UNIVERSITÀ DEGLI STUDI
DI MILANO

# Procedural Content Generation

*Part 3*

*A.I. for Video Games*

Davie Gadia (gadia@di.unimi.it) and Dario Maggiorini (dario@di.unimi.it)

# Grammar based techniques

- Use of *formal grammars* to generate content
- Usually, complex graphical content composed by an arrangement of simpler modules
  - E.g.: plants, buildings, cities
- However, grammars can be applied also to non-graphical content, like to generate the different steps/actions of a mission

# Formal Grammars

- Formal Grammar: set of production rules for rewriting strings

- Rule: <symbol(s)> → <other symbol(s)>
  - A → AB
  - B → b

- Example:
  - AB → ABb → ABbb → ABbbb → Abbbbb → ….

# L-systems

- Parallel rewriting system

- Introduced by biologist Aristid Lindenmayer in 1968
  - to model growth processes of organic systems (plants, algae)
  - characterized by auto-similarity

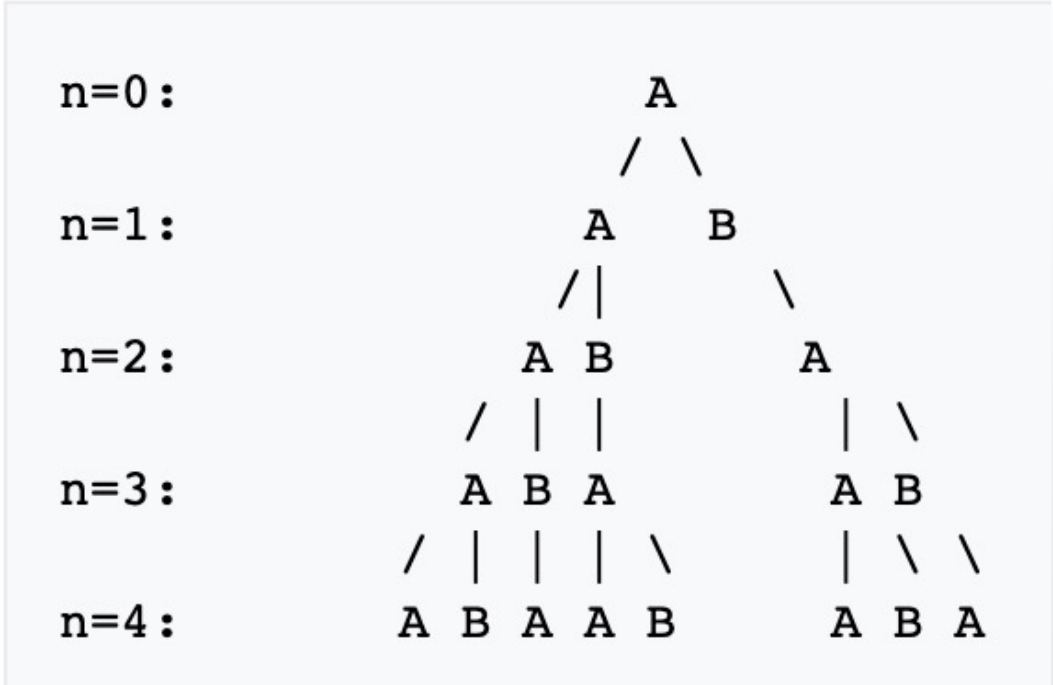
At least there is something nice in it…

# L-systems

- **variables** : A B
- **axiom** : A
- **Production rules** : (A → AB), (B → A)

The initial sentence of the L-system

$n = 0$ : A
$n = 1$ : AB
$n = 2$ : ABA
$n = 3$ : ABAAB
$n = 4$ : ABAABABA



# Categories of L-systems

- Context-free
  - a rule refers only to an individual symbol and not to its neighbours
- Context-sensitive
  - a rule depends not only on a single symbol but also on its neighbours
- Deterministic
  - there is exactly one production for each symbol
- Stochastic
  - there are several rules, each chosen with a certain probability in each iteration

# Graphical interpretation of L-systems

- Instructions for a turtle in Turtle Graphics
  - **F** : move towards and draw
  - **+** : turn left $\alpha$ degrees
  - **-** : turn right $\alpha$ degrees

# Koch snowflake

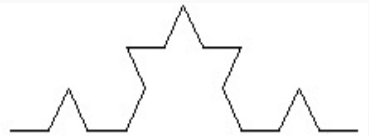
- Example of D0L-system (deterministic and context-free)
  - $\alpha$ = 60°
  - axiom: F
  - rule: F ⟶ F+F--F+F

**Step1**

F+F—F+F

**Step2**

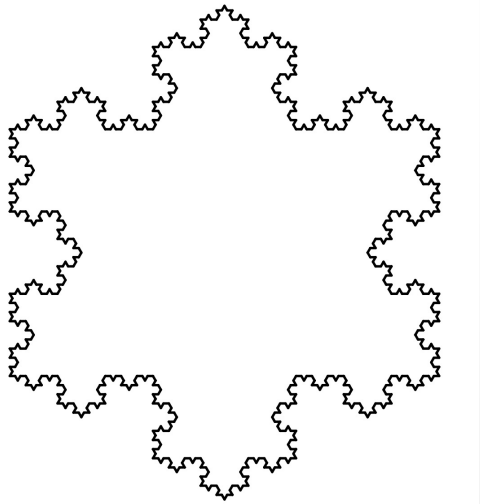
F+F—F+F+F+F--F+F--F+F--F+F+F+F--F+F

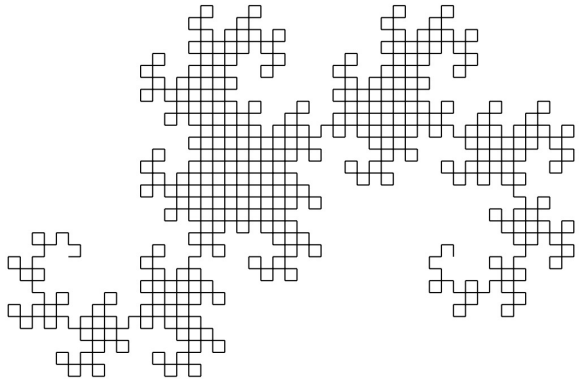


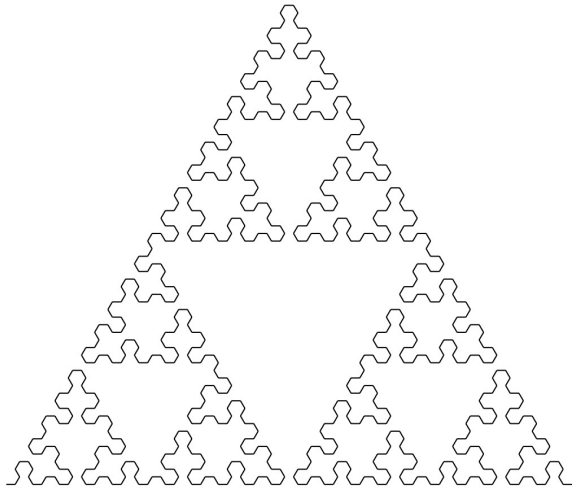
**Step3**

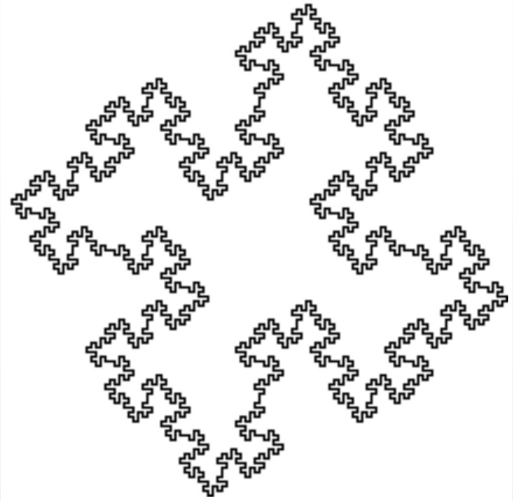
F+F--F+F+F+F--F+F--F+F--F+F+F+F--F+F+F+F
--F+F+F+F--F+F--F+F--F+F+F+F--F+F--F+F--
F+F+F+F--F+F--F+F--F+F+F+F--F+F+F+F--F+F
+F+F--F+F--F+F--F+F+F+F--F+F

# Other examples of D0L-systems







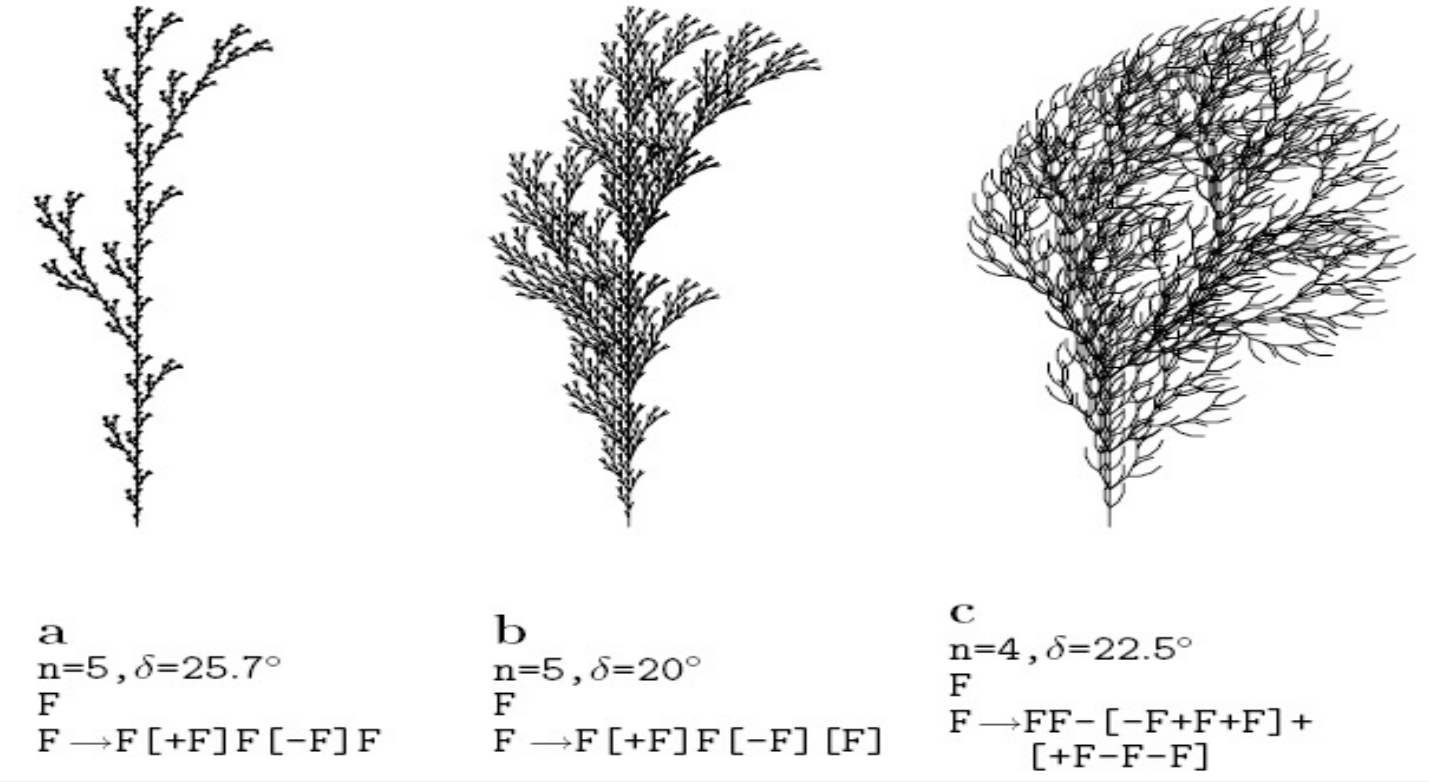


# Bracketed L-systems

- They allow to draw separate *branches*
  - in a plant, several branches start from the same stem
  - once drawn a branch, we need a mechanism to make the turtle go back to the main stem

- Bracketed L-systems introduce 2 additional symbols to manage a LIFO stack
  - **[** : saves current position and orientation of the turtle (*push*)
  - **]** : retrieves last saved position and orientation, and resets the turtle to that position (*pop*)

# Bracketed L-systems and generation of plants

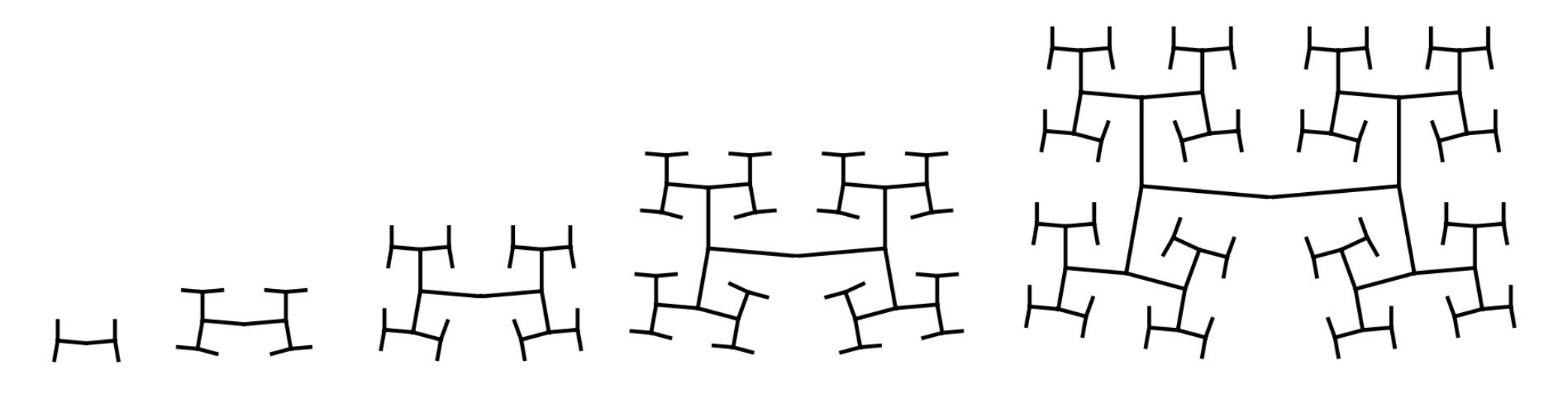



Da P. Prusinkiewicz, A. Lindenmayer, "The Algorithmic Beauty of Plants"
http://algorithmicbotany.org/papers/#abop

# Yes, you can go in 3D

# Our case study

- axiom: [F]--F
- rule: F → [+F][-F]



# Our case study

- A good scheme to procedurally create a suburban area

# L-system in Unity

Source: LSystemGenerator
Folder: PCG

The input is the axiom, defined in the GUI of the MonoBehaviour

At the end, it returns the final rewritten sentence to the main application

recursively, symbols are processed:
- ProcessRules will check the symbol against the rule
- if the symbol is replaced, GenerateSentence will be called recursively

```
// in the beginning: word --> axiom
// when GenerateSentence is called recusively from ProcessRules:
// word --> the rewriting rule from Rule asset
public string GenerateSentence(string word, int iterationIndex = 0)
{

    // when we have done the last iteration, it returns the generated sentence
    // (this is the return to the main application)
    if (iterationIndex >= iterationLimit)
    {
        return word;
    }

    // we use a StringBuilder (we are dynamically modifying a string)
    StringBuilder newWord = new StringBuilder();

    // we append each char in the word in the StringBuilder instance,
    // and we process them
    foreach (var c in word)
    {
        newWord.Append(c);
        ProcessRules(newWord, c, iterationIndex);
    }
    // we return the StringBuilder instance to ProcessRules
    return newWord.ToString();
}
```

this returns to ProcessRules

# L-system in Unity

Source: LSystemGenerator
Folder: PCG

```
private void ProcessRules(StringBuilder newWord, char c, int iterationIndex)
{
    foreach (var rule in rules)
    {
        // for each rule defined in the Rule asset,
        // we check if the character is the letter (in our example, 'F')
        // we need to rewrite
        if (rule.letter == c.ToString())
        {
            // if we opt for a stochastich L-system, there is a probability the rule is not applied
            if (randomIgnoreRuleModifier && iterationIndex > 1)
            {
                if(UnityEngine.Random.value < chanceToIgnoreRule)
                {
                    return;
                }
            }
            // if we find 'F', we get from Rule asset the string to write instead of 'F',
            // we call recursively GenerateSentence, we increment the iteration counter,
            // and we append the result to the StringBuilder instance
            newWord.Append(GenerateSentence(rule.GetResult(), iterationIndex + 1));
        }

    }
}
```

From deterministic to stochastic L-system: we set a probability to have a rule ignored during the parsing stage

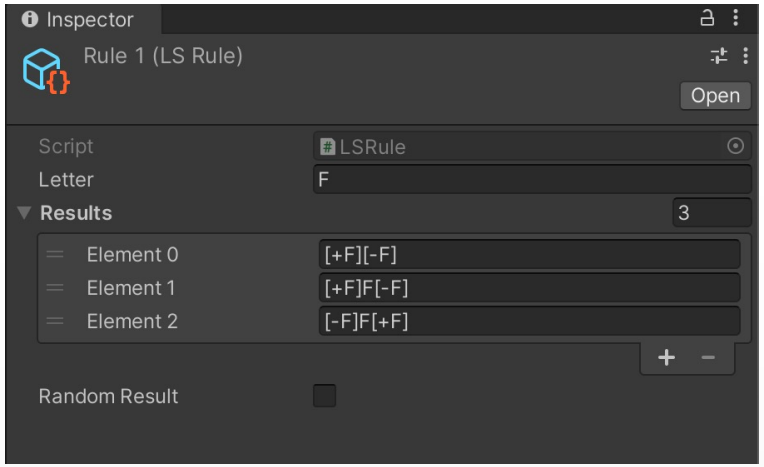
If the processed symbol is equal to the letter defined in a rule, we get the production rule, and we pass it recursively to GenerateSentence

# Production Rule

Source: LSRule
Folder: PCG

Using instances of ScriptableObject, Rules are assets, and they can be modified in the Editor GUI.

However, other solutions are possible, like using text files to store the rules if they must be used in different environments.




```
[CreateAssetMenu(menuName ="LSMapGenerator/LSRule")]
    public class LSRule : ScriptableObject
    {
        public string letter;
        [SerializeField]
        private string[] results = null;
        [SerializeField]
        private bool randomResult = false;

        public string GetResult()
        {
        // Second option to make the L-System stochastic: a random rule is picked from the list
        if (randomResult)
            {
                int randomIndex = UnityEngine.Random.Range(0, results.Length);
                return results[randomIndex];
            }
            return results[0];
        }

    }
```

new rules can be created using submenu in Assets/Create

From deterministic to stochastic again: we can have a set of rules, and one is chosen randomly

# Parse the sequence

Source: LSMapGenerator
Folder: PCG

```
foreach (var letter in sequence)
{
    EncodingLetters encoding = (EncodingLetters)letter;
    switch (encoding)
    {
        // '['
        case EncodingLetters.save:
            lsystem_stack_states.Push(new TurtleParameters
            {
                currentPosition = currPos,
                prevPosition = prevPos,
                direction = dir,
                length = Length
            });
            break;
        // ']'
        case EncodingLetters.load:
            if (lsystem_stack_states.Count > 0)
            {
                var TurtleParameters = lsystem_stack_states.Pop();
                currPos = TurtleParameters.currentPosition;
                prevPos = TurtleParameters.prevPosition;
                dir = TurtleParameters.direction;
                Length = TurtleParameters.length;
            }
            else
            {
                throw new System.Exception("No saved point in our stack");
            }
            break;
        // 'F'
        case EncodingLetters.draw:
            prevPos = currPos;
            currPos += (dir * Length);

            if (drawDebug)
                renderDebug(prevPos, currPos);
            else
                renderMap(prevPos, currPos, dir, length);

            // the length for the next iterations is reduced
            Length -= 2;
            break;
        // '-'
        case EncodingLetters.turnRight:
            dir = Quaternion.AngleAxis(angle, Vector3.up) * dir;
            break;
        // '+'
        case EncodingLetters.turnLeft:
            dir = Quaternion.AngleAxis(-angle, Vector3.up) * dir;
            break;
        default:
            break;
    }
}
```

In the MonoBehaviour application, we parse step-by-step the symbols of the generated sentence (ParseSequence method)

the ParseSequence method can be used as a Coroutine, to see an animation of each step of the process (see LSMapGeneratorCR source file)

# Parse the sequence

Source: LSMapGenerator
Folder: PCG

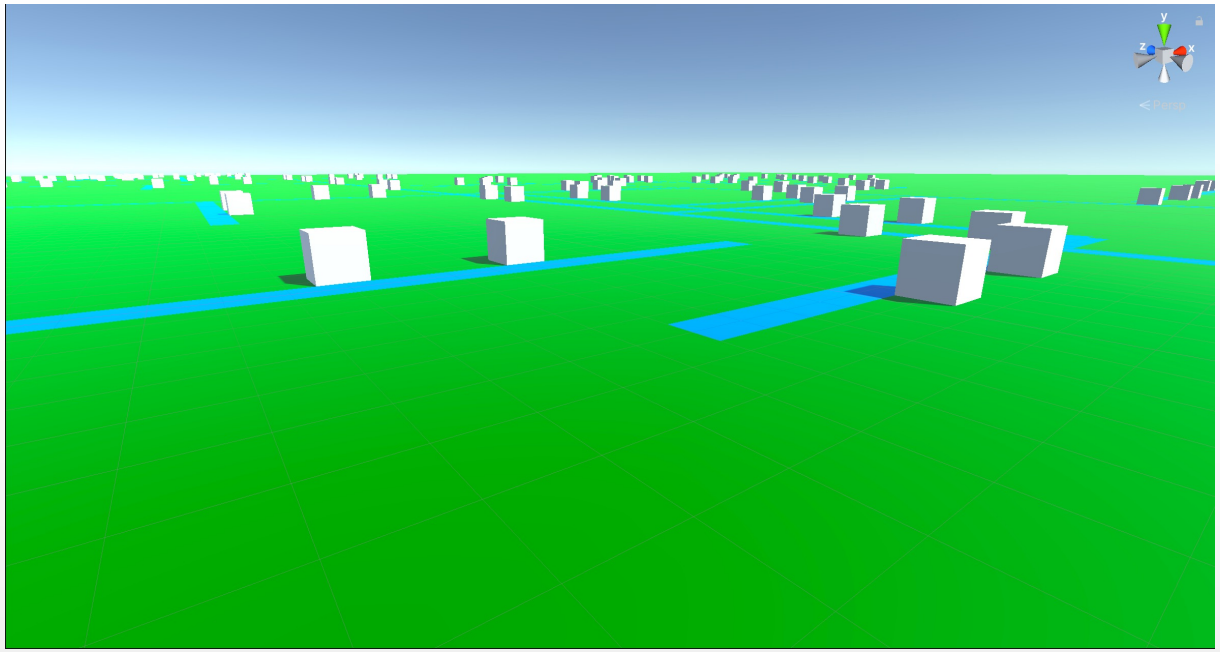
```
foreach (var letter in sequence)
{
    EncodingLetters encoding = (EncodingLetters)letter;
    switch (encoding)
    {
        // '['
        case EncodingLetters.save:
            lsystem_stack_states.Push(new TurtleParameters
            {
                currentPosition = currPos,
                prevPosition = prevPos,
                direction = dir,
                length = Length
            });
            break;
        // ']'
        case EncodingLetters.load:
            if (lsystem_stack_states.Count > 0)
            {
                var TurtleParameters = lsystem_stack_states.Pop();
                currPos = TurtleParameters.currentPosition;
                prevPos = TurtleParameters.prevPosition;
                dir = TurtleParameters.direction;
                Length = TurtleParameters.length;
            }
            else
            {
                throw new System.Exception("No saved point in our stack");
            }
            break;
        // 'F'
        case EncodingLetters.draw:
            prevPos = currPos;
            currPos += (dir * Length);

            if (drawDebug)
                renderDebug(prevPos, currPos);
            else
                renderMap(prevPos, currPos, dir, length);

            // the length for the next iterations is reduced
            Length -= 2;
            break;
        // '-'
        case EncodingLetters.turnRight:
            dir = Quaternion.AngleAxis(angle, Vector3.up) * dir;
            break;
        // '+'
        case EncodingLetters.turnLeft:
            dir = Quaternion.AngleAxis(-angle, Vector3.up) * dir;
            break;
        default:
            break;
    }
}
```

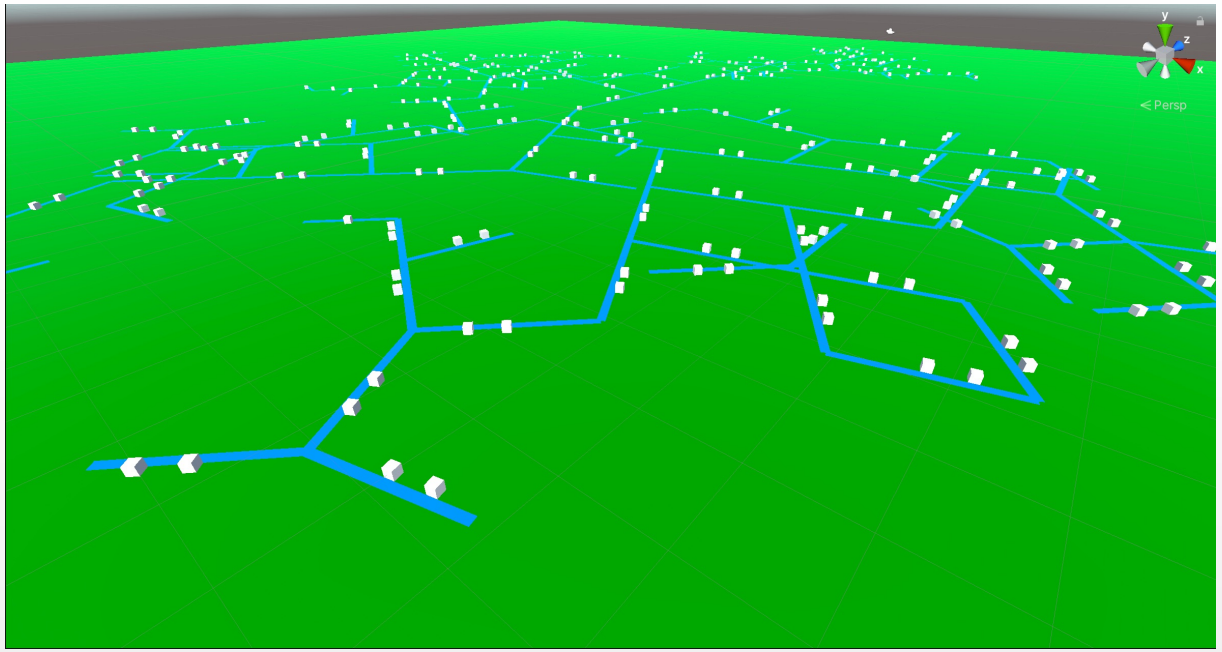
Using the Turtle graphics approach, we are not logging in any way the final positions, directions, etc

If we need to apply further elaborations after the parsing of the L-system, we can save the current Turtle state in a list, and to perform all the rendering in a later step (see LSMapGeneratorPostRender source file)

# L-system results

Scene: LSystemMap
Source: LSMapGenerator
Folder: PCG

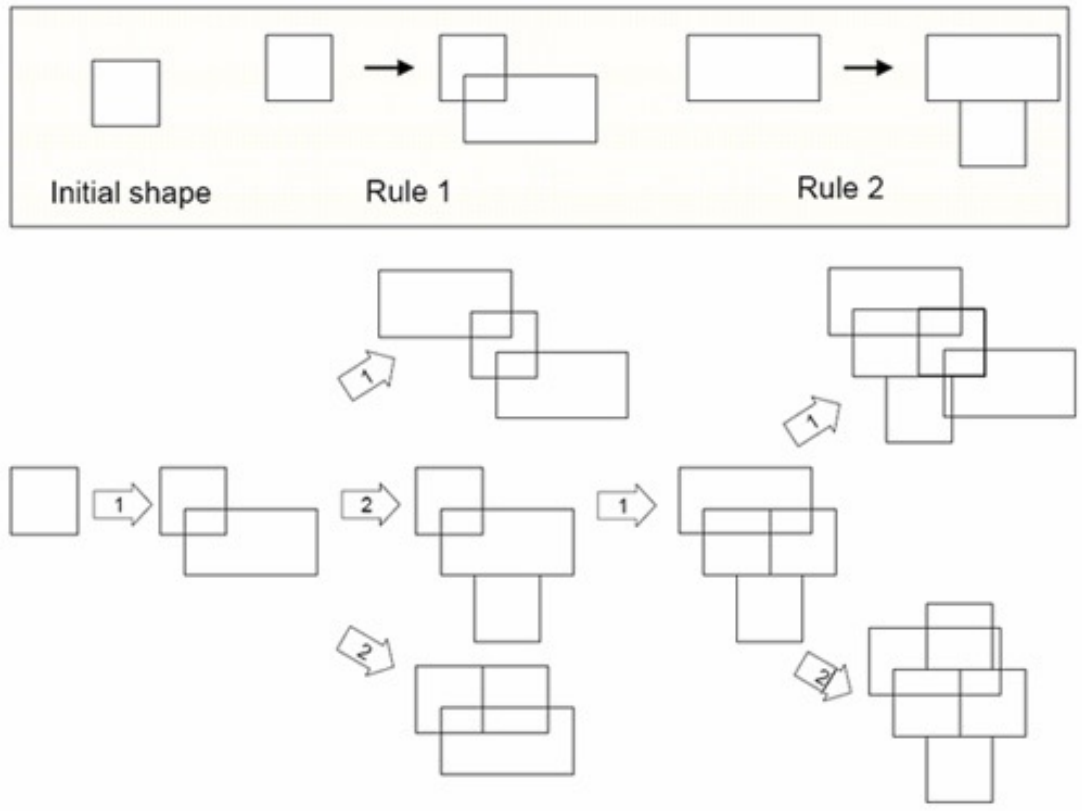




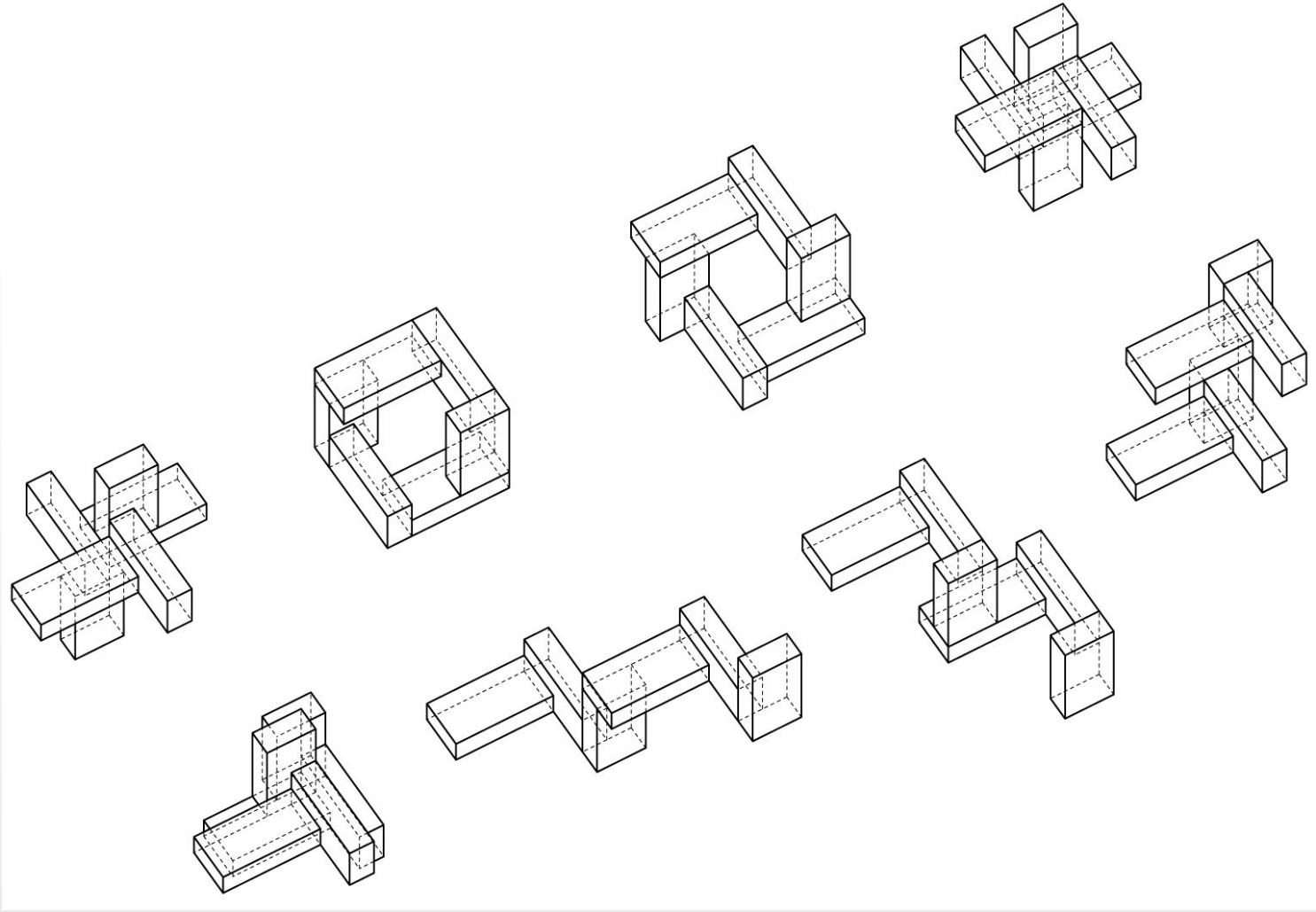
# Shape Grammars

- Production systems to generate geometric shapes

- Similar approach of L-systems
  - set of basic shapes
  - Production rules to place/combine/transform them in new shapes



# Shape Grammars



# Shape Grammars

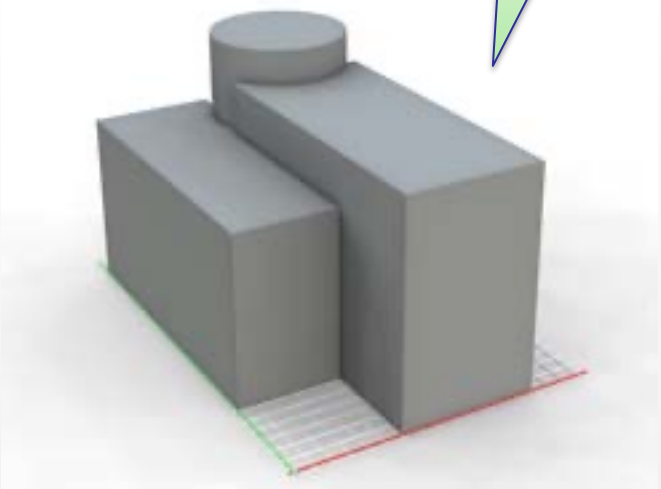
- A popular application of Shape Grammars is the procedural generation of buildings

basic shapes

production rule

[ T(0,0,6) S(8,10,18) I(”cube”) ]
T(6,0,0) S(7,13,18) I(”cube”)
T(0,0,16) S(8,15,8) I(”cylinder”)

the final result



P. Mueller, P. Wonka, S. Haegler, A. Ulmer, L. Van Gool. “Procedural Modeling of Buildings” ACM Transactions on Graphics 25(3) pp 614-623, 2006

# Shape Grammars

- A popular application of Shape Grammars is the procedural generation of buildings

P. Mueller, P. Wonka, S. Haegler, A. Ulmer, L. Van Gool. “Procedural Modeling of Buildings” ACM Transactions on Graphics 25(3) pp 614-623, 2006

# On the generation of graphical assets using grammars

- CG rendering **requires** correct 3D meshes
  - polygonal meshes must have a correct topology
  - without duplicates, intersections, stretches, etc.

These are aspects strictly related to CG, and they are investigated in other courses.

However, you can not avoid them completely (i.e., you can not avoid to lose your temper for them….)


- The procedural creation of models and environments requires an accurate processing of the generated CG assets
  - applying specific mesh processing techniques to clean-up the connections between previous and new ‘’branches’’ of polygonal mesh in a model
  - by checking (and correcting) possible overlaps when determining procedurally the positions of a set of generated models on a map
  - ….

# A more complex example

The technique was used to develop the first versions of CityEngine software

- Procedural generation of a city
  - Y.I. Parish, P. Müller, “Procedural modeling of cities”. Proceedings of ACM SIGGRAPH '01, pp. 301-308, 2001

- Based on a combination of different L-systems and image-based techniques

# Step 1: roads generation

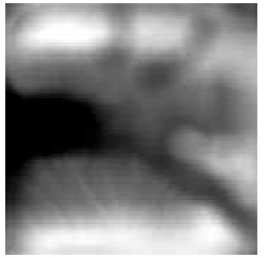
- advanced L-System to generate streets
  - parameters for the production rules are evaluated and (possibly) changed on the basis of some global and local constraints
  - constraints represents characteristics of the city and of the environment
    - population density
    - elevation
    - presence of water (sea, rivers, lakes, etc)
    - The desired pattern for the roads (radial, checker, etc)
  - some of these informations are passed to the generator using *maps*

# Step 1: roads generation

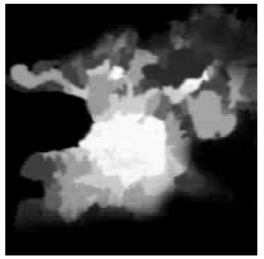
The maps can be based on real data (GIS) or they can be created procedurally (e.g., using Perlin noise)

presence of water

elevation

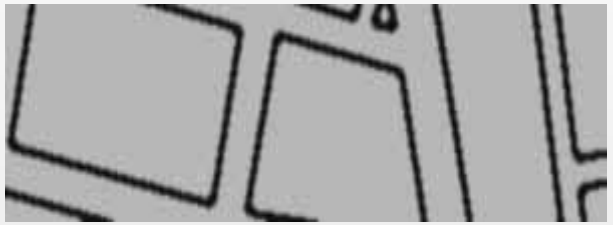
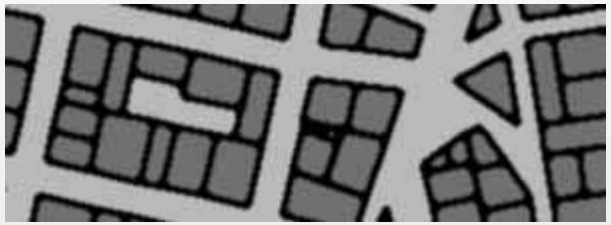


population density



# Step 2: modeling of buildings

- The generated roadmap, and the maps, are used to subdivide the ‘’populated’’ area in lots
  - areas delimited by streets crossing are scaled according to the streets width
  - Lots are created recursively considering again the population density, the presence of tall buildings (skyscrapers), etc.

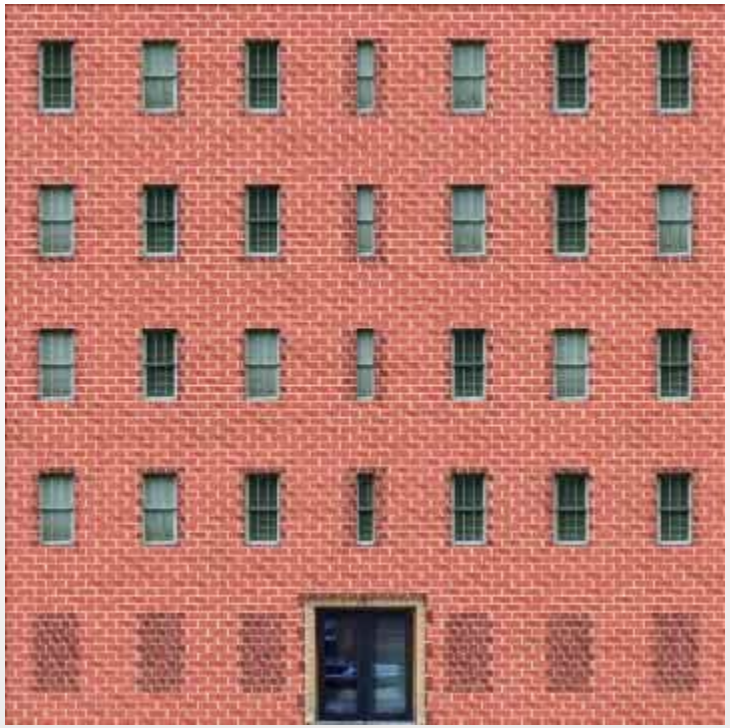




# Step 2: modeling of buildings

- Buildings are placed in lots using a stochastic parametric L-system
  - 3 types of available buildings (skyscrapers, commercial, residential)
  - again, maps are used to select the appropriate building type in each lot
  - Different set of production rules are used for each building type

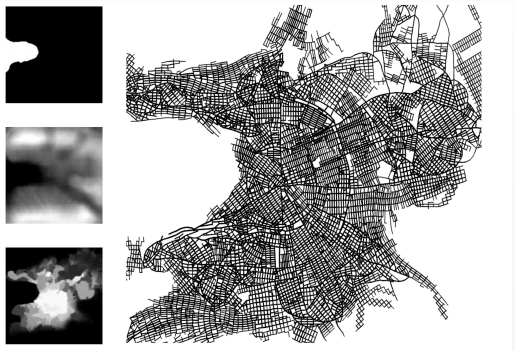
# Step 2: modeling of buildings

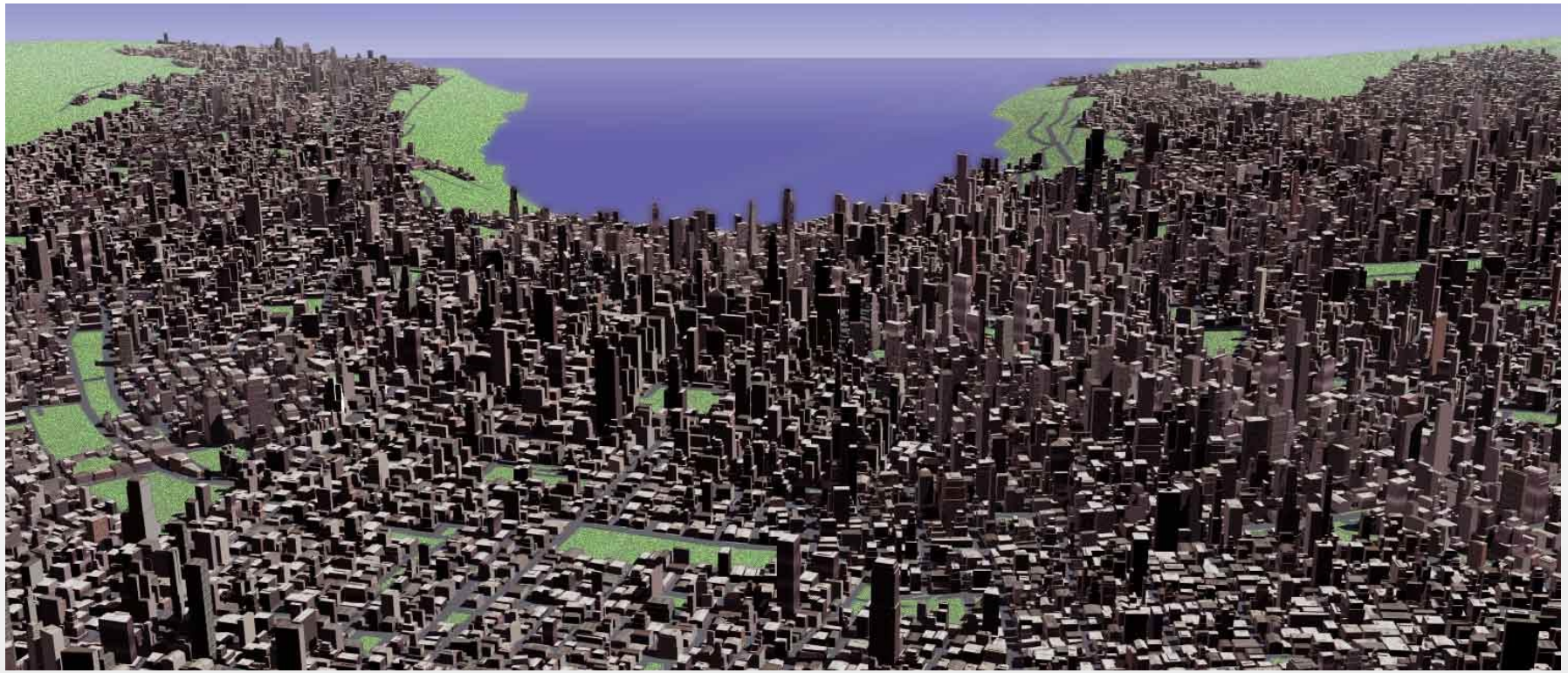
- Finally, textures for the buildings facades
  - using techniques from procedural texturing field


# And putting everything together……

Approx. 260.000 buildings

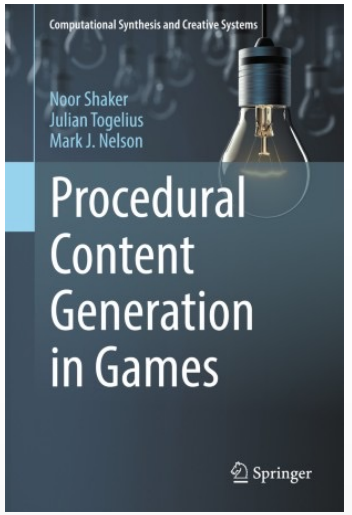




# Study Material

- Procedural Content Generation in Games
  ISBN 978-3-319-42716-4
  by N. Shaker, J. Togelius, M. J. Nelson
  - you can download it for free from Unimi IP addresses from:
    https://link.springer.com/book/10.1007/978-3-319-42716-4
  - an open access version of the book is available at:
    http://pcgbook.com/

  Chapter 5 up to § 5.3 included




- The Algorithmic Beauty of Plants
  by P. Prusinkiewicz, A. Lindemayer
  - you can download it for free from :
    http://algorithmicbotany.org/papers/#abop

  Chapter 1