather inconsiderable changes to geometry do not alter the look of the layout, they are discarded in a post-processing step after a rule production.

7.2 SketchUp Integration

Trimble SketchUp is a lightweight 3D modelling application that is popular with, among others, architects, landscape architects, engineers, and interior designers. We appreciated

the simplicity of SketchUp from the point of view of a novice user, for instance when compared to Maya. The decision to work with this application was determined by a presence of a preconfigured and extensible library of *components* or model prefabs, and *materials*, also because of a lightweight scripting interface.

SketchUp allows augmenting its functionality with **extensions** or plugins, written in the Ruby programming language. The **Park Generator** SketchUp plugin has been created, which loads the layout generator shared library using the Ruby C API. The plugin gets the input from the user and forwards it to the layout generator library. This includes the input region, the rule file and a number of additional attributes that a user can set in the front-end window UI. A JavaScript callback is used to update values in the Ruby interpreter. After generation, materials are assigned and then intrusion (push/pull) is performed, as described below in Section 4.8. A SketchUp *attribute dictionary* (different to the Ruby language dictionary) is used to store shape attributes. The final step of generation involves instancing of SketchUp components from the list of samples.

A user first selects a rule file, and once it is parsed successfully, the generate option becomes available. When syntax errors are encountered, an error message is displayed and the user has to edit the rule file and click on the "update" button. To pass the input region, the user selects a previously created polygon face in the SketchUp viewport and presses the "generate" button in the UI Window to allow for production to start. When completed, the park model(s) are subsequently displayed in the same viewport. The original input region is deleted to avoid the overlap of surfaces. The 'undo' operation can be used to revert the entire park generation task, replacing the park model with the original park region.

7.2.1 Shape Fitting Plugin

Shape fitting methods described in Section 5.2 were implemented in a separate SketchUp plugin. The advantage of a separate plugin for region fitting is in being able to manually tweak the fitted shape to suit user's needs before the park generation step is invoked. This is possible since the fitted shape is linked with the source shape bidirectionally by tracking the SketchUp `entityID` in an attribute dictionary.

CHAPTER 8

Results

8.1 Layouts

Generation of layouts was the major goal of this work and in this section we look at layouts produced by Grid, Cells and Rays rules, since they define the design of the park.

8.1.1 Grid Parks

Regular partitioning of the Grid layout is straightforward to implement and delivers believable park models (see Figure 8.1). The Grid rule scales really well (see Figure 8.2). The rule requires a quad as either an input or a hint shape, which can be distorted (see Figure 8.3) but it loses to some degree its 'regular' quality.

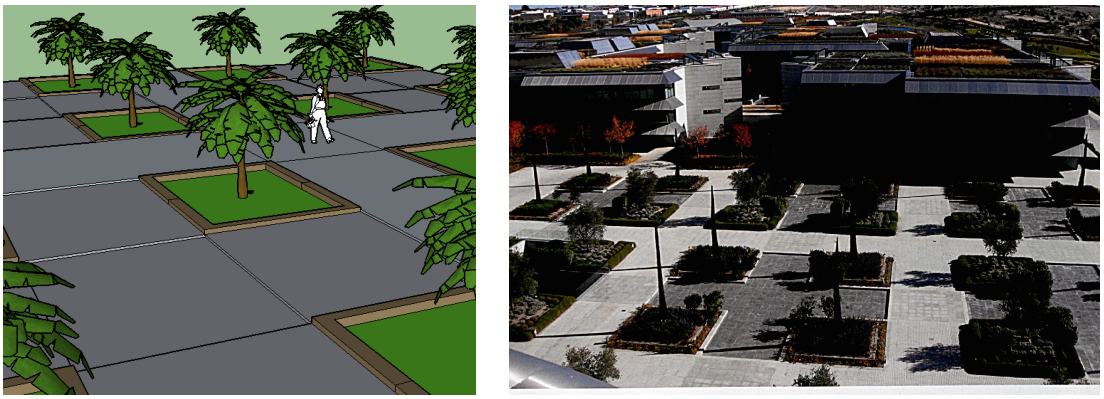


Figure 8.1: Grid Layout variant 1 (Appendix B.1) – comparison of the generated park with the photograph of the real-life park that has inspired it.

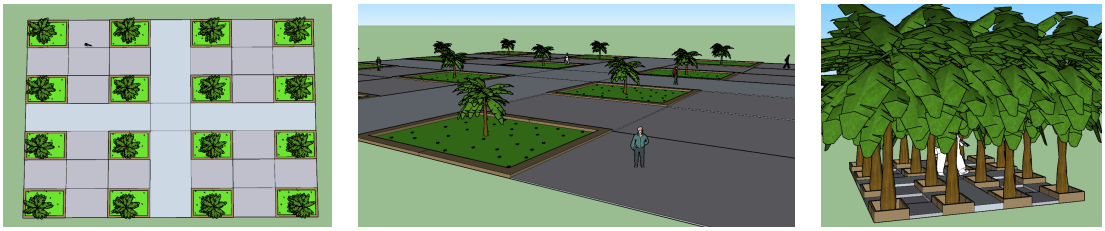


Figure 8.2: Various axiom region sizes applied to Grid Park variant 1. Left: the default size is depicted on the Figure 8.1 left, compare to a large very region (middle) and a very small (right) region.

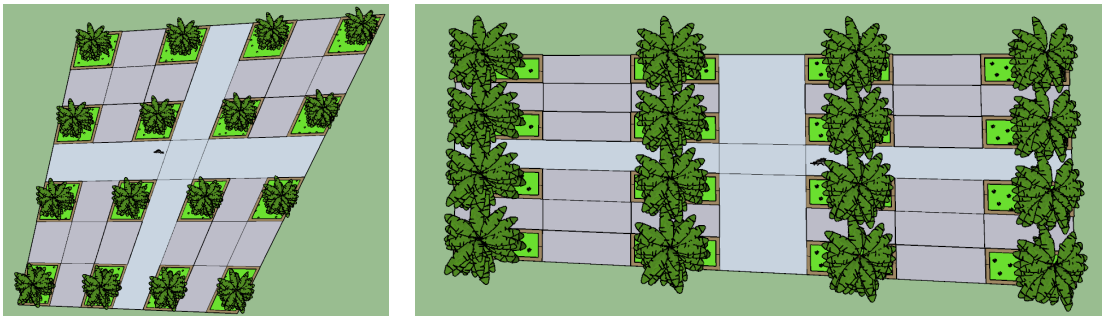


Figure 8.3: Grid Park variant 1: distortion applied to the quad axiom region.

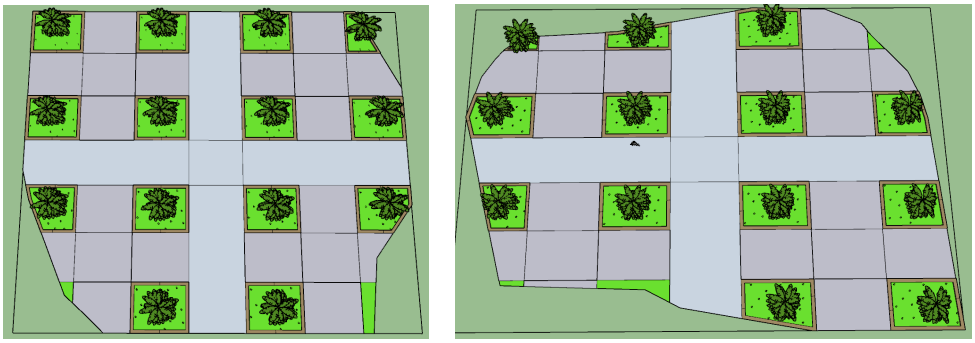


Figure 8.4: Grid Park variant 1: results with two variants of more complex axiom regions.

Fitting Methods

Distortion to the input quad region is directly reflected in the output (see Figure 8.3). Results with various shape fitting methods, implemented in the Shape Fitting plugin, can be seen in Figure 8.7. A generic threshold-based method (Sec. 5.1.3) fails to consistently produce a rectangle-looking quad (with corners close to 90 degrees), and even then, distortions (see Figure 8.7e) make it an unsuitable method when working with Grid designs. Box-based methods are more suitable for fitting Grid layouts. OBB (see Figure 8.5a) is able to fit the content slightly better than AABB (see Figure 8.5b), but

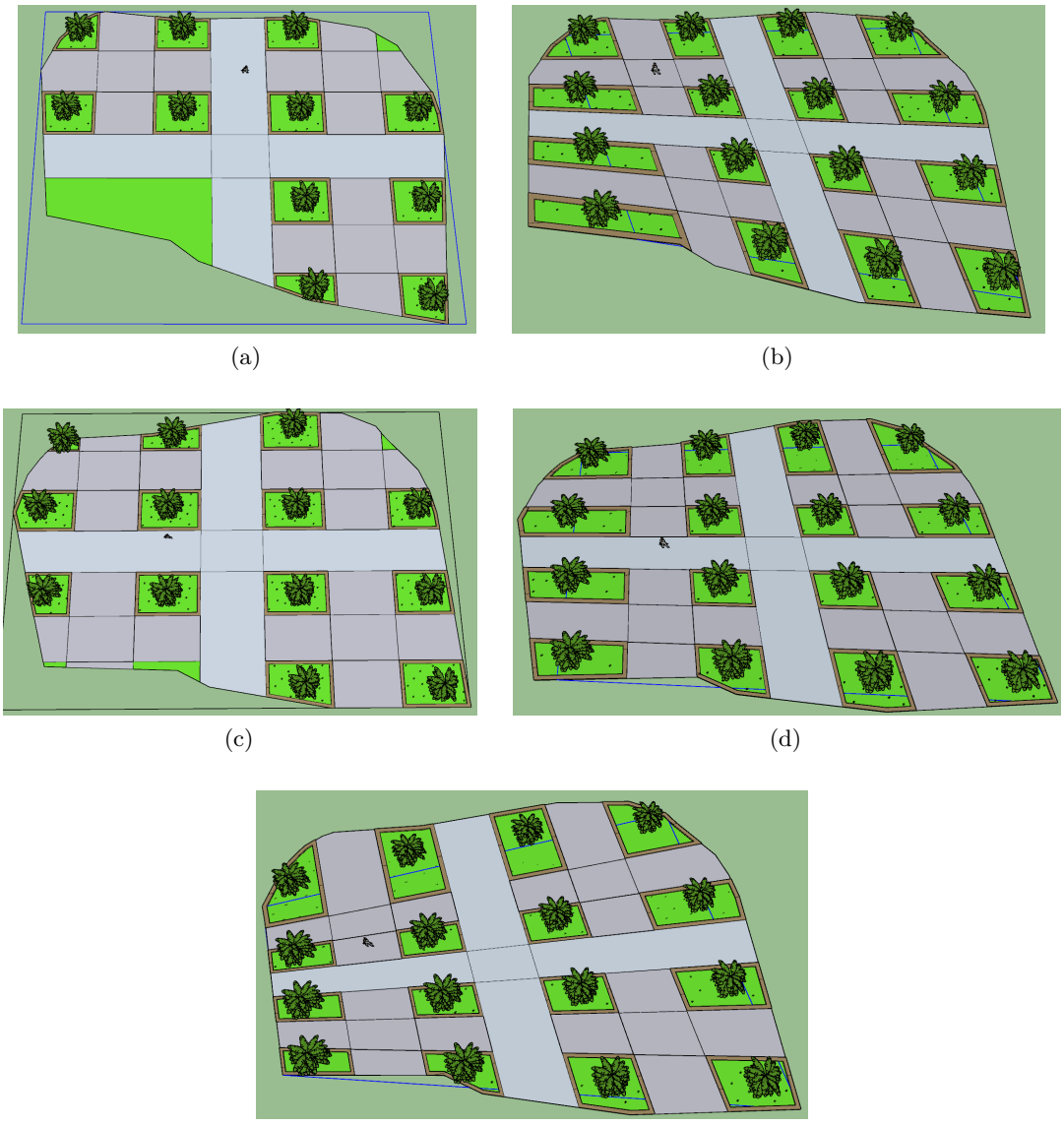


Figure 8.5: Fitting methods applied on the Grid Park example1: (a) Axis Aligned Bounding Box (AABB), (b) Axis Aligned Corner fitting, (c) Oriented Bounding Box (OBB), (d) Oriented Corners fitting, (e) Angle Thresholding.

both result in more consistent looking layouts over other methods, since they do not produce distortions, even if large areas within a Box fall outside of the iregion. The Corners methods still produce some distortion. Since the OBB Corners method (see Figure 8.5d) produces the least distortion while fitting the most content, this method could potentially be used for more flexible Grid designs.

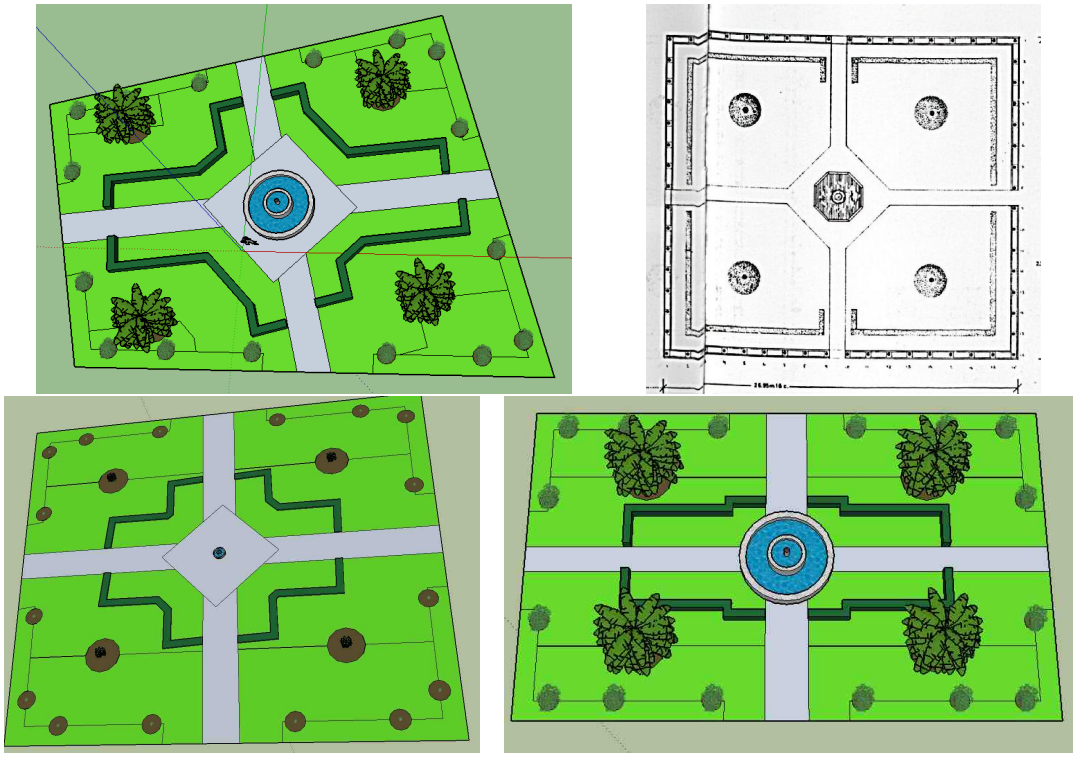


Figure 8.6: Grid Layout variant 2 (Appendix B.2), the model and the layout of the real-life park that has inspired it.

Issues

Besides perceptual issues, higher distortion in fitted shapes translates to 'parallel' path sections getting oriented at greater angles to each other, which leads to intersections between section 'tails' outside of the iregion (Sec. 4.3), which sabotages the production.

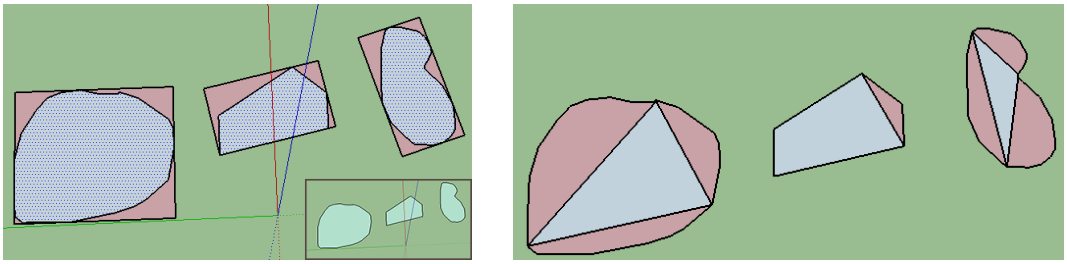


Figure 8.7: Shape Fitting plugin results: OBB (left) and Angle Thresholding (right).

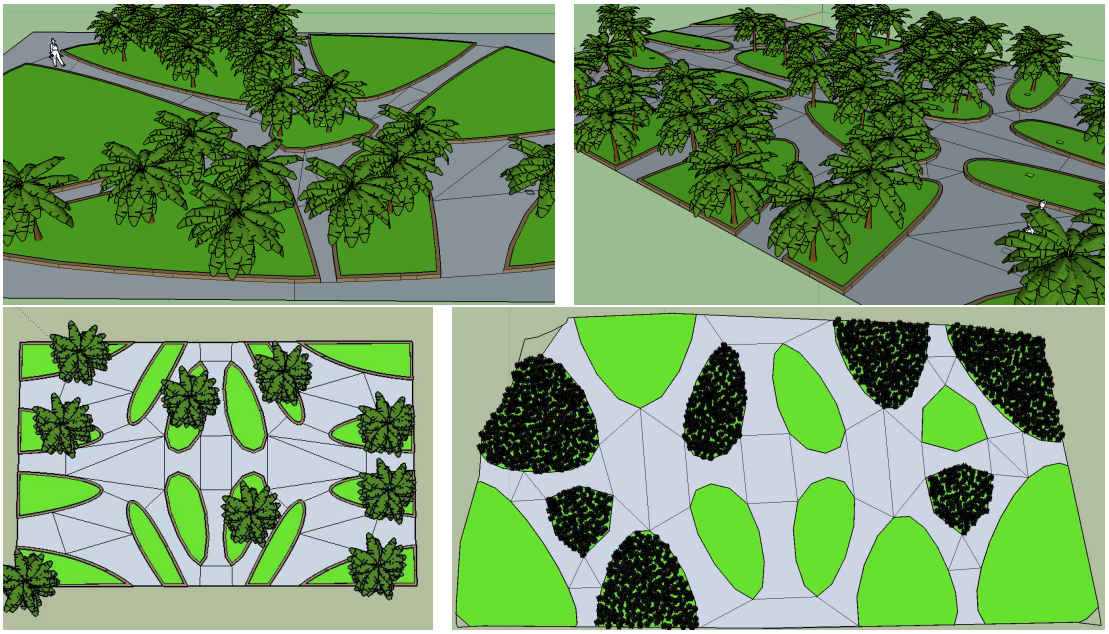


Figure 8.8: Cells results. Top left: variant 1 (Appendix B.3), ithin a smoothed sub-region . Top right and bottom: variant 2 (Appendix B.4), the use of symmetry in cell samples in regions of various sizes.

8.1.2 Cells Parks

Modern parks featuring curved regions vary greatly, yet Cells manages to capture a subset of real-life free-form park designs (see Figure 8.8). For example, concave regions, as observed in the CityEngine garden scene (see Figure 2.3), are not possible and regions may appear somewhat blob-like. Nevertheless, we have shown a novel way to construct parks using the procedural techniques, which also allows to incorporate symmetry (see Figure 8.8).

Issues

Samples placed further than a given distance from the iregion corner had resulted in a hole (see Figure 8.8 bottom right). Paths may become too small and the quarters can be too close to each other (consider the paths shown in the Figure 8.8 left). This results from extending and smoothing cells at the boundary to avoid holes and blob-like appearance. This could be avoided with improved boundary clipping and smoothing methods, merging of quarters or larger path sizes.

A *number of samples* can be specified instead of the *sample radius*, although this does result in a sparse sample distribution, and may result in odd-looking parks. Sampling issues also concern Rays parks.

8.1.3 Rays Parks

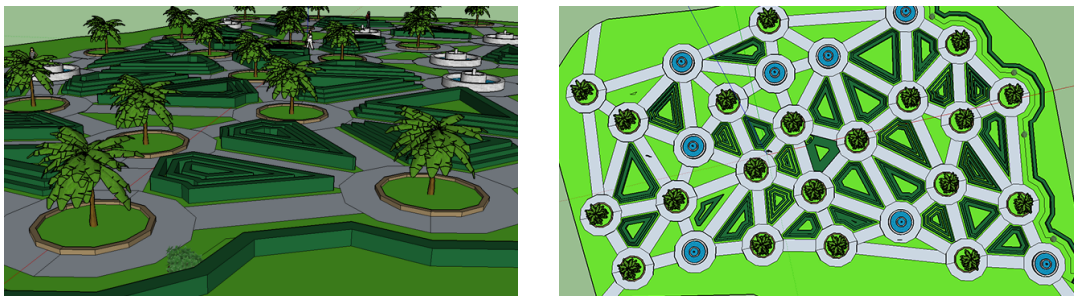


Figure 8.9: Rays results (Appendix B.5), the side view (left) and the top view showing the path structure (right).

Rays rule works the best for one or two junctions. Two issues have been encountered with Rays. Firstly, the best method for specifying entrance location was sought, for instance, it becomes problematic to target park corners with a high number of vertices on the boundary. The second issue is connecting junctions in a manner that would resemble real-life parks. The Delaunay method produces a planar graph, but with a large number of junctions, the 'artificial' triangular look becomes rather prominent (see Figure 8.9 right). Random removal of edges (see Section 4.5.3) as a quick solution produces better results (for instance, see Figure 4.8b). However, these are still not sufficiently realistic. Finally, a problem of maximal distribution for a given number of samples is also shared with the Cells rule.

8.1.4 Scattering

Sample-based object scattering works well for trees. For smaller objects, like grass plants, the algorithm is a performance bottleneck. This could be improved by resorting to a completely random scattering, ignoring the collisions, or, stitching of tiles with the precomputed patches of samples.

8.2 Performance

We have tested three stages of synthesis in SketchUp 2016 – *parsing*, *generation* and *instancing*. The timings are available on the Table 8.1. The test machine had an AMD A8-5600K APU 3600 Mhz CPU with 4 Logical Processors, observed to be running at around 3800 Mhz during the timing process, and 8GB of system memory clocked at about 1866 Mhz.

Parsing includes the generator object creation and destruction, and reading in the textual rulefile into a data structure. It can be seen that the parsing stage is very efficient, which mostly relies on the external Spirit library. Nonetheless, the performance gain could be attributed to lower cache pressure because of the smaller code size. Longer times for the Rays park is reflected in the rulefile size.

The generation stage corresponds to the actual production process, which takes almost two orders of magnitude longer and includes all layout geometry processing. It can be noticed that the Grid variant 1 (Appendix B.1) and the Rays examples are much slower for a larger region. This is because sampling is used to distribute the grass objects in the Grid park and also since the number of junctions in the Rays park is not fixed, but dependent on the radius, meaning a larger region size will create more junctions and hence more content. Here a quadratic increase in timing can be observed between the smaller and the larger regions, and since the area is 16 times greater, more content needs to be generated. In comparison, the Cells variant 2 (Appendix B.2) uses a fixed number of cells, and the smaller difference in times can be attributed to the tree placement. A generally higher timing for the generation step, as exemplified by the larger Rays park, shows that our implementation has some potential for optimisation.

The SketchUp instancing stage includes the creation of SketchUp shapes, material assignment, volume extruding and instancing of objects. It is considerably slower, partly because a script is used for execution instead of native code. This stage also accounts for the overhead of interfacing with SketchUp and instancing of more complex geometric objects. We did not time the actual rendering time.

Rule	Parsing	Generation		Instancing	
		small	large	small	large
Grid 1	0.19	12.98	72.29	926.68	1937.45
Grid 1 (no grass)	0.20	10.73	10.52	598.91	619.74
Grid 2	0.29	14.10	14.15	364.54	328.29
Cells 2	0.18	19.06	32.07	749.95	984.63
Rays	0.65	10.31	182.09	213.51	2049.20

Table 8.1: Timing of park generation using four different rules given in miliseconds. Rulefile for Grid park variant 1 was tested with (the second row) and without grass samples. Two square testing regions were used, one sized 500 (small) and another 2000 (large) units.

8.3 Comparison to Other Work

CityEngine [PM01], [MWH$^+$06] is only able of modelling *grid*-like parks without the custom junctions. This, however, requires two steps for *split*-based partitioning along x and y. Moreover, the *extended L-system* could be modified to perform productions for both *grid*-like and *ray*-like designs. On the other hand, CityEngine and CGA lack support for curved regions. Besides Krecklau [KPK10], we did not find any procedural modelling methods dealing with curved regions, and even then, the operations occur at the terminal shapes. Modelling of parks requires creation of regions with curved boundaries early on within the derivation.

Ulmer's work on VegeZones [Ulm05], concentrated on object placement as the main element of park design assisted with a single partitioning operation based on an edge split. Our method, on the other hand, focuses on partitioning supplemented with an elementary placement functionality. Also, selection in VegeZones is based on physical geometry, while our indexing methods are independent of underlying geometry, for instance, capable of handling curves in the discrete form. Comparing to Guerrero's use of features in placement [GJWW14], we have a more limited set of features, constrained only to the position along the boundary and the distance from the boundary. We are not able to mix multiple placements within the same polygon unless we modify the production system to allow it. Likewise, in contrast to the CGA, our system is incapable of multiple region productions within the same context, or within the same space.

CHAPTER 9

Conclusion

In the beginning, we have put forth the following goals for this thesis:

1. Create a grammar system for park layout generation.
2. Distribute of objects within the individual regions.
3. Fitting of a quad to a region of higher complexity,

We have covered all of these topics, as described in the previous chapters. However, further work is required to either complete our implementation for commercial exploitation.

9.1 Summary

In this work, we have developed a CGA-like textual grammar system for park generation, based on the three observed park layout patterns.

Layout generation has been the primary focus of this work. We have analysed a number of parks and abstracted an overall look of each park into one of the three patterns for partitioning of regions – *Grid*, *Cells* and *Rays*. Based on these rules, our system allows structural partitioning into Grid-like layouts with custom junctions by a single production, layouts consisting of regions with curved boundaries and layouts with regions partitioned by ray-like path arrangements. We have introduced region insetting along an arbitrary length around the region boundary, the functionality that was also used for insertion of new regions within an existing one. To support this method, we have derived a novel edge labelling system called *01-indexing*, based on the concept of edge groups into so-called *logical edge*s. In order to provide a framework for placing paths within a Grid-like layout, we looked into *region fitting* methods, for instance, using a minimal rectangle to wrap around the input region or, otherwise, unclipped production artefacts as *hint shapes*.

We also looked at object placement, which we used to complement the layout generation to construct parks. For example, we can evenly distribute vegetation within a region or place a tree at a region's centre.

9.2 The Focus of the Tasks

Following CityEngine, we have chosen to use textual rules as input. However, we are aware that recently there was an interest in graph-based representation in both research ([SMBC13]) and commercial environments (SideFX Houdini [Hou15]). Our system could be modified to accommodate any such alternative rule representation with minimal effort. We have chosen not to extend CGA, but rather to create a CGA-influenced novel simpler grammar, that targets generation of parks.

We resorted to using simple polygons, even though employing *polygons with holes* would remove the necessity to split the target shape when inserting a new shape. This representation, however, leads to problems when considering inner 'hole' regions in a multitude of geometry algorithms. Likewise, not to overcomplicate our methods, we did not use parametric boundaries, like curves, in our regions, instead utilising polylines abstracted in logical edges.

We have implemented our work around the SketchUp application, which allows us to easily supply the input, assign materials, instantiate park objects, elevate regions and to display the results. In addition to the main Layout Generator plugin, we have also made available plugins for region fitting and object placement, which could be used as separate steps when necessary.

Side issues, like interactive editing, rendering of vegetation, optimisation of our derivation system and multiprocessing were left out since these topics have been explored by others already. In practical terms, it would have been more beneficial to focus only on the implementation of the park partitioning patterns and leave methods like sampling, placement, rule productions, polygon smoothing to other external subsystems.

9.2.1 Incomplete Features

Implementation for some parts of our system remains incomplete. The 01-indexing has not been extended to paths, meaning we cannot reliably place objects on them. Instead of using a single timestamp-based seed for all random number generation, the implementation should allow controlling a collection of seed values mapped to the input regions, like in CityEngine. Junctions in the Rays rule are currently limited to circles, and the Cells rule are left with the default junctions. For the Rays rule, indexing detection should be employed for the park corner selection by using 'logical vertices' instead of the actual vertices.

9.3 Future Work

9.3.1 Grammar

Our grammar could benefit from a number of the CGA features. Conditional operators would allow emulating loops by terminating the recursive calls, thus reducing the number of redundant statements, created by copying and pasting, (consider an example in Appendix B.5). Arithmetic operators would further enhance the expressions.

9.3.2 Layout Generation

Some designs could be achieved by adding the curved paths functionality for the Grid and Rays layouts (such as shown in Figure 2.4). Better methods for connection of junctions with inner sections would allow the creation of more realistic looking Rays layouts. Although we used soothed Voronoi cells, generation of the quarters for the Cells rule is open to any other methods. Moreover, extending our method by using other smoothing algorithms and combining of regions, would improve the expressiveness of the Cells rule.

9.3.3 Indexing

A better region fitting method would combine Bounding and Inscribed maximum rectangles. Perhaps complex polygons could be split into subpolygons, in the manner, a block is split into lots in CityEngine, where a set of smaller neighbouring parks could be generated. Use of *hint shape*s should be considered other rules besides Grid.

9.3.4 Object Placement

It should be possible to generate a denser nearly maximal distribution given the number of samples, instead of the radius.

APPENDIX A

Grammar Reference

A.1 Parameter Types

A.1.1 Numeric Type

An expression of the numeric type can be used to store either a *floating point* value, a *real random range* or an *integer random range* (Sec. A.1.2). This expression allows representation of both *real* and *integer* numbers within the same type, and, most importantly, permits stochastic evaluation within both the real and the integer ranges.

A.1.2 Random Range

Real random range evaluates to a randomly selected real number from the $[min\_value, max\_value]$ range at evaluation time using the uniform distribution.

```
ru(min_value..max_value)
```

Integer random range evaluates to a randomly selected integer number from the $[min\_value, max\_value]$ range at evaluation time using the uniform distribution. Can be thought of as throwing a dice within the given range.

```
ri(min_value..max_value)
```

A.2 Partitioning Rules

A.2.1 Grid

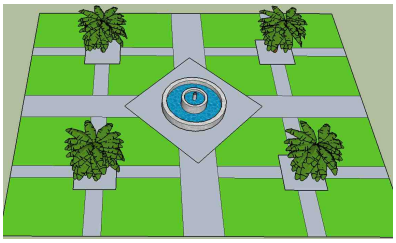
Partition an input region into quarters using a regular grid.

```
grid(idx_x, idx_y)
 { QuarterSel } { SectionSel } { JunctionSel }
```

- **`idx_x`**, **`idx_y`** (integer)
  The number of cell subdivisions along x and y axes. For instance, *grid(2, ri(1..3))* can produce cell arrangements 2×1, 2×2, or 2×3.

- **`QuarterSel`**, **`SectionSel`**, **`JunctionSel`** (sellabel)
  Label selector statements for the *quarter*, *path section* and *path junction* regions accordingly. In the Grid rule (only) the cell index can be used to assign a label to a particular grid quarter based on the quarter position in the grid.

Example



```
Park --> grid(2, 2)
 { Quarter1 }
 { ParkPath }
 { i("builtin:rhombus", %22) ParkCentreLarge }
Quarter1 --> grid(2, 2)
 { Quarter2 }
 { ParkPath }
 { i("builtin:square", %14) ParkCentreSmall }
ParkCentreLarge --> scatter(CENTRE)
 {  ParkCentreLarge_wfountain : "Fountain1Marble" }
ParkCentreSmall --> scatter(CENTRE)
 { ParkCentreSmall_wtree: "3dast_banana_tree" }
```

A.2.2 Cells

Partition input region into more natural looking "cell-like" quarters that resemble biological cells. **Voronoi Diagram** partitioning method is used.

Sample radius determines centre of a cell:

```
cells(Rradius)
 { QuarterSel } { SectionSel } { JunctionSel }
```

Alternatively, a number of samples can be requested, and best matching radius will be guessed (currently not optimal):

```
cells(number_of_samples)
 { QuarterSel } { SectionSel } { JunctionSel }
```

With an optional smoothing parameter:

```
cells(..., M(smoothing_factor))
 { QuarterSel } { SectionSel } { JunctionSel }
```

Symmetry parameter is added at the end and is also optional:

```
cells(..., S(symmetry_id))
 { QuarterSel } { SectionSel } { JunctionSel }
```

- **`radius`** (real)
  The minimum distance between any two path junctions; larger radius means fewer junctions. Alternatively, `number_of_samples` can be used:

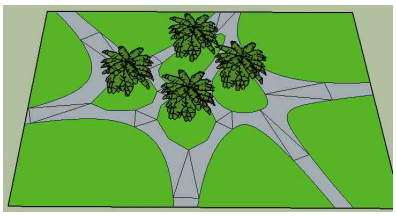
- **`number_of_samples`** (integer)
  Specify an actual number of junctions to place within the region. The algorithm tries to guess the most appropriate radius. However, the actual radius is usually less than the ideal one for the given number of samples would be.

- **`smoothing_factor`** (real)
  How much should the cells be smoothed. Smoothing reduces cell size.

- **`symmetry_id`** (integer)
  Valid values are 0: no symmetry 1: x symmetry 2: y symmetry 3: x and y symmetry (which is also bitwise `AND` of 1 and 2).

- **`QuarterSel`**, **`SectionSel`**, **`JunctionSel`** (sellabel)
  Label selector statements, see the Grid rule (Sec. A.2.1).

Example



```
path_width = %3
Park --> cells(R110, M(0.5))
 { border: QuarterOuter | else: QuarterInner }
 { PathSection }
 { PathJunction }
QuarterInner --> scatter(CENTRE)
 { QuarterInner_wtree: "3dast_banana_tree" }
```

A.2.3 Rays

Partition input region using straight path segments in the manner of 'rays'. Delaunay Triangulation is used (related to Voronoi Diagram, which is its dual).

Entrances (boundary intersections) are determined by the list of boundary parameters or selections. Junction placement (ray intersections) is determined by sample distribution radius:

```
rays(boundary_selections, Rradius)
 { QuarterSel } { SectionSel } { JunctionSel }
```

Alternatively, a number of samples can be specified, and the algorithm tries to guess the sample radius:

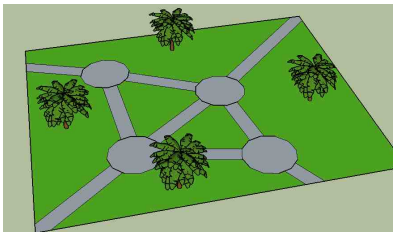
```
rays(boundary_selections, number_of_samples)
 { QuarterSel } { SectionSel } { JunctionSel }
```

Optionally, minimal offset from junctions to the boundary can be provided:

```
rays(boundary_selections, ..., offset)
 { QuarterSel } { SectionSel } { JunctionSel }
```

- **`boundary_selections`** (list)
  List of parameter values within the range [0..1]. For example, $[0, 0.25, 0.5, 0.75]$ will place entrances at the corners of a square.
- **`radius`** (real), **`number_of_samples`** (integer)
  Same as for the Cells rule (Sec. A.2.2) with the difference that here the junctions are placed instead of the quarters.
- **`offset`** (real)
  The minimum distance from the boundary for a junction centre.
- **`QuarterSel`**, **`SectionSel`**, **`JunctionSel`** (sellabel)
  Label selector statements, see the Grid rule (Sec. A.2.1).

Example



```
path_width = %5
rays_inner_graph_thinout = 0
Park --> rays([0, 0.2, 0.5, 0.8], R230, 75)
```

```
 { border: QuarterOuter | else: QuarterInner }
 { PathSection }
 { PathJunction }
QuarterOuter --> scatter(CENTRE)
 { QuarterInner_wtree: "3dast_banana_tree" }
```

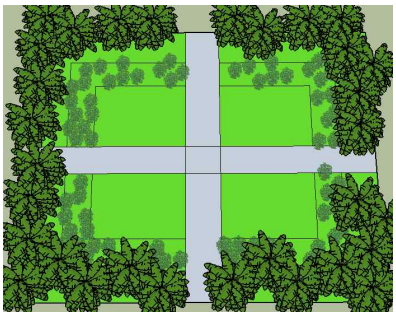
A.2.4 Peel

'Intrude' a subset of the boundary inwards of the input region.

```
peel(boundary_selection, offset)
 { IntrudedSel } { SubstrateSel }
```

- **`boundary_selection`** (selrange)
  A subset of the input region boundary, specified as one of the selection range types (Sec. A.6.1).
- **`offset`** (real)
  How much to peel or intrude.
- **`IntrudedSel`**, **`SubstrateSel`** (sellabel)
  Label selector statements (Sec.A.7.1) for the *intruded* and the *substrate* regions accordingly. The intruded region is the newly created region, and the substrate region occupies the remaining space of the input region.

Example



```
Park --> grid(2, 2)
 { Quarter }
 { ParkPath }
 { Junction }
Quarter --> peel([0, 1], %25)
 { IntrudedBoundary } { QuarterSubstrate }
QuarterSubstrate --> peel([1, 2], %25)
 { IntrudedInner } { QuarterSubstrate2 }
```

```
IntrudedBoundary --> scatter(R60)
 { IntrudedBoundary_wnettle: "3dast_banana_tree" }
IntrudedInner --> scatter(R30)
 { IntrudedBoundary_wnettle: "nettle_bush" }
```

A.3 Layout Boundary-based Placement Rules

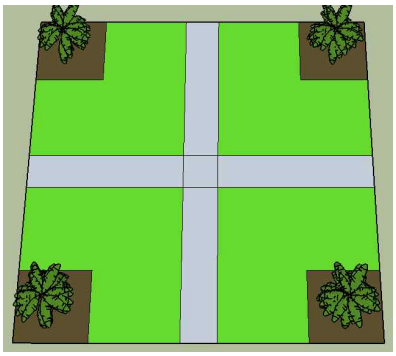
A.3.1 Insert

Insert a new shape at the boundary.

```
insert(value_selection)
 { InsertedShapeSel } { SubstrateSel }
```

- **`value_selection`** (float list)
  Insert locations parametrised along the input region boundary length.
- **`InsertedShapeSel`** (sellabel)
  Label selector statement (Sec.A.7.1) for the *inserted* shape. It contains the *insert* operator (Sec. A.7.4)that should supply the shape that is being inserted.
- **`SubstrateSel`** (sellabel)
  Label selector statement (Sec.A.7.1) for the and the remaining part of the input region.

Example



```
Park --> grid(2, 2)
 { Quarter }
 { ParkPath }
 { Junction }
Quarter --> insert([0.5])
 { i("builtin:square", %45) TreeRegion }
 { QuarterSubstrate }
```

```
TreeRegion --> scatter(CENTRE)
 { TreeRegion_wtree: "3dast_banana_tree" }
```

A.3.2 Place

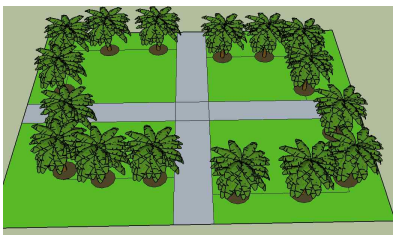
Inserts new shapes. Because new shapes cannot be placed in the middle of the polygon the input region needs to be the subdivided along the line where the new shapes are inserted. Placement occurs at an offset distance from the boundary or at the centre of a given region when single argument "CENTRE" is given.

```
place(boundary_selection, offset, locations)
 { IntrudedSel } { InsertedShapeSel }
 { SubstrateSel }

place(CENTRE)
 { IntrudedSel } { InsertedShapeSel }
```

When only a single argument "CENTRE" is given object is placed at the centre of the region.

- **`boundary_selection`** (selrange), **`offset`** (real)
  See the Peel rule (Sec. A.2.4).
- **`locations`** (sel_line_samples)
  Where to place inserted shapes. Either a list of floating numbers a repeat expression.
- **`InsertedShapeSel`** (sellabel)
  Specifies the new shapes, see the Insert rule (Sec. A.3.1).
- **`IntrudedSel`**, **`SubstrateSel`** (sellabel)
  Label selector statements for remaining shapes, see the Peel rule (Sec. A.2.4).

Example



```
Park --> grid(2, 2)
 { Quarter }
 { ParkPath }
 { Junction }
Quarter --> place([0, 1], ru(%20 .. %40), 0.1:n4:0.1)
```

```
 { IntrudedBoundary }
 { i("builtin:circle", 25) TreeRegion}
 { QuarterSubstrate }
TreeRegion --> scatter(CENTRE)
 { TreeRegion_wtree: "3dast_banana_tree" }
```

A.4 Object Placement Rules

A.4.1 Scatter

Scatters objects within the input region.

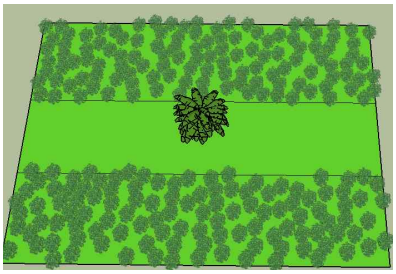
```
scatter(Rradius, offset)
 { RegionSel }

scatter(number_of_samples, offset)
 { RegionSel }
```

Place an object at the centre of the input region:

```
scatter(CENTRE)
 { RegionSel }
```

- **`radius`** (real), **`number_of_samples`** (integer)
  Same as for the Cells rule (Sec. A.2.2), but for the sample positions instead of the cell vertices.
- **`offset`** (real)
  Same as for the Rays rule (Sec. A.2.3), but for the sample positions instead of the junctions.
- **`RegionSel`** (sellabel)
  Label selector statement (Sec.A.7.1), allowing to re-write the input region label to avoid infinite recursive calls.

Example



```
Park --> select { set(type, "quarter") Quarter }
Quarter --> peel(ea[85, 95], %33)
 { Region1 }
 { Substrate }
Substrate --> peel(ea[85, 95], %33)
 { Region2 }
 { Region1 }
Region1 --> scatter(R30)
 { Region1_wnettle: "nettle_bush" }
Region2 --> scatter(CENTRE)
 { Region2_wtree: "3dast_banana_tree" }
```

A.5 Re-writing Rules

A.5.1 Select

Re-write incoming geometry to a different set of labels.

```
select { RegionSel }
```

- **`RegionSel`** (sellabel)
  Label selector list (Sec. A.7.1), rewriting the shapes to the new label(s) when evaluated.

No geometry is modified. The purpose of this rule is to remove redundancy of repeating the same operations on one label in different selector blocks. Rule application makes even more sense when selector expression lists are longer, or when the same label is assigned in a number of different rules.

Example

```
InputLabel --> place(CENTRE)
 { 50%: Border1 | else: Border2  }
 { i("circle", 10) Inserted }
 { 50%: Border1 | else: Border2  }
```

can be simplified to:

```
InputLabel --> place(CENTRE)
 { Border } { i("circle", 10) Inserted } { Border }

Border --> select { 50%: Border1 | else: Border2 }
```

A.6 Selection

A.6.1 Ranged Selection

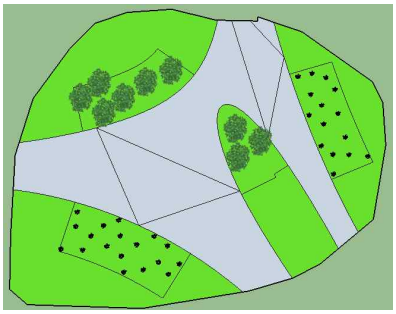
Ranged selection references a subset of the boundary parametrised by either *logical edge*s or a set of edges matching a given orientation (*edge orientation*).

```
[lower, upper]
```

- **`lower`**, **`upper`** – Selection range. Values can be specified as a double, percent, Random Range or a named attribute.
  - *Numeric selection*, corresponding to the logical edge range.
  - *Edge orientation selection*, corresponding to a range of angles, in degrees, resulting in a selection of a continuous set of edges bounded by these angles.

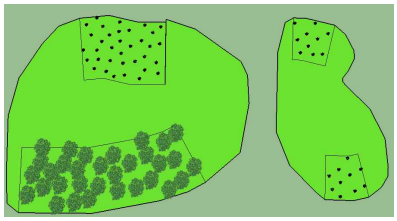
Examples

Numeric selection:



```
extend_corners = 0
Park --> cells(R110, M(0.5))
 { border: QuarterOuter | else: QuarterInner }
 { PathSection }
 { PathJunction }
QuarterOuter --> peel([1.25, 1.75], %35)
 { 50%: PlantRegion1 | else: PlantRegion2 }
 { GrassRegion }
PlantRegion1 --> scatter(R30)
 { Region1_wnettle: "nettle_bush" }
 PlantRegion2 --> scatter(R20)
 { Region2_wtree: "3dast_grass00" }
```

Edge Orientation selection:



```
extend_corners = 0
Park --> select { set(type, "quarter") Quarter }
Quarter --> peel(ea[70, 110] ea[-120, -60], %35)
 { 50%: PlantRegion1 | else: PlantRegion2 }
 { GrassRegion }
PlantRegion1 --> scatter(R30)
 { Region1_wnettle: "nettle_bush" }
 PlantRegion2 --> scatter(R20)
 { Region2_wtree: "3dast_grass00" }
```

A.7 Selectors

A.7.1 Label Selector Statement

```
{ LabelExpr1 | LabelExpr2 | ... LabelExprN }
```

`LabelExpr1`, **`LabelExpr2`**, ...**`LabelExprN`**
Label expressions (Sec. A.7.2). For each produced shape one of the label expression is chosen and evaluated to a label that the current shape is assigned. Selection criteria are determined by *match conditions* (Sec. A.7.3). More than one may match, but only the first matching one is selected. The last selector (`LabelExprN`) is expected to be a plain/default selector – otherwise, the rule application may fail.

A.7.2 Label Expression

Evaluates to a label when the condition matches.

A plain selector expression is simply a label, and it **always matches**:

```
Label
```

Alternatively, label selection may be limited to a particular condition:

```
MatchCondition: Label
```

Additional attribute modifications or shape insertion can be requested before the label:

```
MatchCondition: OperatorExpression Label
```

- **`MatchCondition`**
  Match condition (Sec. A.7.3).

- **`Label`** (label)
  The label that is assigned to a derived shape when the *match condition* (Sec. A.7.3) evaluates to `true`.

- **`OperatorExpression`**
  Optional selector operator expression (Sec. A.7.4).

A.7.3 Match Conditions

Border Keyword

Regions that are touching a boundary at an edge are selected (however, regions sharing a vertex with iregion boundary are not!).

```
border
```

Index Match

```
idx(x_expr, y_expr)
```

- **`x_expr`**, **`y_expr`**
  Evaluates to Grid index. Optionally prefixed with "!" which negates the meaning. Can be either:

  - **`idx`** (integer)
    A single integer index.
  - **`odd`** (string)
    Evaluates to a range of all odd index values. Even indices are evaluated with "!odd".

Example. Checkerboard pattern:

```
{ idx(odd, odd): LabelWhite |
  idx(!odd, !odd): LabelWhite | LabelBlack }
```

Stochastic Match

```
percent %
```

A selector is chosen with a probability of percent value. For instance "50%" would accept the label roughly half of the time.
Example. Select *Label2* half of the time, and *Label1* and *Label3* equally quarter of the time each:

```
{ 25%: Label1 | 50%: Label2 | else: Label3 }
```

Default Match Keyword

```
else
```

Default match condition always matches (the keyword it can also be omitted).

A.7.4 Selector Operators

Insert Operator

Insert operator creates new geometry. Depending on the rule it is either ignored (e.g. in the Peel rule), optional (junction selector statement of the Grid rule) or required (the Insert and Place rules).

```
i(geometry_path)

i(geometry_path, r1)

i(geometry_path, r1, r2)
```

- **r1**, **r2** (real)
  The first and the second parameters to the geometry.

- **geometry_path** (quoted_string)
  A string that specifies the new geometry to be inserted, which can be either:

  - **"builtin:<primitive_shape>"** – a predefined shape based on primitives where **<primitive_shape>** is either *circle*, *square* or *rhombus*. `r1` is the size of the shape. If `r2` is not specified, `r1` is used in its place; in such a case, the circle becomes an ellipse, and the square – a rectangle.
  - **"iregion:shrink"** – iregion is shrunk by `r1` and used as the inserted shape.
  - **"iregion:smoothen"** – iregion is shrunk by `r1` (see above) and then smoothed with `r2` weight; if `r1` is not given value 0.5 is used.

Examples. Insert a circle with the radius 15% of the input region (provided rule accepts shape insertion) into a shape labelled InsertedCircle:

```
{ i("builtin:circle", %15) InsertedCircle }
```

And rhombus 20 units wide and 15 units high:

```
i("builtin:rhombus", 20, 15)
```

Set Operator

Creates a new *attribute* or changes a *propagated attribute* that is inherited from the rule input shape. For instance, the "type" attribute is assigned using this operator.

```
set(attribute_name, attribute_value)
```

- **`attribute_name`** (string)
  The attribute name.
- **`attribute_value`** (real, integer, string, quoted_string)
  The attribute value. When the string is *not quoted*, another attribute is referenced using this string value as the name.

Example. Change path width to a third for any further productions on *QuarterLabel*:

```
{ set(path_width, %33.3) QuarterLabel }
```

APPENDIX B

Examples

B.1 Grid Park Variant 1

```
path_width = %14

Park --> grid(2, 2)
        { set(path_width_even, 1) InnerPathsRegion }
        { set(elevation, 4) PathOuter }
        { set(elevation, 4) Junction1 }

InnerPathsRegion --> grid(2, 2)
        { TreeBed }
        { set(variant, 2) set(elevation, 2) PathInner }
        { set(variant, 2) Junction2 }

TreeBed --> place(CENTRE)
        { set(type, "border") Boundary }
        { i("iregion:shrink", %6) set(type, "quarter") GrassQuarter }

GrassQuarter --> scatter(R30)    { TreeQuarter : "3dast_grass00" }
TreeQuarter --> scatter(CENTRE) { "3dast_banana_tree" }
```

B.2 Grid Park Variant 2

```
bush_width = %8
bush_offset = %20
centre_tree_bed_width = ru(%7 .. %14)
centre_r = ru(%10 .. %24)
```

```
Park --> grid(2, 2)
        { TopQuarter }
        { ParkPath }
        { i("builtin:rhombus", centre_r) ParkCentre }

TopQuarter --> place([0.1, 0.9], %10, [0.1,0.4,0.6,0.9])
        { PlantsQuarterBorder }
        { i("builtin:circle", %5) BoundaryPlantsRegion }
        { PlantsQuarterInner }

PlantsQuarterInner --> peel([0.3, 0.7], bush_offset)
        { InnerBushQuarter }
        { InnerGrassQuarter }

InnerBushQuarter --> peel([1, 2], bush_width)
        { set(type, "bushes") BushRegion }
        { CentreGrassRegion }

InnerGrassQuarter --> place(CENTRE)
        { TreeQuarter1 }
        { i("builtin:circle", centre_tree_bed_width)
                set(type, "loam") QuarterTreeRegion }
        { TreeQuarter2 }

// add objects
// fountain
ParkCentre --> scatter(CENTRE)
        {  ParkCentreWithFountain : "Fountain1Marble" }

// tree
QuarterTreeRegion --> scatter(CENTRE)
        { QuarterTreeRegionWithTree: "3dast_banana_tree"  }

// bush plants
BoundaryPlantsRegion --> scatter(CENTRE)
        {  BoundaryPlantsRegionWithPlants : "nettle_bush" }
```

B.3 Cells Park Variant 1

```
path_width = %3
grass_elevation = 5
border_width = 5
```

```
Park --> place(CENTRE)
        { Outer1}
        { i("iregion:smoothen", %3, 0.7) set(type, "quarter")
                SmoothCentre }
        { Outer1 }

Outer1 --> select { set(type, "path") PathOuter }

SmoothCentre --> cells(9, M(0.5), S(0))
        { QuarterRegion } { PathSection } { PathJunction }

QuarterRegion --> place(CENTRE)
        { Border1 }
        { i("iregion:shrink", border_width) set(type, "quarter")
                set(elevation, grass_elevation) QuarterRegion }
        { Border1 }

Border1 --> select { set(type, "border") BorderRegion }

QuarterRegion --> select { 50%: TreeRegion | GrassRegionEmtpy }

TreeRegion --> scatter(R80) {
        QuarterTreeRegionWithTree: "3dast_banana_tree"  }
```

B.4 Cells Park Variant 2

```
path_width = %3
grass_elevation = 5
border_width = 5

Park --> cells(9, M(0.5), S(3))
        { QuarterRegion }
        { PathSection }
        { PathJunction }

QuarterRegion --> place(CENTRE)
        { Border1 }
        { i("iregion:shrink", border_width)
                set(type, "quarter") set(elevation, grass_elevation)
                GrassRegion }
        { Border1 }
```

```
Border1 --> select
        { set(type, "border") QBorder }

GrassRegion --> select
        { 50%: TreeRegion | GrassRegionEmpty }

TreeRegion --> scatter(R80)
        { TreeRegionWithTree: "3dast_banana_tree"  }
```

B.5 Rays Park

```
clipping_mode = 1
path_width = 70
rays_inner_graph_thinout = 0
rays_remove_path_overlaps = 1
selection_internal_minimal = 1

Park --> rays(
        [0, 0.2, 0.5, 0.8]
        , R300
        , 125
        )
        { border: Quarter1 | else: QuarterInner }
        { Path1 }
        { Junction1 }

////////////////////////////////////////////////////////////////
// junctions
////////////////////////////////////////////////////////////////

// at a junction centre either a fountain or a tree
Junction1 --> select { 75%: JunctionTree | JunctionFountain }

JunctionFountain --> scatter(CENTRE) { jf: "Fountain1Marble" }

JunctionTree -->  place(CENTRE)
        { JunctionPath }
        { i("iregion:shrink", %25) JunctionTree1 }

JunctionTree1 --> place(CENTRE)
        { set(type, "border") JunctionPath1 }
        { i("iregion:shrink", %6) set(type, "quarter")
                set(elevation, 2) JunctionTreeQuarter }
```

```
JunctionTreeQuarter --> scatter(CENTRE) { jt: "3dast_banana_tree" }

////////////////////////////////////////////////////////////
// quarters
////////////////////////////////////////////////////////////

// outer (boundary-touching)
Quarter1 --> peel([1, 2], 80) {
        BushQOuter} { QuarterBoundary }

// outer boundary
QuarterBoundary --> place([0.1, 0.9], 80, [0.1,0.4,0.6,0.9])
        { Quarter2A  }
        { i("builtin:circle", 20) BoundaryPlantsRegion}
        { Quarter2B }

// outer plants
BoundaryPlantsRegion --> scatter(CENTRE) {
        BoundaryPlantsRegionWithPlants : "nettle_bush" }

BushQOuter--> peel([1, 2], 19)
        { set(type, "bushes") BushQ2Outer} {  Quarter3 }

Quarter3 --> place([1,2], 30, [0.1,0.4,0.6,0.9])
        { Quarter3A  }
        { i("builtin:circle", 16) BoundaryPlantsRegion}
        { Quarter3B }

// two variations for inner regions
QuarterInner --> select { 50%: QuarterInner1 | QuarterInner2 }

QuarterInner1   --> peel([0, 1], 50)
                    { BushQInner }  { Grass }
BushQInner      --> peel([0, 1], 30)
                    { BushQ2Inner } { set(elevation, 24) Bushes }
BushQ2Inner     --> peel([0, 1], 20)
                    { BushQ3Inner } { set(elevation, 16) Bushes }
BushQ3Inner     --> peel([0, 1], 10)    { Q4Inner }
                    { set(elevation, 8)     Bushes }
Q4Inner             --> select { Grass}

QuarterInner2 --> peel([0, 1], 60) { BushQ1Inner2 } { Grass }
```

```
BushQ1Inner2  --> peel([0, 1], 50) { BushQ2Inner2 } { Bushes }
BushQ2Inner2  --> peel([0, 1], 40) { BushQ3Inner2 } { Grass }
BushQ3Inner2  --> peel([0, 1], 30) { BushQ4Inner2 } { Bushes }
BushQ4Inner2  --> peel([0, 1], 20) { BushQ5Inner2 } { Grass }
BushQ5Inner2  --> peel([0, 1], 10) { BushQ6Inner2 } { Bushes }
BushQ6Inner2  --> select { Grass}

// materal attribute assignment
Grass --> select { set(type, "quarter") QuarterGrass}
Bushes --> select { set(type, "bushes") QuarterBushes}
```

Bibliography

[AA96] Oswin Aichholzer and Franz Aurenhammer. Straight skeletons for general polygonal figures in the plane. In *Proceedings of the Second Annual International Conference on Computing and Combinatorics*, COCOON '96, pages 117–126, London, UK, UK, 1996. Springer-Verlag.

[AVB08] Daniel G. Aliaga, Carlos A. Vanegas, and Bedřich Beneš. Interactive example-based urban layout synthesis. *ACM Trans. Graph.*, 27(5):160:1–160:10, December 2008.

[BCMRC93] Laurence Boxer, Chun-Shi Chang, Russ Miller, and Andrew Rau-Chaplin. Polygonal approximation by boundary reduction. *Pattern Recognition Letters*, 14(2):111 – 119, 1993.

[Bri07] Robert Bridson. Fast poisson disk sampling in arbitrary dimensions. In *ACM SIGGRAPH 2007 Sketches*, SIGGRAPH '07, New York, NY, USA, 2007. ACM.

[Cit15] Esri cityengine, 2015.

[Coo86] Robert L. Cook. Stochastic sampling in computer graphics. *ACM Trans. Graph.*, 5(1):51–72, January 1986.

[DHL+98] Oliver Deussen, Pat Hanrahan, Bernd Lintermann, Radomír Měch, Matt Pharr, and Przemyslaw Prusinkiewicz. Realistic modeling and rendering of plant ecosystems. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '98, pages 275–286, New York, NY, USA, 1998. ACM.

[FW00] Ken Fieldhouse and Jan Woudstra. *The Regeneration of Public Parks*. Taylor & Francis, first edition, 2000.

[Gam] Sir, you are being hunted. http://www.big-robot.com/2012/03/12/sir-you-are-being-hunted/. Accessed: 2016-12-20.

[GJWW14] Paul Guerrero, Stefan Jeschke, Michael Wimmer, and Peter Wonka. Edit propagation using geometric relationship functions. *ACM Transactions on Graphics*, 33(2):15:1–15:15, March 2014.

[GSMCO09] Ran Gal, Olga Sorkine, Niloy Mitra, and Daniel Cohen-Or. iWires: An analyze-and-edit approach to shape manipulation. *ACM Transactions on Graphics (proceedings of ACM SIGGRAPH)*, 28(3):33:1–33:10, 2009.

[Gut84] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. *SIGMOD Rec.*, 14(2):47–57, June 1984.

[Hav05] Sven Havemann. *Generative Mesh Modeling*. PhD thesis, Braunschweig University of Technology, Nov 2005.

[Hou15] Houdini, side effects software, 2015.

[IFPW10] Martin Ilčík, Stefan Fiedler, Werner Purgathofer, and Michael Wimmer. Procedural skeletons: Kinematic extensions to CGA-shape grammars. In *Proceedings of the Spring Conference on Computer Graphics 2010*, pages 177–184. Comenius University, Bratislava, May 2010.

[IMAW15] Martin Ilčík, Przemyslaw Musialski, Thomas Auzinger, and Michael Wimmer. Layer-based procedural design of facades. *Computer Graphics Forum*, 34(2):2015.

[KP88] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, August 1988.

[KPK10] Lars Krecklau, Darko Pavic, and Leif Kobbelt. Generalized use of non-terminal symbols for procedural modeling. *Computer Graphics Forum*, 29(8):2291–2303, 2010.

[LC88] Jia-Guu Leu and Limin Chen. Polygonal approximation of 2-D shapes through boundary merging. *Pattern Recognition Letters*, 7(4):231 – 238, 1988.

[LHP11] Luc Leblanc, Jocelyn Houle, and Pierre Poulin. Component-based modeling of complete buildings. In *Graphics Interface 2011*, pages 87–94, May 2011.

[LSWW11] Markus Lipp, Daniel Scherzer, Peter Wonka, and Michael Wimmer. Interactive modeling of city layouts using layers of procedural content. *Computer Graphics Forum (Proceedings EG 2011)*, 30(2):345–354, April 2011.

[LWW08] Markus Lipp, Peter Wonka, and Michael Wimmer. Interactive visual editing of grammars for procedural architecture. In *ACM SIGGRAPH 2008 Papers*, SIGGRAPH '08, pages 102:1–102:10, New York, NY, USA, 2008. ACM.

[McG] Max McGuire. The half-edge data structure. `http://www.flipcode.com/archives/The_Half-Edge_Data_Structure.shtml`. (Accessed on 07/01/2017).

[MSK10] Paul Merrell, Eric Schkufza, and Vladlen Koltun. Computer-generated residential building layouts. *ACM Trans. Graph.*, 29(6):181:1–181:12, December 2010.

[MSL+11] Paul Merrell, Eric Schkufza, Zeyang Li, Maneesh Agrawala, and Vladlen Koltun. Interactive furniture layout using interior design guidelines. *ACM Trans. Graph.*, 30(4):87:1–87:10, July 2011.

[MVLS14] Chongyang Ma, Nicholas Vining, Sylvain Lefebvre, and Alla Sheffer. Game level layout from design specification. In *Eurographics 2014*, pages 95–104, 2014.

[MWH+06] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. In *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, pages 614–623, New York, NY, USA, 2006. ACM.

[NGDGA+16] Gen Nishida, Ignacio Garcia-Dorado, Daniel G. Aliaga, Bedřich Beneš, and Adrien Bousseau. Interactive sketching of urban procedural models. *ACM Transactions on Graphics (SIGGRAPH Conference Proceedings)*, 2016.

[Pat12] Gustavo Patow. User-friendly graph editing for procedural modeling of buildings. *Computer Graphics and Applications, IEEE*, 32(2):66–75, March 2012.

[PH15] Peter Palfrader and Martin Held. Computing mitered offset curves based on straight skeletons. *Computer-Aided Design and Applications*, 12(4):414–424, 2015.

[PM01] Yoav I. H. Parish and Pascal Müller. Procedural modeling of cities. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 301–308, New York, NY, USA, 2001. ACM.

[Sha78] Michael Ian Shamos. *Computational geometry.* PhD thesis, Yale University, 1978.

[Sit13] Tom Sito. *Moving Innovation: A History of Computer Animation.* MIT Press, 2013.

[SM15] Michael Schwarz and Pascal Müller. Advanced procedural modeling of architecture. *ACM Transactions on Graphics*, 34(4 (Proceedings of SIGGRAPH 2015)):107:1–107:12, August 2015.

[SMBC13] Pedro Silva, Pascal Müller, Rafael Bidarra, and Antonio Coelho. Node-based shape grammar representation and editing. In *Proceedings of PCG 2013 - Workshop on Procedural Content Generation for Games*, Chania, Crete, Greece, may 2013.

[Syd12] Andrii Sydorchuk. Voronoi basic tutorial. `http://www.boost.org/doc/libs/1_63_0/libs/polygon/doc/voronoi_basic_tutorial.htm`, 2012. (Accessed on 07/01/2017).

[Ulm05] Andreas Ulmer. Modeling and rendering of plant ecosystems in urban environments. Master's thesis, ETH Zürich, 5 2005.

[VAW+10] Carlos A. Vanegas, Daniel G. Aliaga, Peter Wonka, Pascal Müller, Paul Waddell, and Benjamin Watson. Modeling the appearance and behaviour of urban spaces. *Computer Graphics Forum*, 29(1):25–42, 2010.

[VKW+12] Carlos A. Vanegas, Tom Kelly, Basil Weber, Jan Halatsch, Daniel G. Aliaga, and Pascal Müller. Procedural generation of parcels in urban modeling. *Comp. Graph. Forum*, 31(2pt3):681–690, May 2012.

[WWSR03] Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. Instant architecture. *ACM Trans. Graph.*, 22(3):669–677, July 2003.

[YWVW13] Yong-Liang Yang, Jun Wang, Etienne Vouga, and Peter Wonka. Urban pattern: Layout design by hierarchical domain splitting. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2013)*, 32:Article No. xx, 2013.