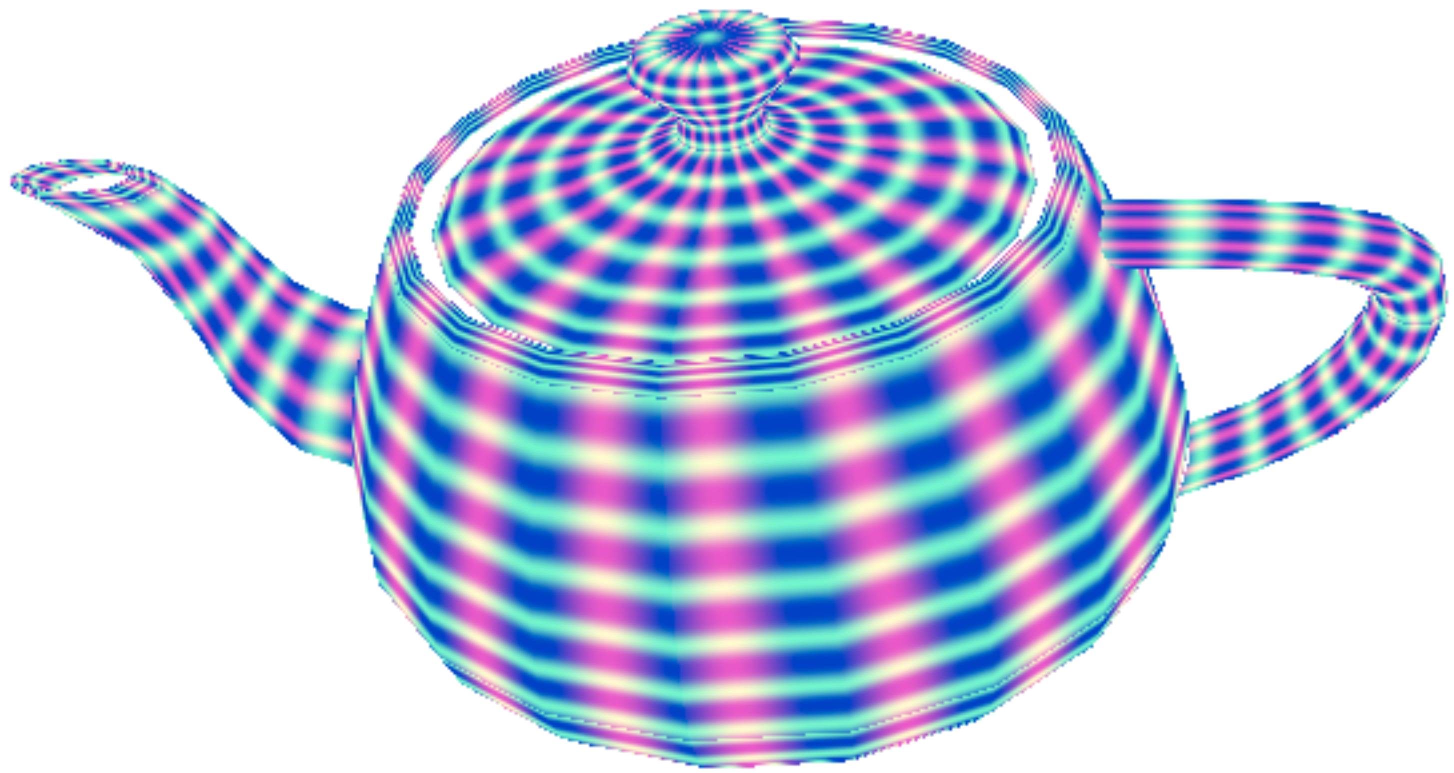
TNM084
Procedural images

Ingemar Ragnemalm, ISY



Lecture 7

L-systems with variations

Fractal Brownian Motion

Fractal terrains and other applications

Lab 3

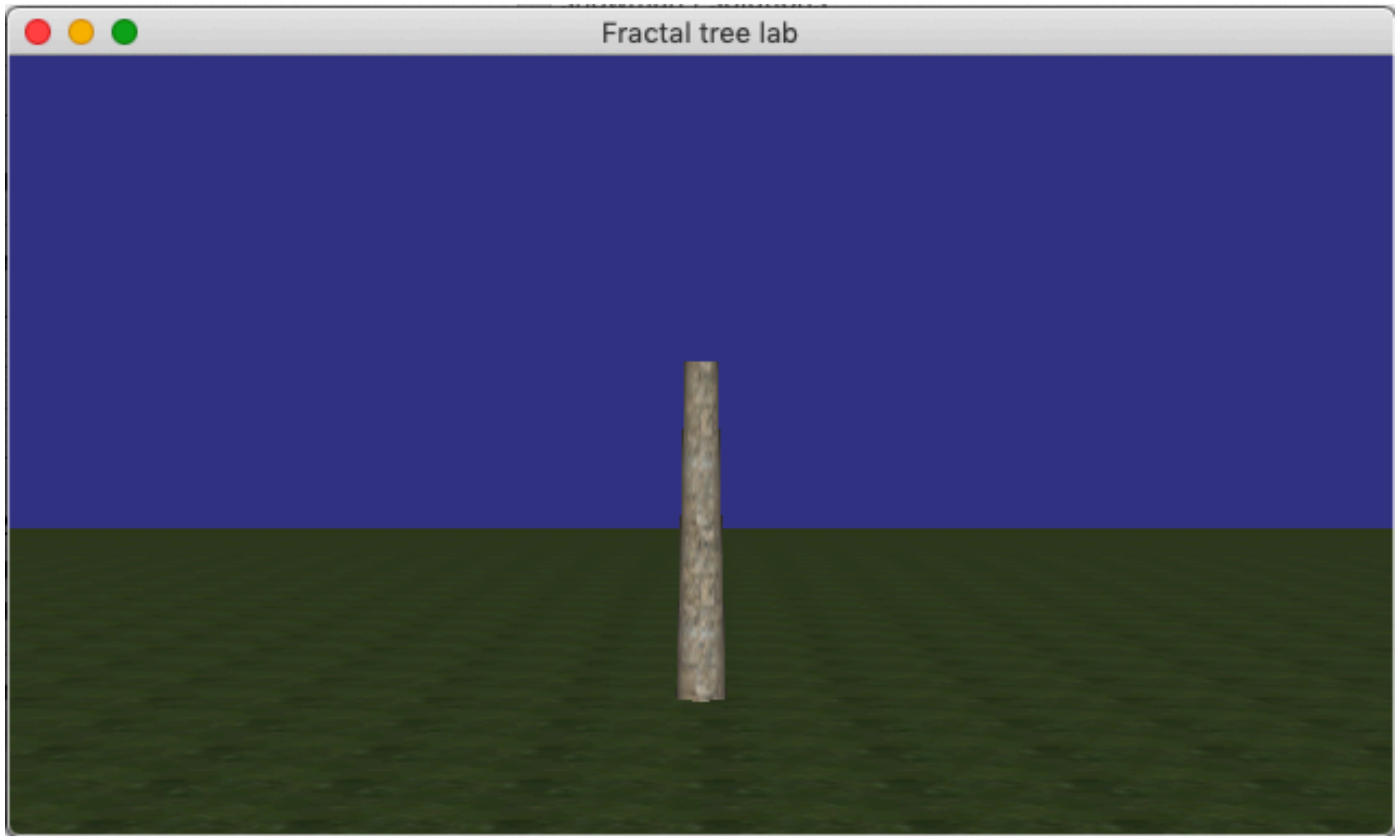
Packaged with the lab material.

Theme: Fractals

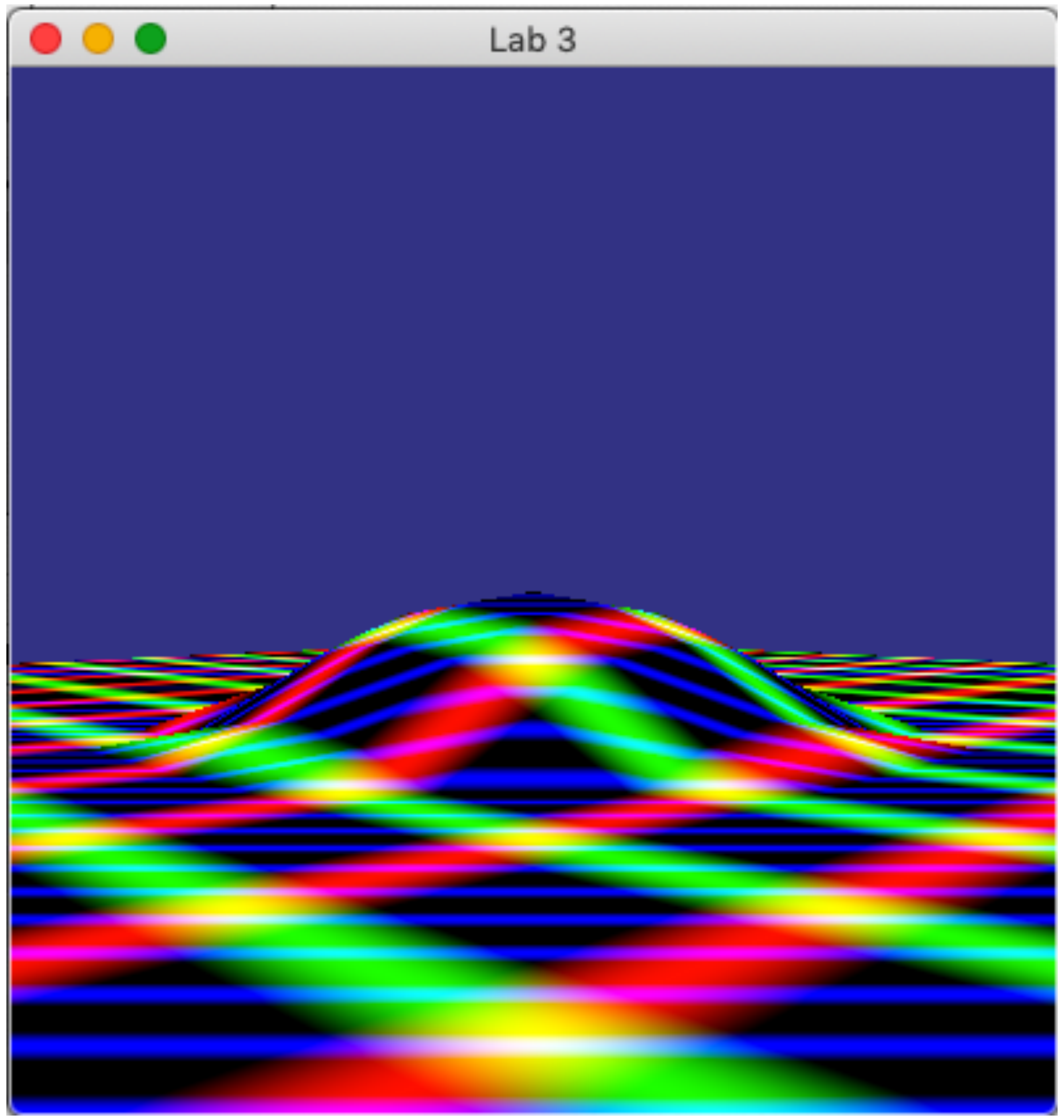
3a: Procedural tree

3b: Procedural terrrain

3a: I give you one piece of wood. You make the tree.



3b: I give you a boring surface. You make a nice terrain.



Lecture questions

1. What is the connection between L-systems, turtle graphics and geometric fractals?

2. How can you describe a Koch fractal with an L-system?

3. What symbols are need for an L-system to describe a tree?

4. How do you rescale noise between FBM levels?

5. What FBM method has the lowest computational complexity?

6. What is the computational advantage of gradient noise for FBM?

Last time: Fractals

Geometric generations of self-similar fractals

Fractal dimension

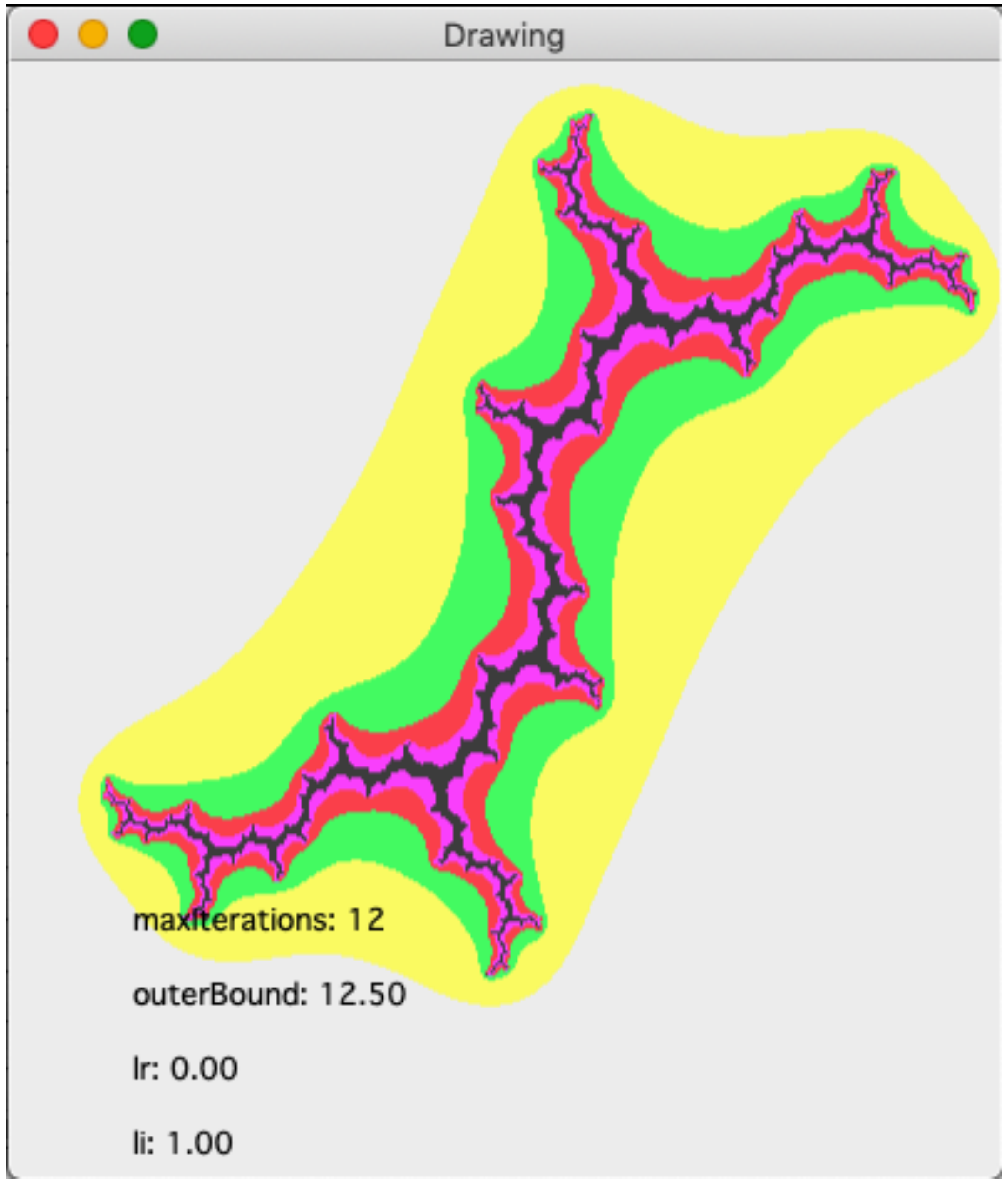
Statistically self-similar fractals

Self-squaring fractals in complex space

Self-squaring fractals

The Julia set

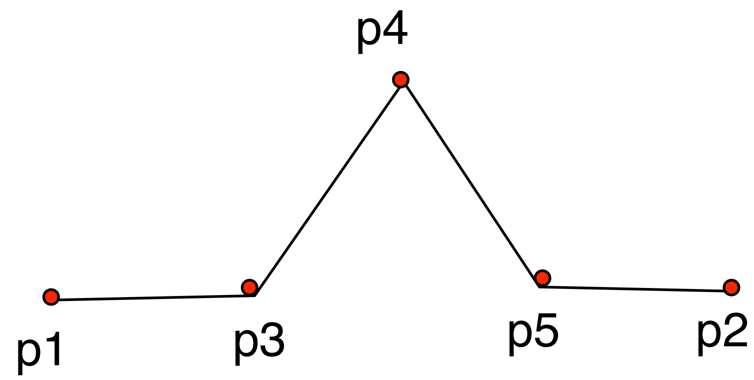
$$z_{k+1} = z_k^2 + \lambda$$



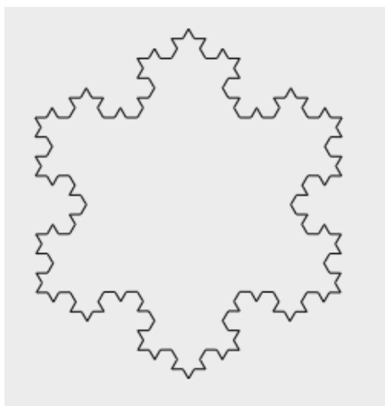
Julia set for $\lambda = (0, 1) = 0 + j$

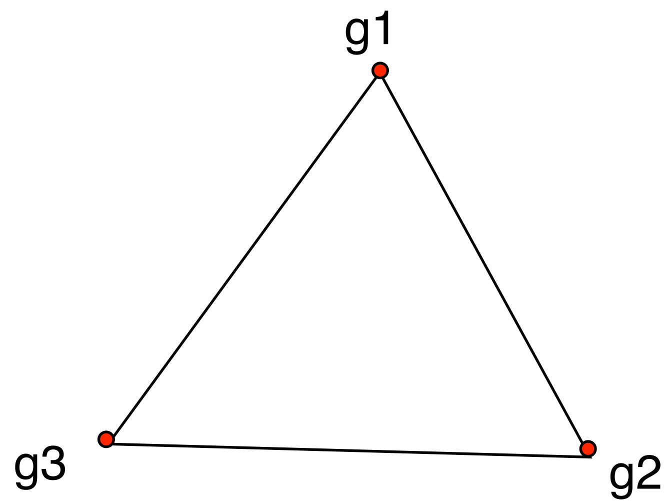
We also did
Geometric self-similar fractals

```
procedure DrawKoch(p1, p2, depth)

if depth >= maxDepth then

    MoveTo(p1)
    LineTo(p2)
    return

else

    calculate p3, p4, p5 as the three points inside the generator

    DrawKoch(p1, p3, depth+1)
    DrawKoch(p3, p4, depth+1)
    DrawKoch(p4, p5, depth+1)
    DrawKoch(p5, p2, depth+1)
```



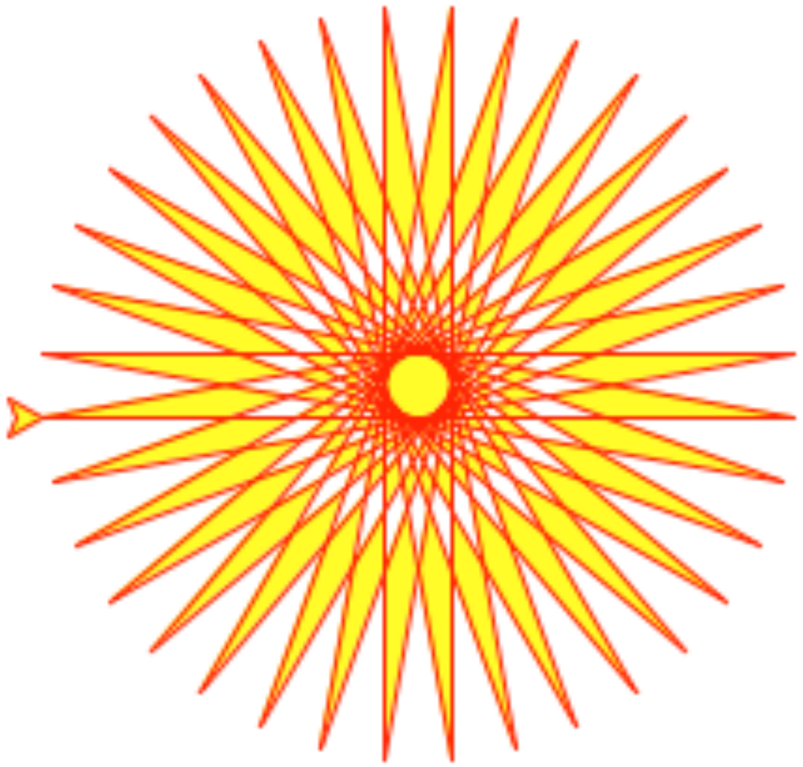

```
main procedure:

Choose three generator points, g1, g2, g3

DrawKoch(g1, g2, 0)
DrawKoch(g2, g3, 0)
DrawKoch(g3, g1, 0)
```





And I also demonstrated
Turtle graphics

```
from turtle import *
color('red', 'yellow')
begin_fill()
while True:
    forward(200)
    left(170)
    if abs(pos()) < 1:
        break
end_fill()
done()
```



We will now combine these two!

L-systems

Developed by A Lindenmayer to model the development of plants

Based on parallel string-rewriting rules

Excellent for modelling organic objects and fractals

(Information mostly from the book and from a course presentation of lost origin)

L-systems basics

Begin with a set of "productions", replacement rules, and a "seed" axiom

Example:

Rules (productions): B -> ACA and A -> B

Axiom: AA

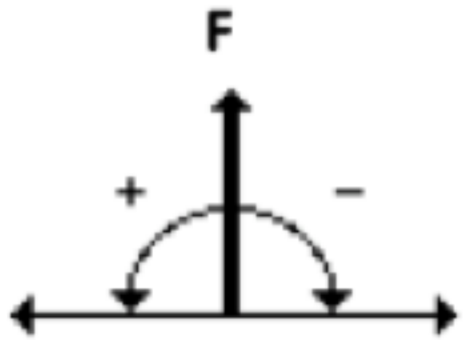
Produces the sequence AA, BB, ACAACA, BCBBCB, ACACACACACA...

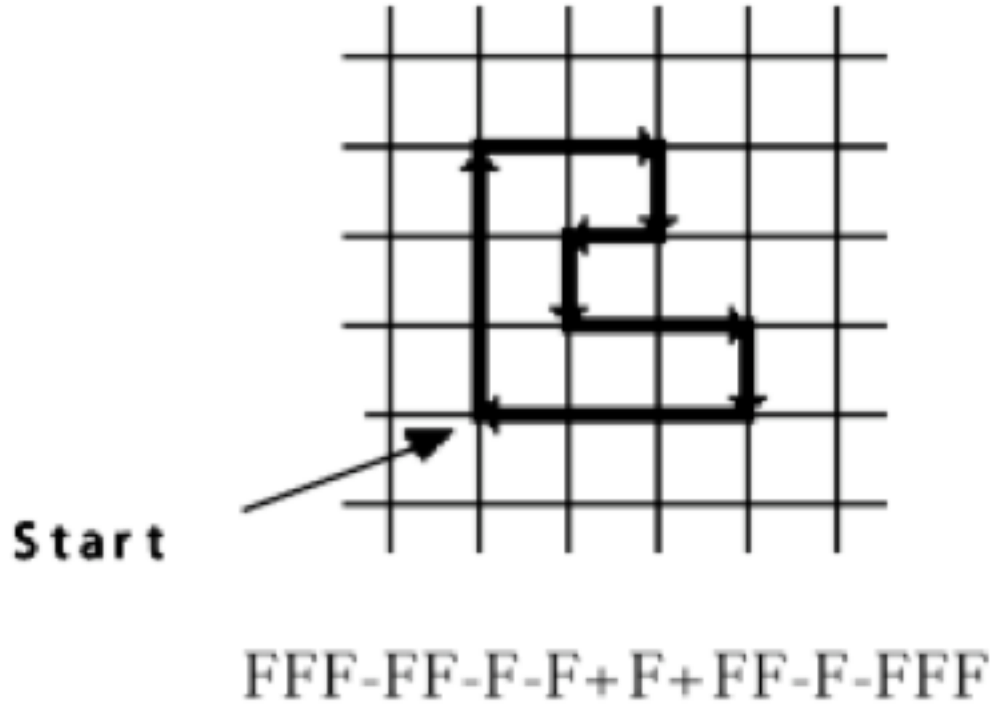
Strings are converted to graphics representaions as turtle graphics commands

L-systems to turtle graphics

Turtle commands:

F: move forward while drawing
f: move forward without drawing
+: Turn left by angle ∂
-: Turn right by ∂





Koch curve from L-systems

F → F + F - - F + F

produces a Koch curve if

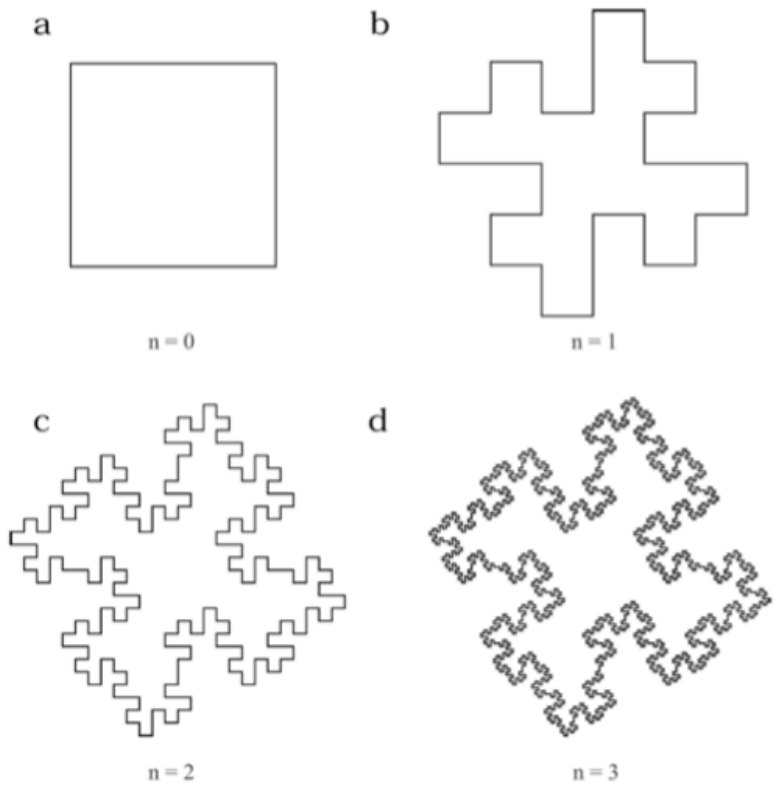
+ = turn left
- = turn right

Needs a rescaling for each step

Koch (90 degree version)

Axiom: F-F-F-F

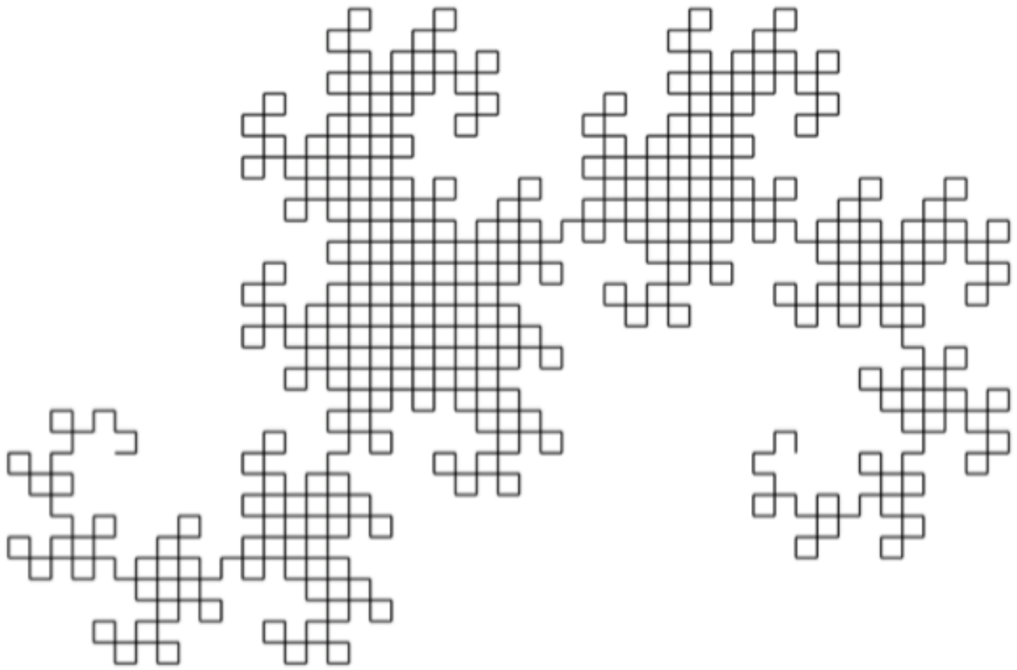
Production: F -> F-F+F+FF-F-F+F



Dragon curve

Axiom: Fl

Productions:
Fl -> Fl+Fr+
Fr -> Fl-Fr-



Extensions to L-systems

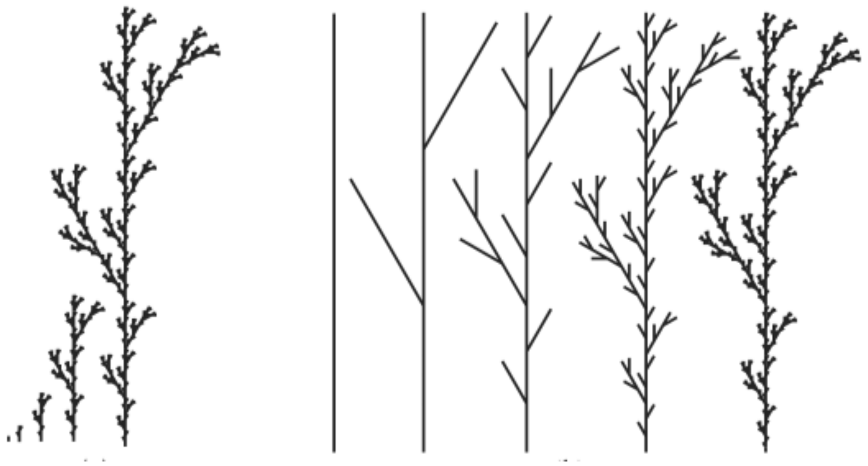
- Productions dependent on neighboring symbols
- Stack support (bracket symbols)
- Stochastic: Choose productions randomly
- Parametric: Variables can be passed between productions
- Numerical arguments

Extended L-system by stack brackets: Simple tree

F → F [+ F] F [-F] F

produces the simple branch using

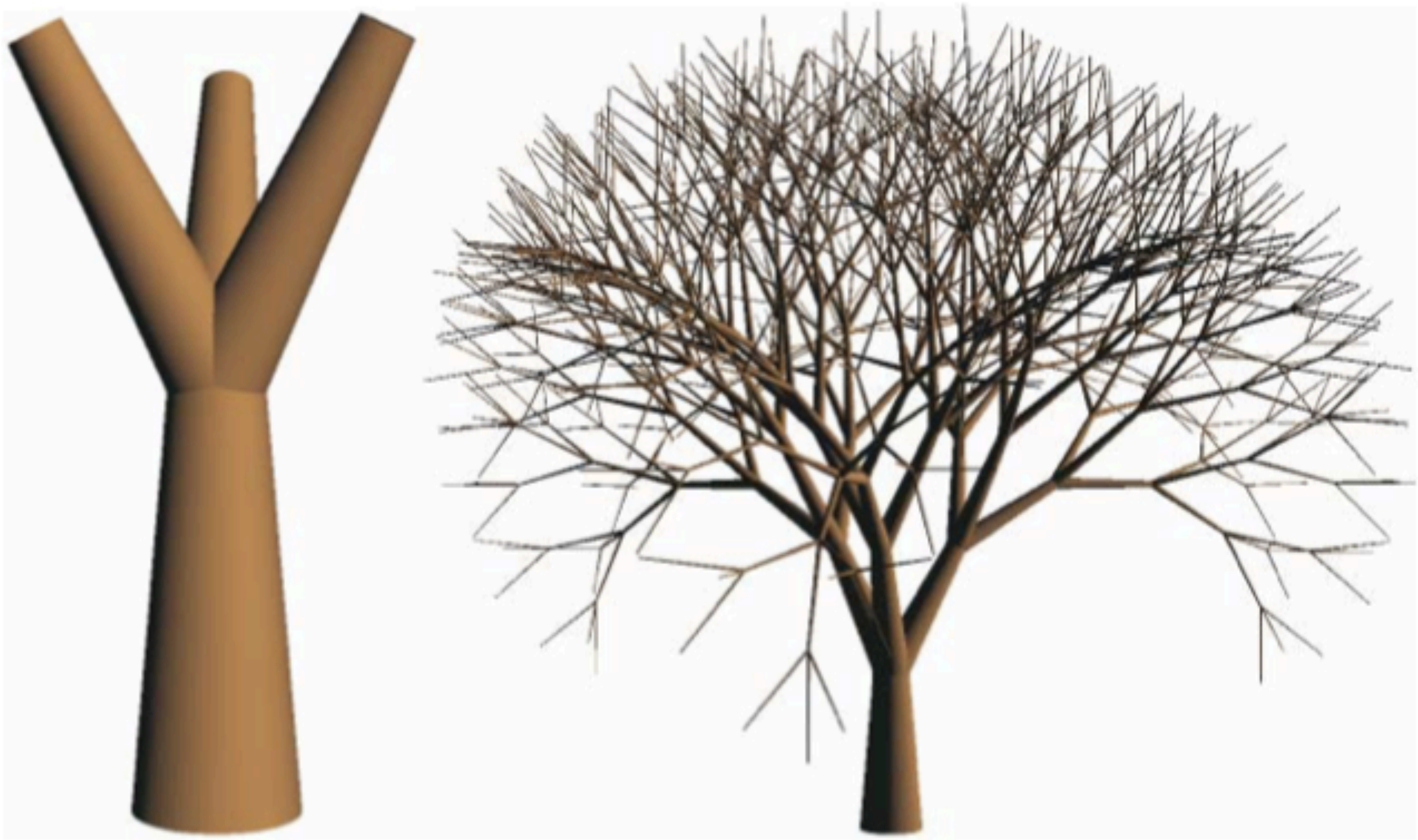
+ = turn left
- = turn right
[= push state
] = pull state



Better tree

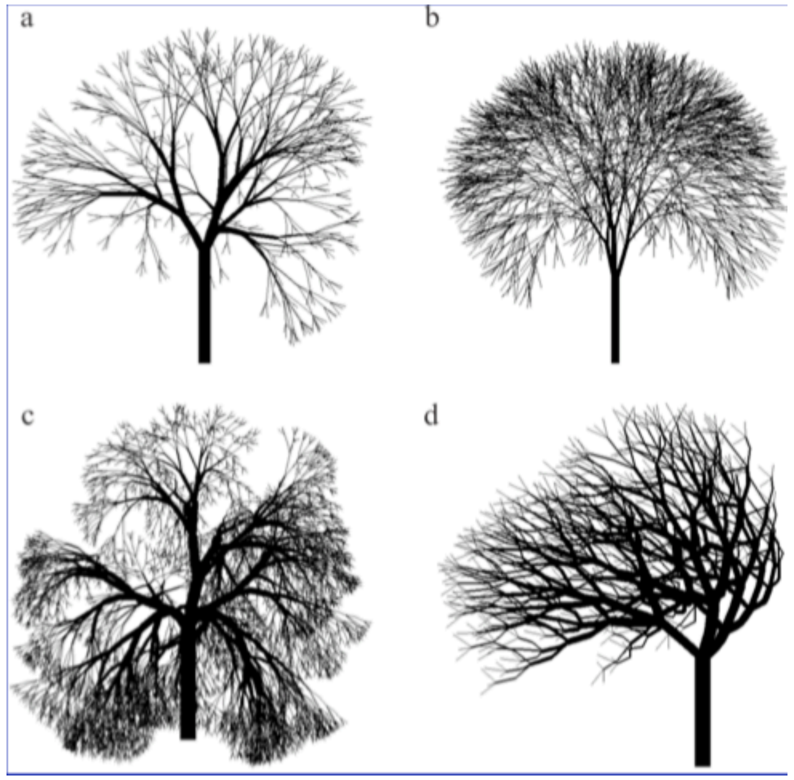
F → F [&F] [/F][&\F]

with additional symbols for rotation.



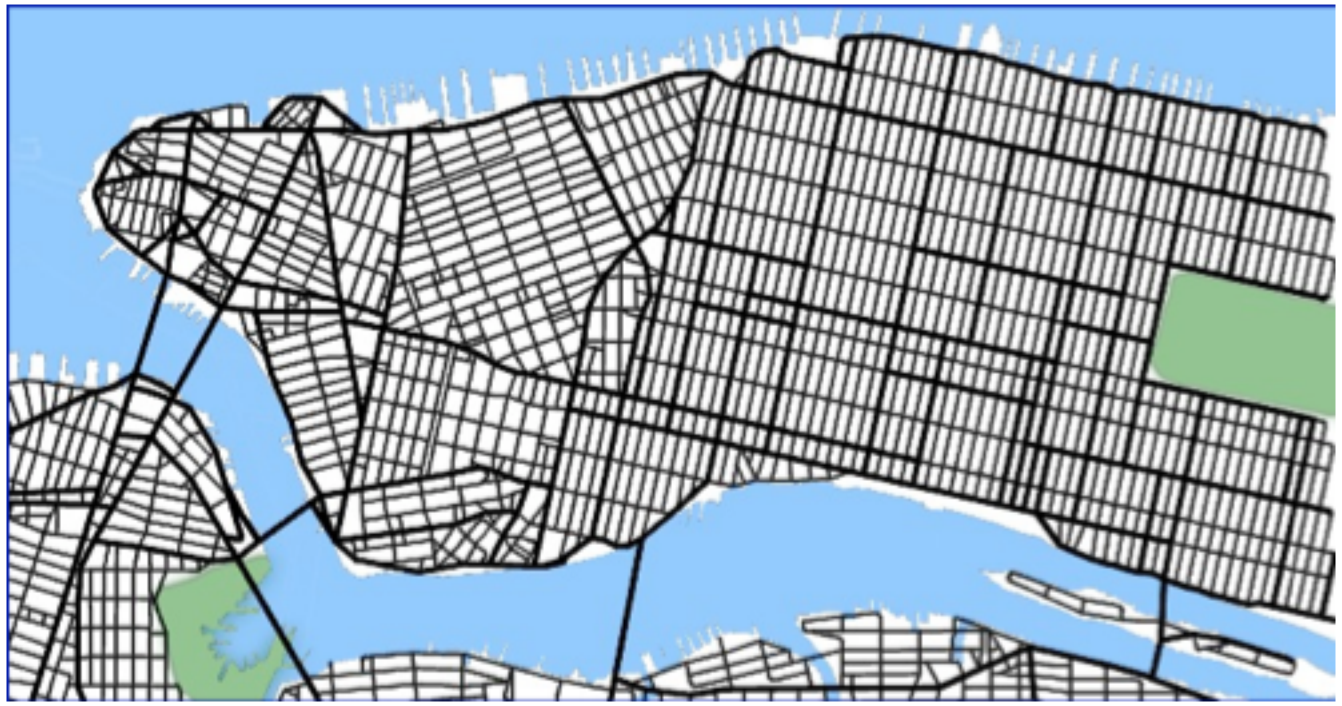
Plants made by L-systems

Many plants can be produced, but finding the production rules is challenging



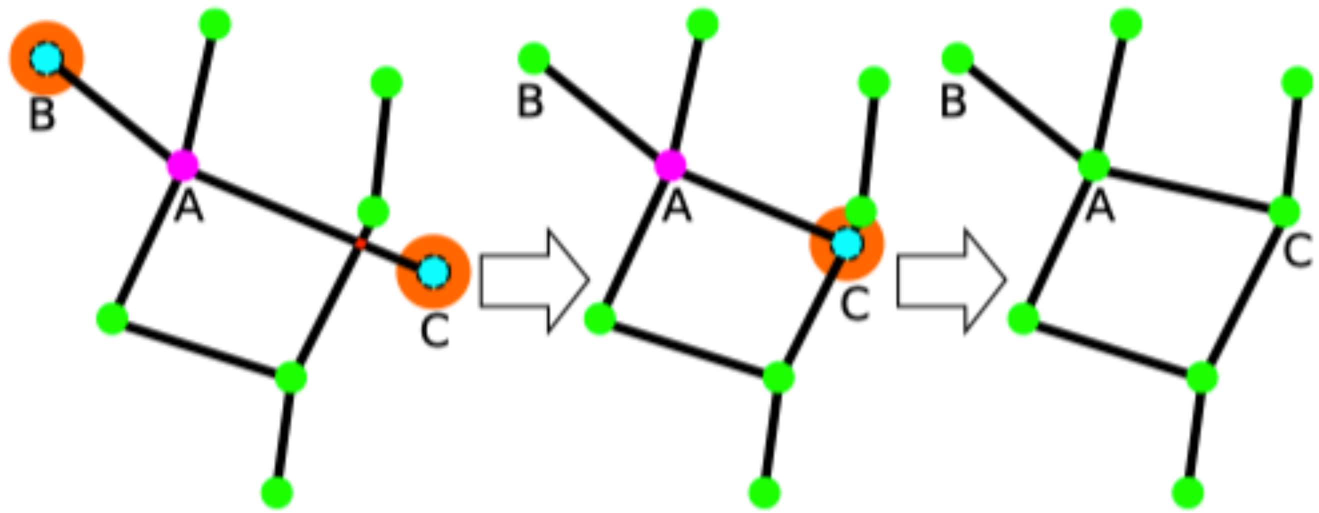
Road networks

- Start with a single street
- Branch and extend with parametric L-system
  - Parameters tweaked by custom values for goals and constraints
  - Constraints allow for parks, bridges etc



Road networks by graph-based L-systems

Checks for overlaps, rewrites the result to allow loops.
(Thesis by Martin Jormedal.)



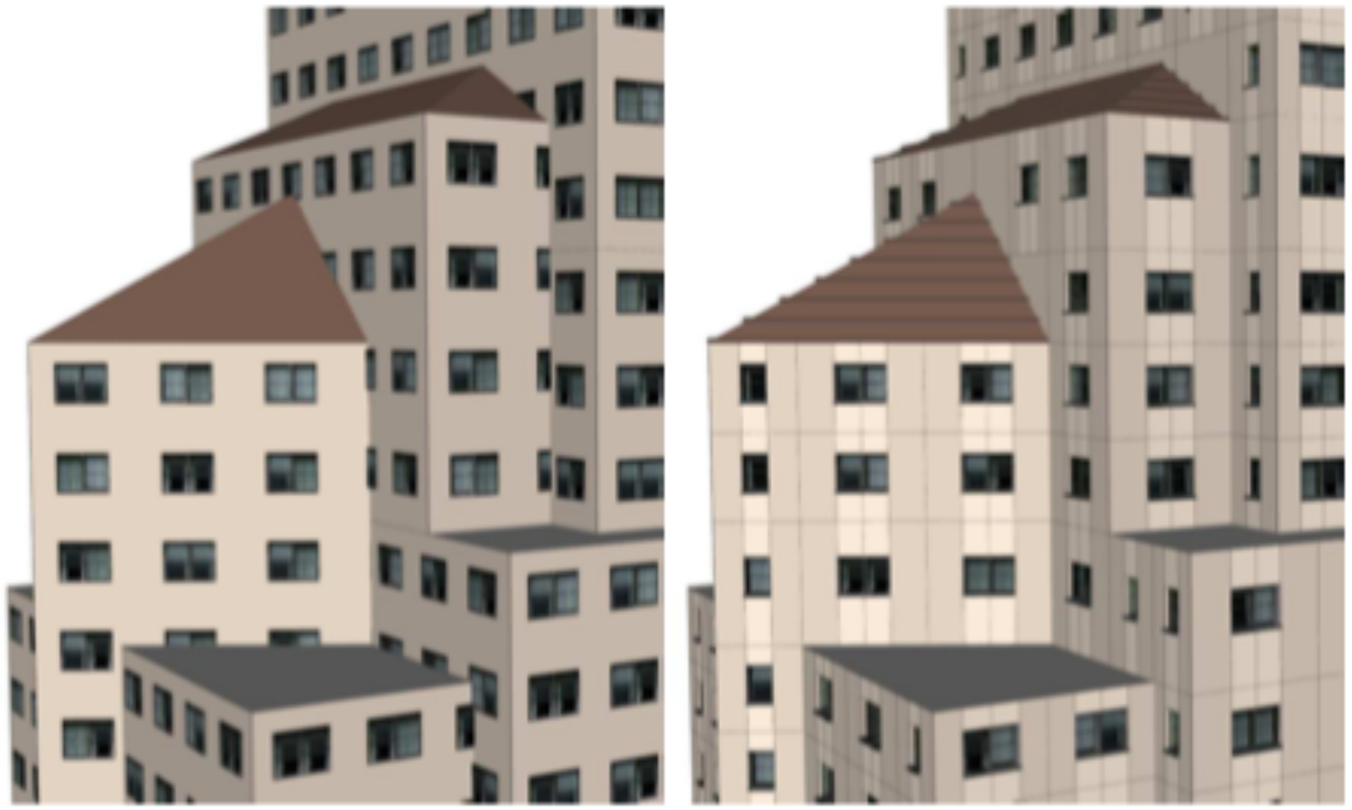
Generation of buildings and cities

Given the street map, generate buildings

Base the buildings on simple shapes given by the city blocks.

CGA (Computer Generated Architecture)

CGA is a shape grammar used to create procedural buildings. (Müller et al 2006). It works with operations such as splits and repetitions.



CGA basic rules

CGA has four basic rules:

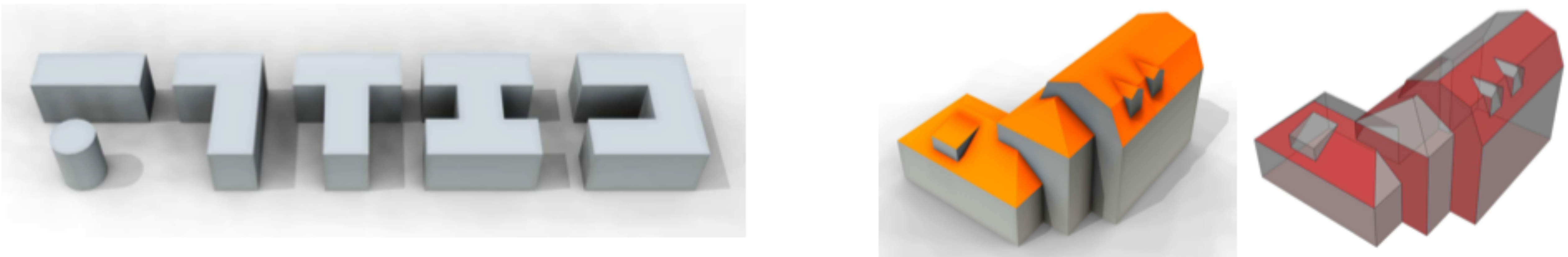
Basic split
Scaling
Repeat
Component split

These are far from enough for buildings but can still describe complex shapes.

Mass modelling

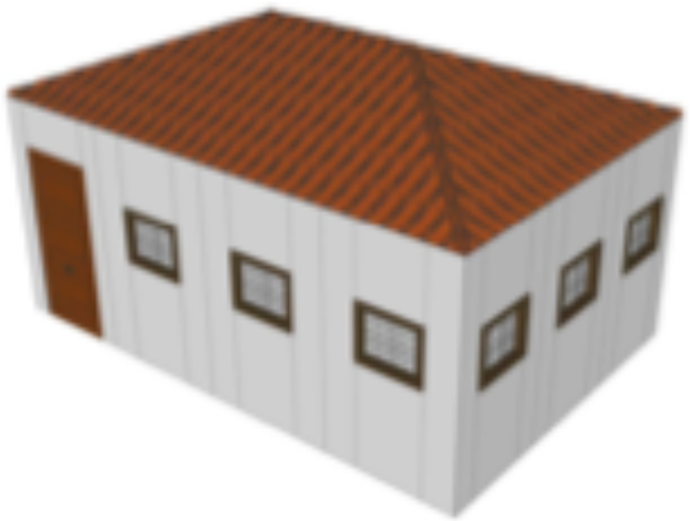
The CGA method also includes a stage of "mass modelling", buidling the basic shape from a set of components.

A box is the most fundamental shape, from which a set of basic shapes are formed:

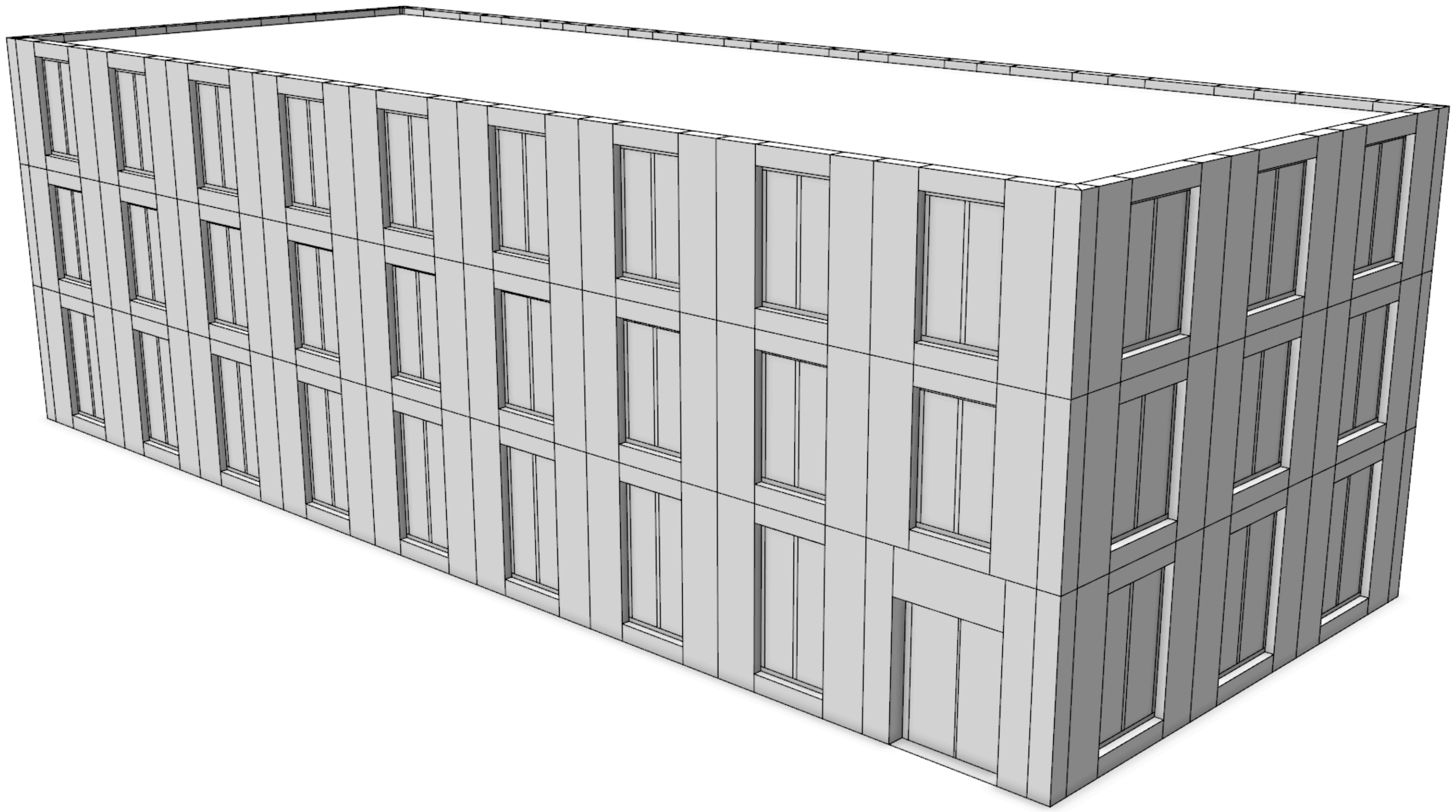


CGA basic grammar

The simplest CGA grammar builds from 16 rules, including the "footprint", and on top of that rules about windows, doors, roofs and more. Even with that many components, only this simple buildings can be made:



Simple example from CityEngine



https://doc.arcgis.com/en/cityengine/latest/tutorials/tutorial-6-basic-shape-grammar.htm

My example

A few rules similar to CityEngine/CGA

base -> foundation + flat

flat -> walls + flat

flat -> top

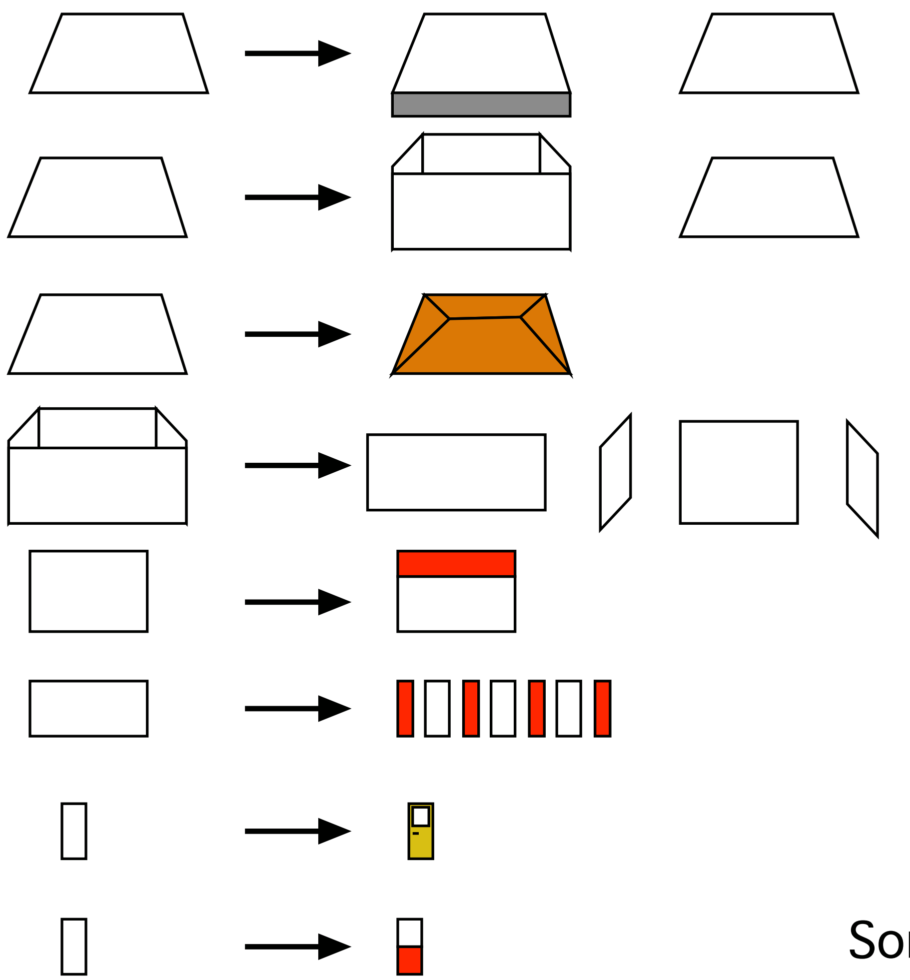
walls -> $side_1$+$side_2$+$side_3$+$side_4$

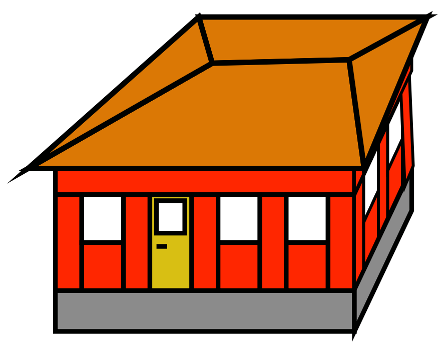
side ->p_{top} + p_{bottom}

p_{bottom} -> split to p_{part} and p_0

p_{part} -> door

p_{part} -> window

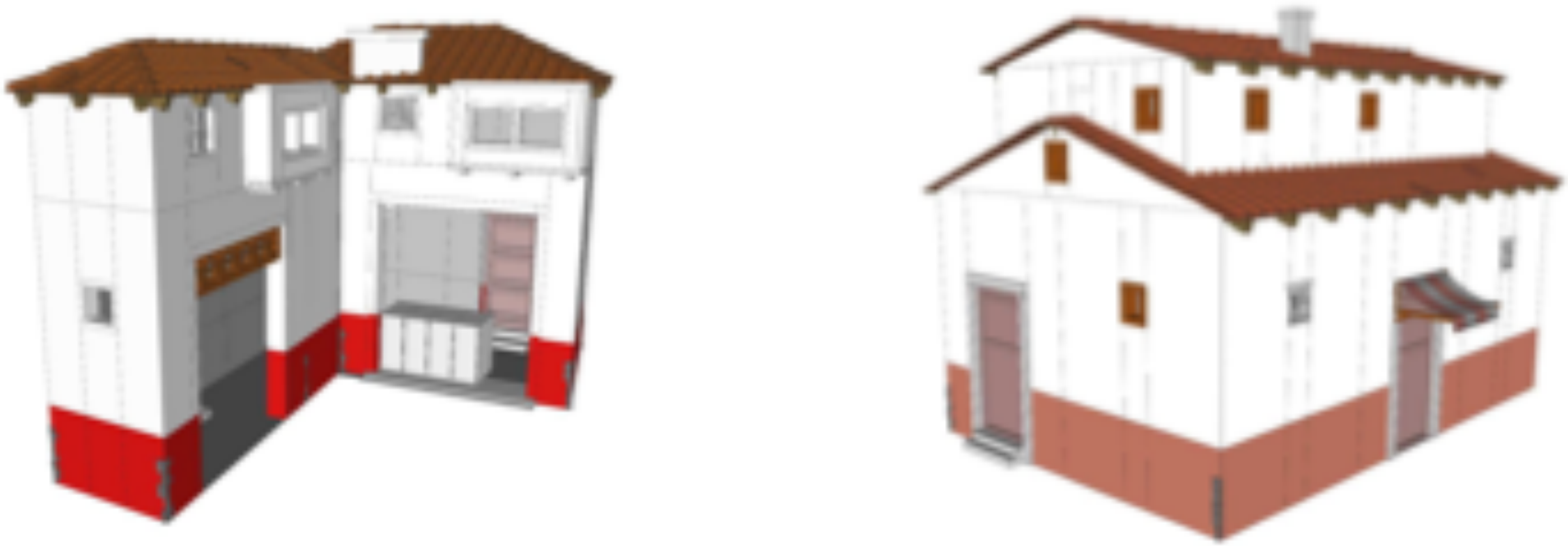




Something like this (or better if I had had the time)

CGA extended grammar

CGA was extended further to produce more complex models. It is clear that the grammar must be hand-tailored for each type of architectures, but the results are impressive.



CGA vs L-systems

Important difference between L-systems and CGA:

L-systems are for *growth*. True also for road networks.
L-systems typically are very self-similar on different resolutions.

CGA is made for subdivision, and has very different rules on different levels. Thus, it is *not* a fractal!

Interior

Procedural generation of interiors was studied by Andersson (2019).

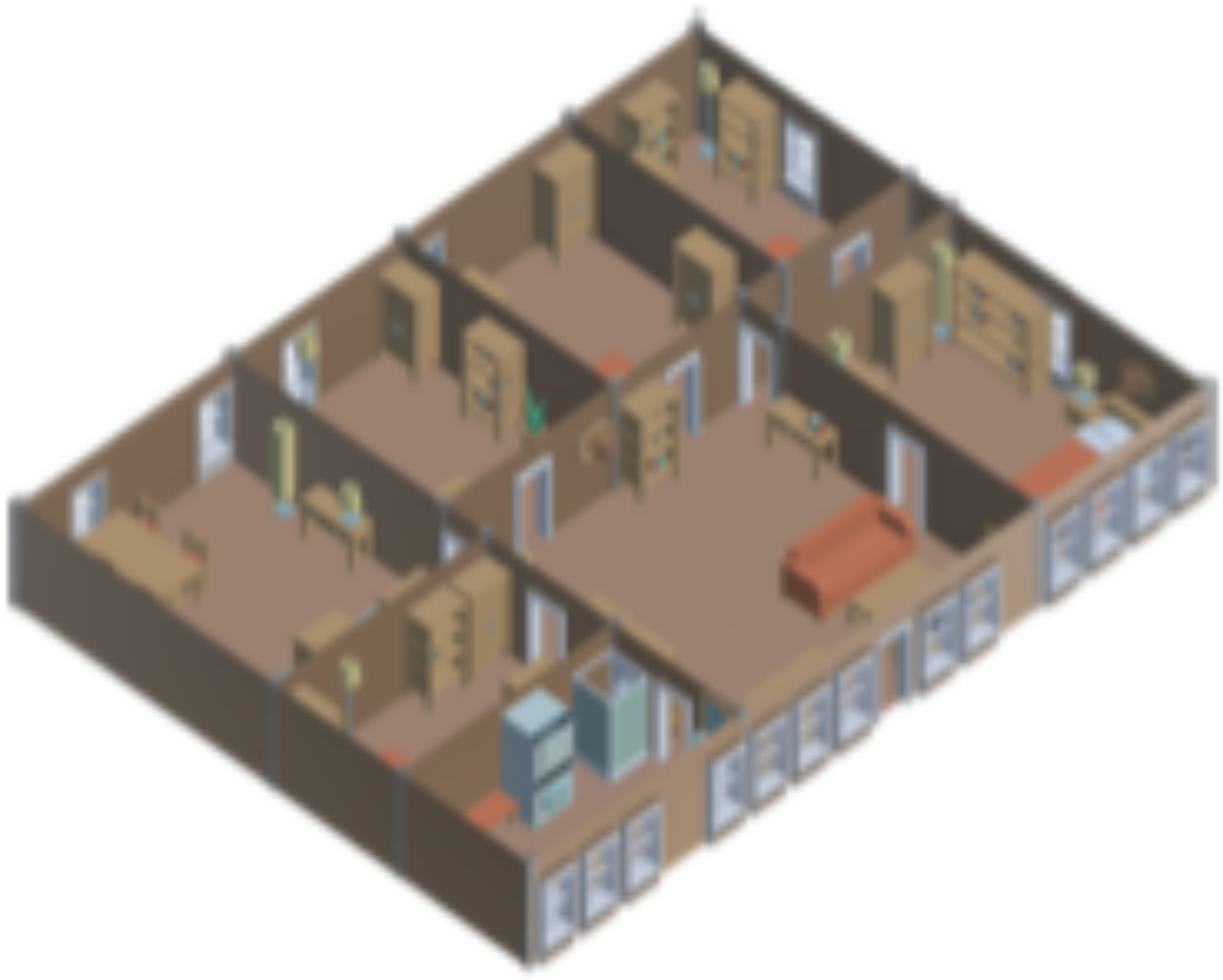
The problem includes:

- Splitting into rooms
- Placement of furniture

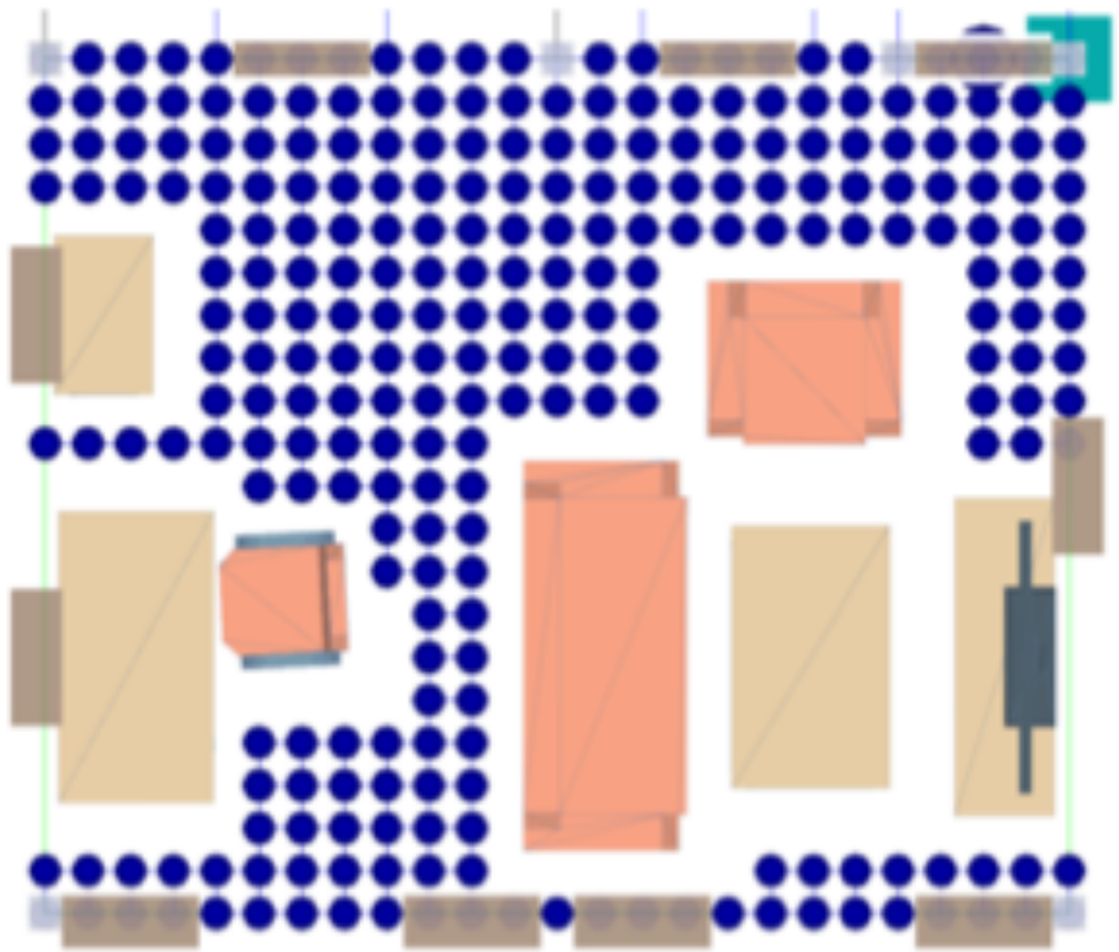
The latter includes:

- Collision detection
- Analysis of free space for acceptable paths

A space split into rooms



Analysis of free space by a grid



Buildings + road networks = city generator

Several algorithms for city generation exists

- Algorithmic, L-systems or similar
- Organic, build roads and buildings based on previous generation of the map

Grammars or code?

Should we use grammars or recursive code?

Grammar: Write grammar, insert in reusable evaluator - specific for each variant of the grammar!

Recursive code: New program every time, but much more flexible.

When is either to prefer?

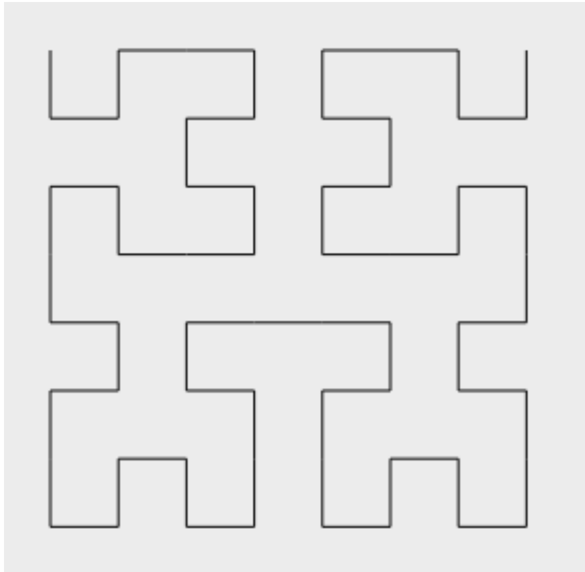
Every grammar can be rewitten as a program

Examples: Koch, dragon, etc

Example: Hilbert curve.

Grammar: Process the string by the productions rules N times. Then parse the string to do turtle graphics.

Code: Formulate the grammar as code. Each production rule is a function call.



Buildings?

May use a sequence of function calls

• Make basement, call:

-> Make floors

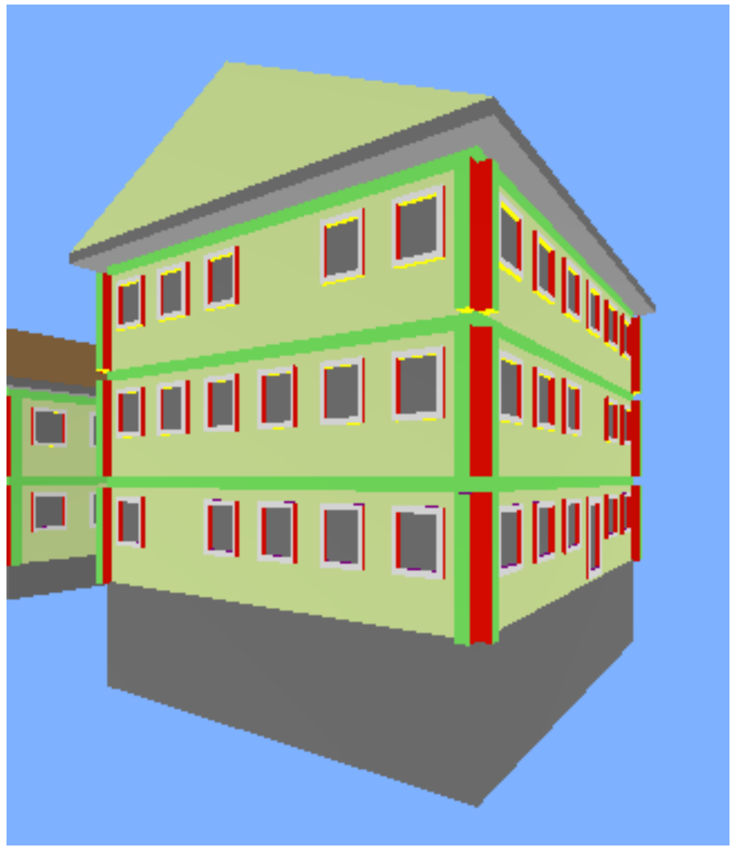
-> Make walls

-> Split walls to parts

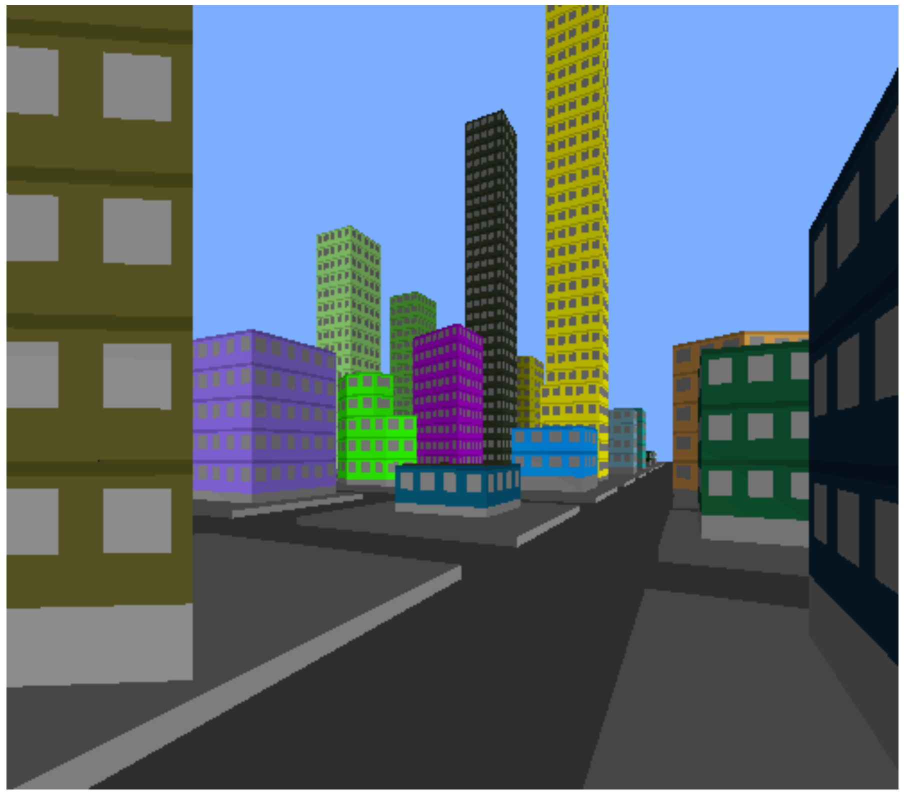
Etc. Or do this as a grammar.

GLUGG City

Simple prototype, needs more work despite considerable freedom and many steps. Based on function calls.



MakeWallPanel
MakeWallPanelExtruded
MakeWindow
MakeWallSection
MakeWallTop
MakeWallCorner
MakeWall
MakeStory
MakeTopBox
MakeFlatRoof
MakeBadRoof
MakeRoof
MakeStories
CreateBasement
MakeBottom



Best method?

Taste?

Which method suits your problem?

Problem suited for recursion?

Code is easier to extend with new options. Grammars are easier to edit.

A grammar might be easier for non-programmers?