

**SimplyRhino, London**  
**February 12-14,2020**

# **Python Scripting for Rhino/Grasshopper**

Day 3

*Long Nguyen*  
*Thu Nguyen-Phuoc*

# DataTree

# Example: Create a list of spheres from a tree of points

Please open file **DataTree.gh**

```
import Rhino.Geometry as rg
spheres = []
for points in iPointTree.Branches:
    for point in points:
        spheres.append(rg.Sphere(point, 2.0));
oSpheres = spheres;
```

# Example: Create a TREE of spheres from a TREE of points

Please open file **DataTree.gh**

```
import Rhino.Geometry as rg
from Grasshopper import *

sphereTree = DataTree[rg.Sphere]()

for path in iPointTree.Paths:
    points = iPointTree.Branch(path)
    spheres = []
    for point in points:
        spheres.append(rg.Sphere(point, 2.0));
    sphereTree.AddRange(spheres, path)

oSpheres = sphereTree;
```

# Parallel Computation

# Example: Intersecting MANY lines with a mesh

Please open the file **Parallel Computation.gh**

```
from Rhino.Geometry.Intersect import Intersection
import time
from ghpythonlib import parallel

def ProcessOneCurve(curve):
    return Intersection.MeshPolyline(iMesh, curve.ToPolyline(0.1, 0.1, 0.1, 0.1))[0]

startTime = time.time()

intersectionLists = parallel.run(ProcessOneCurve, iCurves, False)

allIntersections = []

for intersectionList in intersectionLists:
    allIntersections.extend(intersectionList)

endTime = time.time()

print endTime - startTime

oResult = allIntersections
```

# Recursion

- Recursive Functions
- Fractals and L-Systems

# What is recursion?

Recursion: Defining something using **itself**

## An example (from Maths)

Factorial      Definition:  $n! = 1 * 2 * 3 * 4 * \dots * (n-1) * n$

Alternative definition, using **recursion**:

$$1! = 1$$

$$\text{and, } n! = n * (n-1)!, \text{ where } n > 1$$

Why it works?

$$\begin{aligned} 10! &= 10 * 9! \\ &= 10 * 9 * 8! \\ &= \dots \\ &= 10 * 9 * 8 * \dots * 1! \end{aligned}$$



# Recursion in Programming

- ▶ Recursive function: a function that call **itself**

## Hands-on Example:

**A recursive function that computes the factorial of a given number**

Mathematical definition:  $1! = 1$

$n! = n * (n-1)!$ , where  $n > 1$

A recursive function

```
def Factorial(n):  
    if n == 1:  
        return 1  
    else:  
        return n * Factorial(n-1)  
  
print(Factorial(4))
```

# Recursive function: behind the scene

A recursive function

```
def Factorial(n):  
    if n == 1:  
        return 1  
    else:  
        return n * Factorial(n-1)  
  
print(Factorial(4))
```

When the code is execute:

`Factorial(4)` ...  
... will call `Factorial(3)`, ...  
... which will call `Factorial(2)`, ...  
... which will call `Factorial(1)`, ...  
... which will return 1

**A recursive function  
must have at least one  
EXIT CODITION**

# Can we do it non-recursively?

A non-recursive factorial function

```
def Factorial(n):  
    result = 1  
    for i in range(2, n + 1):  
        result *= i  
    return result
```

In fact, **any** recursive algorithm can be converted to **non-recursive** form

But in many cases, recursive form is:

- Easier to write
- Easier to understand (and debug)
- More elegant/succinct
- More closely follow the original definition (e.g. from maths, fractals)

But, if not used carefully, they may also consume a lot of **memory** and **computation time**

# Live Example: Creating Rhino geometry recursively

Step 1: Recursively create (continuous) line segments

```
import rhinoscriptsyntax as rs

def MoveRecursively(startPoint, moveVector, currentStep, maxStep):

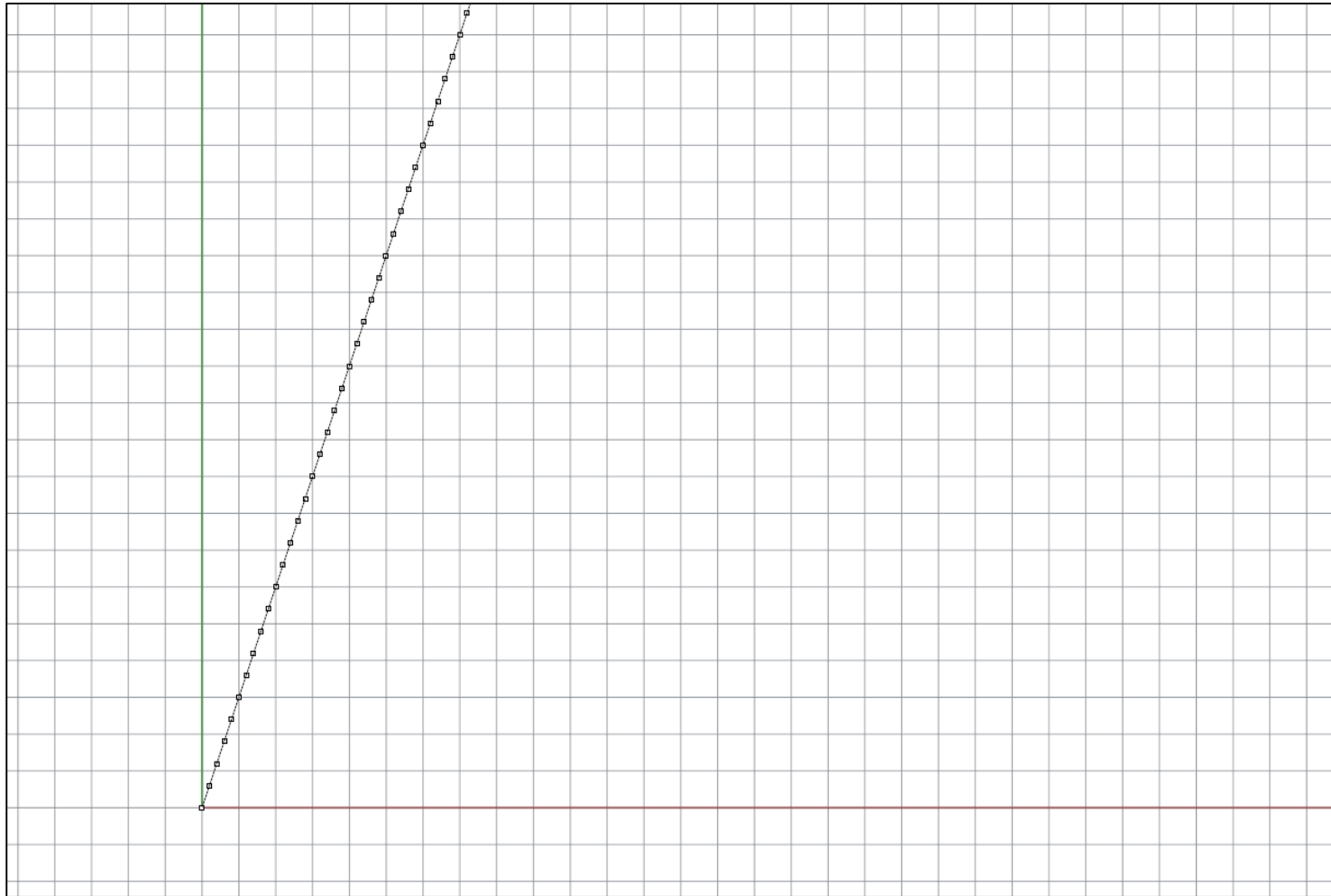
    endPoint = rs.VectorAdd(startPoint, moveVector)
    rs.AddLine(startPoint, endPoint)
    rs.AddPoint(startPoint)

    if (currentStep < maxStep):
        MoveRecursively(endPoint, moveVector, currentStep + 1, maxStep)

MoveRecursively( (0, 0, 0), (0.2, 0.6, 0), 0, 100 )
```

# Live Example: Creating Rhino geometry recursively

Step 1: Recursively create (continuous) line segments



# Live Example: Creating Rhino geometry recursively

Step 2: Slightly change the moveVector by a random amount at each step

```
import rhinoscriptsyntax as rs
import random

def MoveRecursively(startPoint, moveVector, currentStep, maxStep):

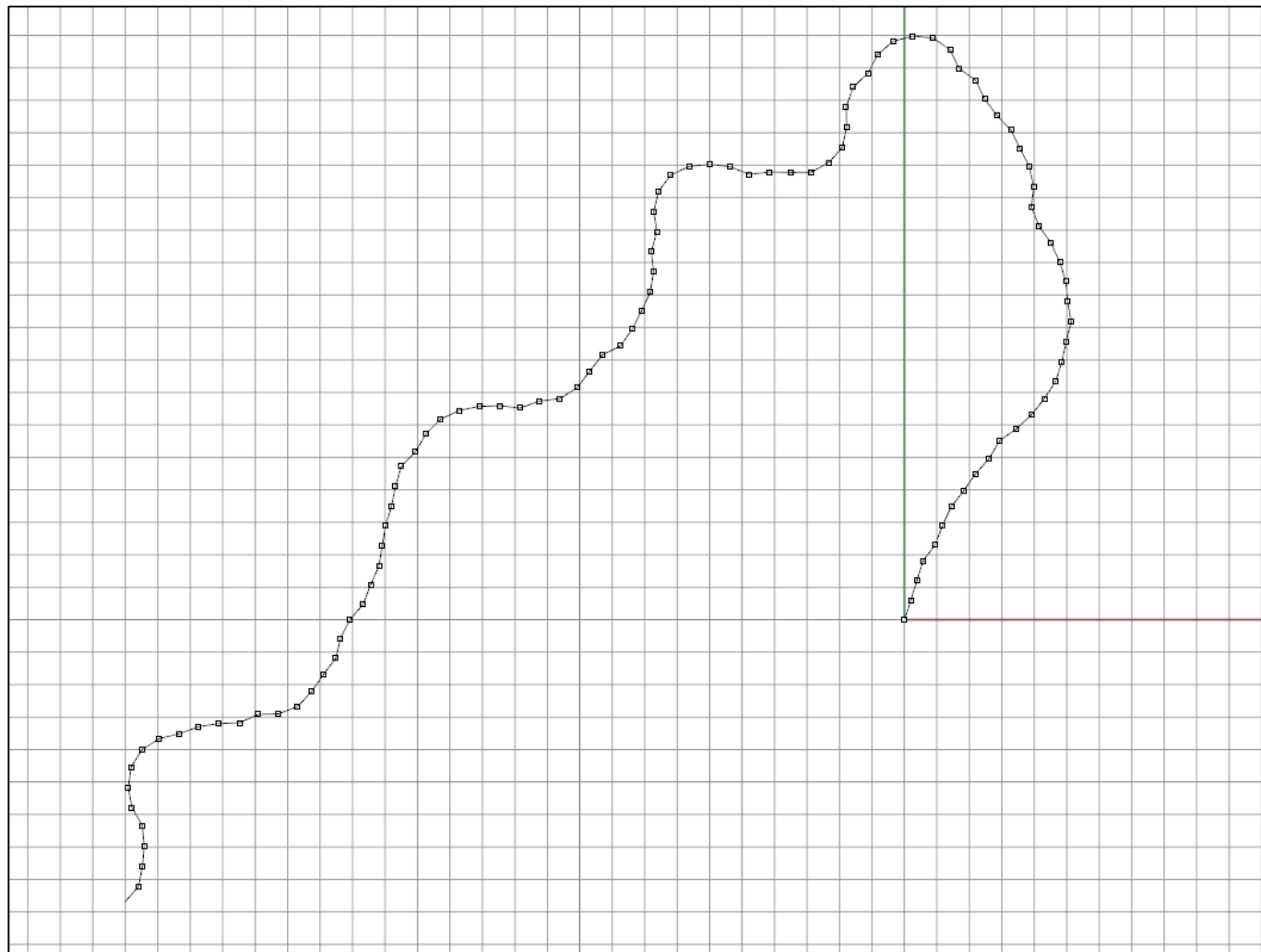
    endPoint = rs.VectorAdd(startPoint, moveVector)
    rs.AddLine(startPoint, endPoint)
    rs.AddPoint(startPoint)

    if (currentStep < maxStep):
        moveVector = rs.VectorRotate(moveVector, random.uniform(-30, 30), (0, 0, 1))
        MoveRecursively(endPoint, moveVector, currentStep + 1, maxStep)

MoveRecursively( (0, 0, 0), (0.2, 0.6, 0), 0, 100 )
```

# Live Example: Creating Rhino geometry recursively

Step 2: Slightly change the moveVector by a random amount at each step



# A typical pattern for writing recursive function

```
def MyRecursiveFunction(inputData)

    # Do something useful with inputData (e.g. draw geometries)
    ...

    # compute the input data for the next/recursive call
    nextInputData = ...

    # Make the recursive call
    MyRecursiveFunction(nextInputData)

    # extra computation, if there is any
    ...
```



# Recursion

## with branching

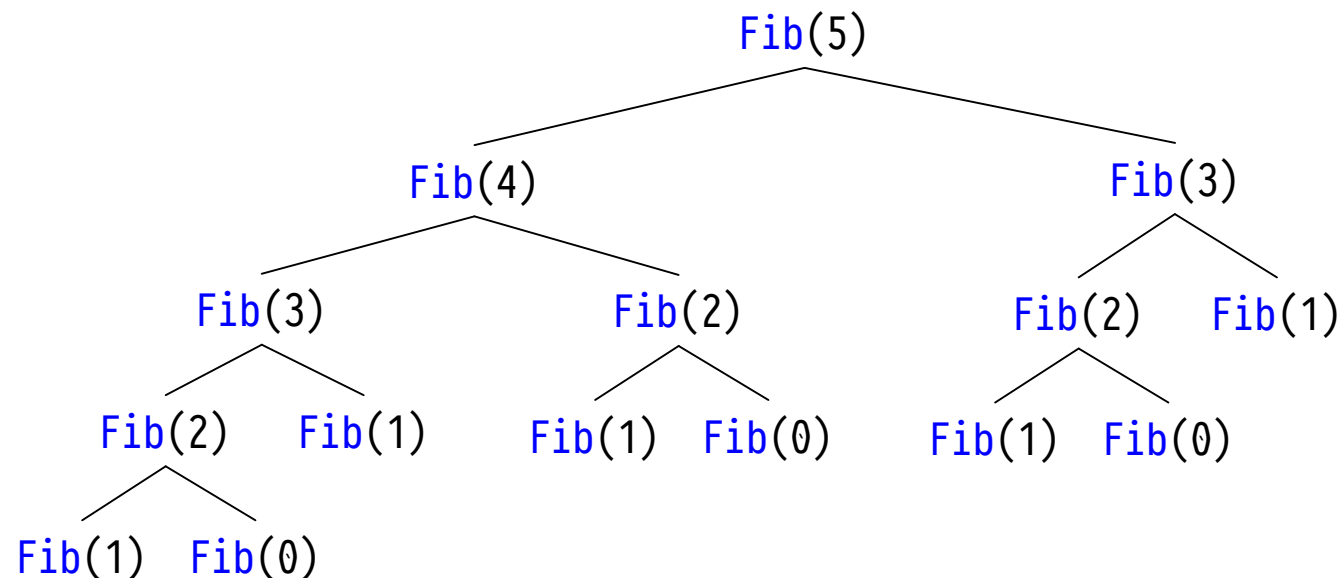
# Branching recursion

A function can call itself **more than once**

The Fibonacci number sequence: 0 1 1 2 3 5 8 13 21 ...

Recursive Fibonacci function

```
def Fib(n):  
    if n == 0:  
        return 0  
    elif n == 1:  
        return 1  
    else:  
        return Fib(n-1) + Fib(n-2)  
  
print(Fib(4))
```



Computation time is proportional to  $2^n$

# Branching recursion

Watchout for exponential growth !!!

Computing the 52<sup>th</sup> term in the sequence will require  $2^{50}$  recursive calls to the Fib function

Even if each call takes only 1 nanosecond to compute, the total time will be 13 days

Similarly the 66<sup>th</sup> term will take almost 300 years

... and the 99<sup>th</sup> term will take 19 billion years !!!

# A non-recursive version

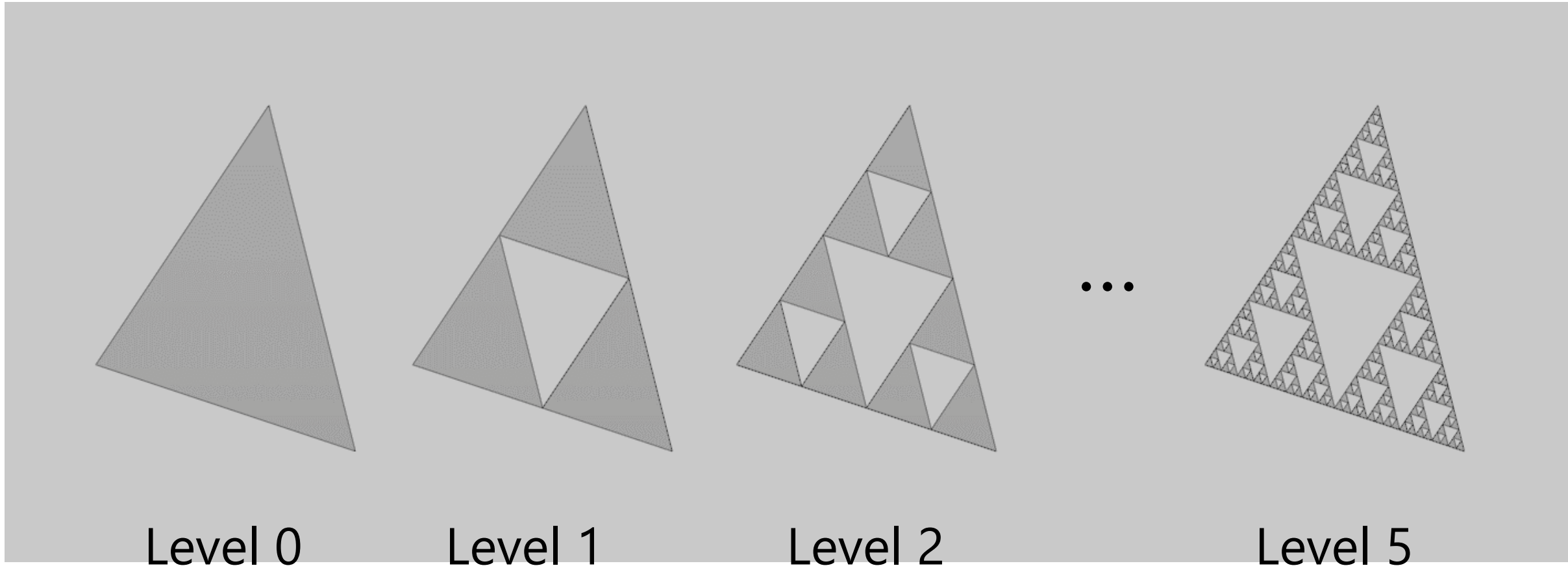
## Non-Recursive Fibonacci function

```
def Fib(n):  
    if n == 0:  
        return 0  
    elif n == 1:  
        return 1  
    else:  
        previous = 0  
        current = 1  
        for i in range(1, n):  
            next = current + previous  
            previous = current  
            current = next  
        return current
```

Computation time is  
proportional to  $n$

# Fractal Geometries

# Live Example: Sierpinski triangle



# Live Example: Sierpinski triangle

23

```
import Rhino.Geometry as rg
```

```
mesh = rg.Mesh()
```

```
def SubdivideTriangle(A, B, C, levelRemained):
```

```
    if levelRemained == 0:
```

```
        mesh.Vertices.Add(A)
```

```
        mesh.Vertices.Add(B)
```

```
        mesh.Vertices.Add(C)
```

```
        f = mesh.Vertices.Count
```

```
        mesh.Faces.AddFace(f-3, f-2, f-1)
```

```
    else:
```

```
        M = 0.5 * (A + B)
```

```
        N = 0.5 * (B + C)
```

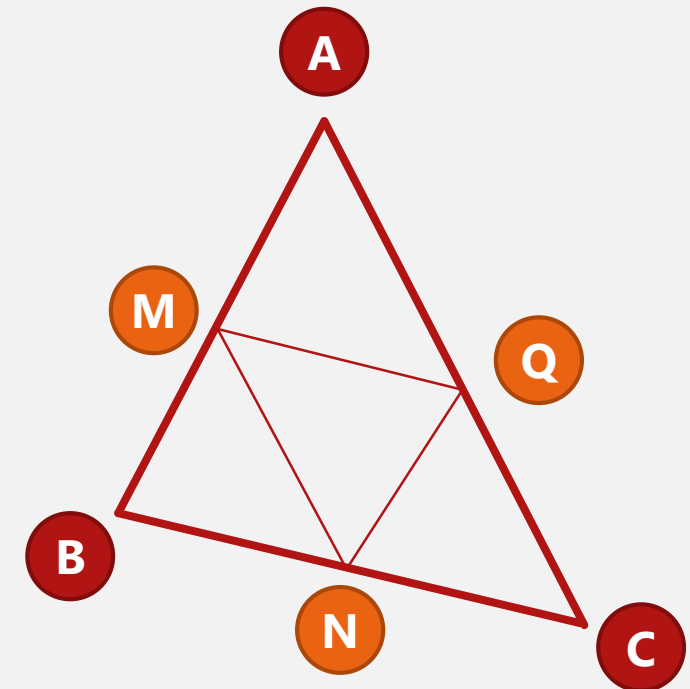
```
        Q = 0.5 * (C + A)
```

```
        SubdivideTriangle(A, M, Q, levelRemained - 1)
```

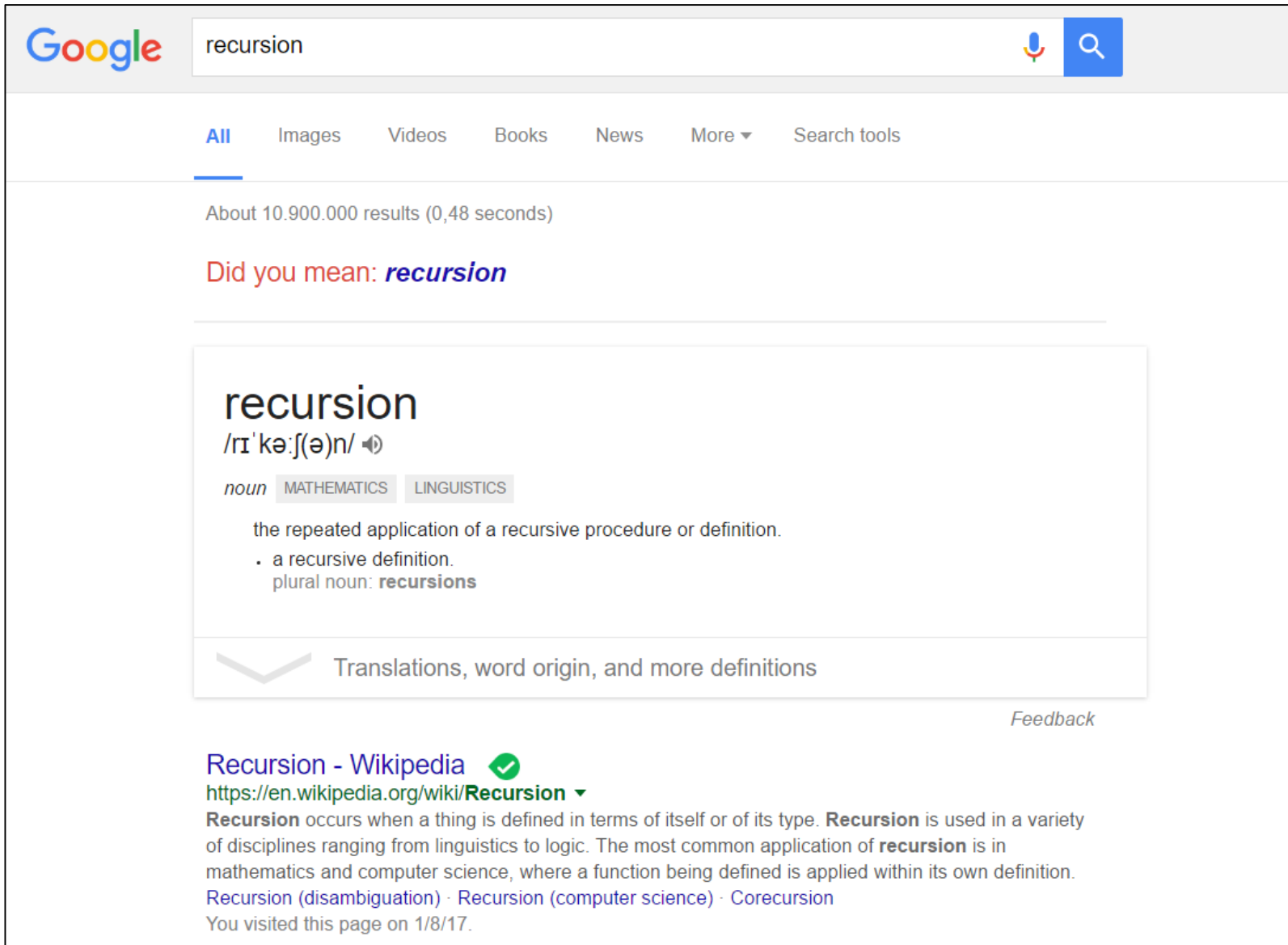
```
        SubdivideTriangle(M, B, N, levelRemained - 1)
```

```
        SubdivideTriangle(Q, N, C, levelRemained - 1)
```

```
SubdivideTriangle(rg.Point3d(2,4,0), rg.Point3d(0,1,0),  
                  rg.Point3d(3,0,0), 5 )
```



# A recursive joke from Google



The screenshot shows a Google search interface. The search bar contains the word "recursion". Below the search bar, the "All" tab is selected. The search results show "About 10.900.000 results (0,48 seconds)". A suggestion "Did you mean: *recursion*" is displayed. The main result is a definition of "recursion" with its phonetic transcription /rɪˈkʌːʃ(ə)n/. It is categorized as a noun in Mathematics and Linguistics. The definition is: "the repeated application of a recursive procedure or definition." followed by a bullet point: "• a recursive definition." and the plural noun "recursions". Below this, there is a section for "Translations, word origin, and more definitions" which is currently collapsed. At the bottom, there is a link to the Wikipedia page for "Recursion" with a green checkmark icon, followed by the URL "https://en.wikipedia.org/wiki/Recursion". Below the link, there is a paragraph explaining that recursion occurs when a thing is defined in terms of itself or of its type, and is used in a variety of disciplines ranging from linguistics to logic. The paragraph concludes that the most common application of recursion is in mathematics and computer science, where a function being defined is applied within its own definition. Below this paragraph, there are links for "Recursion (disambiguation)", "Recursion (computer science)", and "Corecursion". At the very bottom, it says "You visited this page on 1/8/17."

Google recursion

All Images Videos Books News More ▾ Search tools

About 10.900.000 results (0,48 seconds)

Did you mean: *recursion*

**recursion**  
/rɪˈkʌːʃ(ə)n/ 🔊

noun MATHEMATICS LINGUISTICS

the repeated application of a recursive procedure or definition.

- a recursive definition.

plural noun: **recursions**

Translations, word origin, and more definitions

Feedback

[Recursion - Wikipedia](https://en.wikipedia.org/wiki/Recursion) ✓  
<https://en.wikipedia.org/wiki/Recursion> ▾

**Recursion** occurs when a thing is defined in terms of itself or of its type. **Recursion** is used in a variety of disciplines ranging from linguistics to logic. The most common application of **recursion** is in mathematics and computer science, where a function being defined is applied within its own definition.

[Recursion \(disambiguation\)](#) · [Recursion \(computer science\)](#) · [Corecursion](#)

You visited this page on 1/8/17.



# Live Example: Tree growth

L-System invented by Lindenmayer (1968)

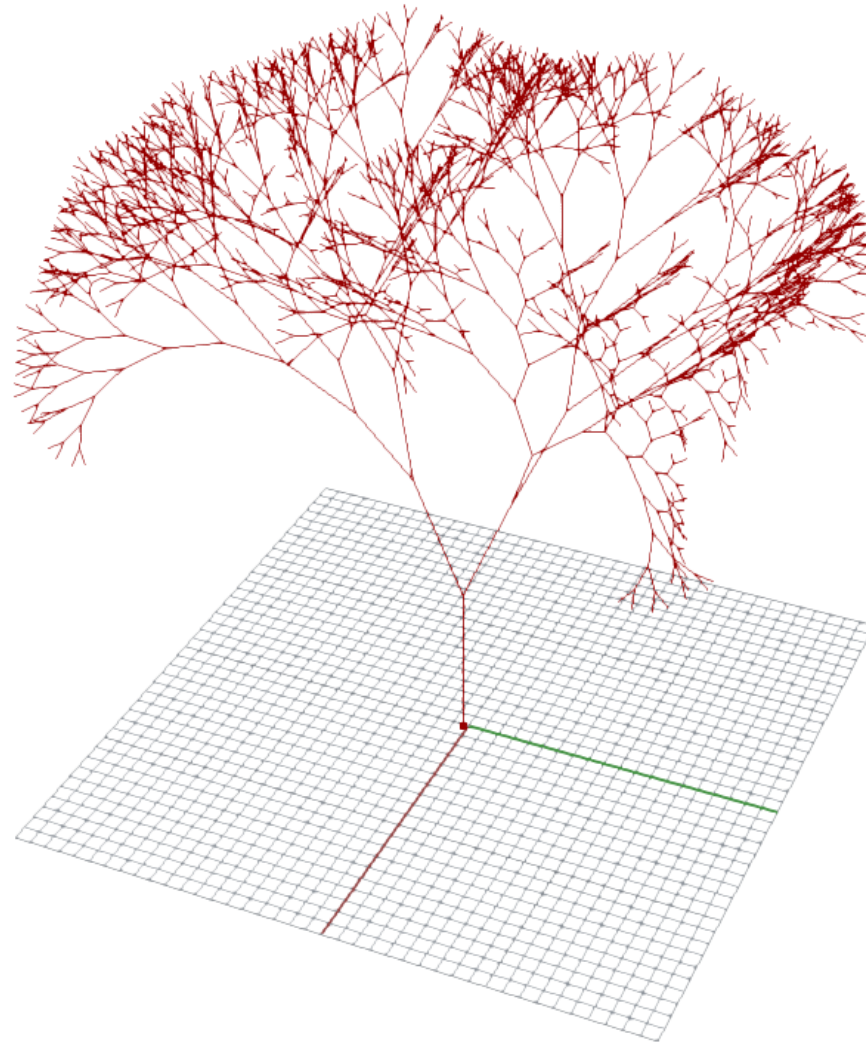
- A formal way to describe the growth of tree
- Based on the so-called "parallel rewriting system": essentially a set of recursive rules, optionally with adjustable parameters and some randomness
- Can realistically generate the shape (morphology) of many trees/plants



# Live Example: Tree growth

Step 1: Defining a branching recursive function

26



# Live Example: Tree growth

## Step 1: Defining a branching recursive function

27

```
import Rhino.Geometry as rg

branches = []

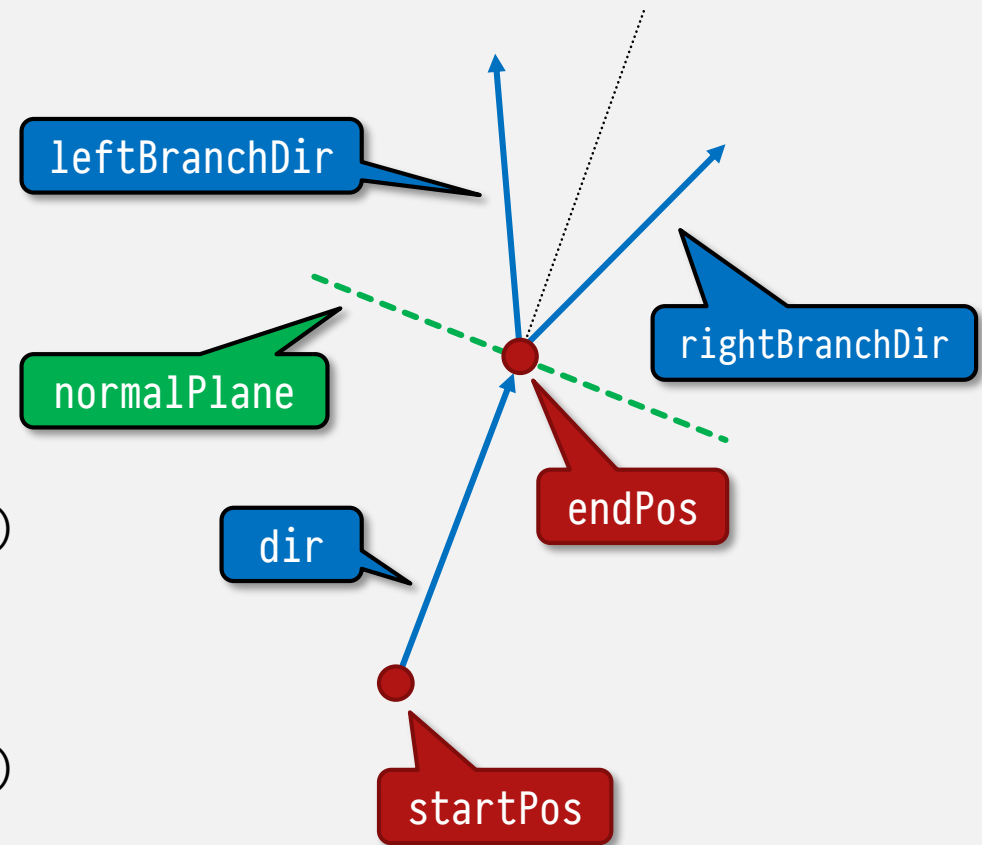
def Branch(startPos, dir, levelRemained):
    endPos = startPos + dir
    branches.append(rg.Line(startPos, endPos))

    if (levelRemained > 0):
        normalPlane = rg.Plane(endPos, dir)

        leftBranchDir = rg.Vector3d(dir)
        leftBranchDir.Rotate(-iSpreadAngle, normalPlane.XAxis)
        leftBranchDir *= iShrink
        Branch(endPos, leftBranchDir, levelRemained - 1)

        rightBranchDir = rg.Vector3d(dir)
        rightBranchDir.Rotate(iSpreadAngle, normalPlane.XAxis)
        rightBranchDir *= iShrink
        Branch(endPos, rightBranchDir, levelRemained - 1)


Branch(iRootPosition, iRootDirection, 10)
```



# Live Example: Tree growth

28

Step 2: Adding a random twist to the normal plane

```
def Branch(startPos, dir, levelRemained):  
    ...  
    if (levelRemained > 0):  
        normalPlane = rg.Plane(endPos, dir)  
         normalPlane = rg.Rotate(random.uniform(0, 3.14), normalPlane.ZAxis)  
        ...
```

# Live Example: Tree growth

Step 3: Adding random variations to the spread angles and shrink factor

```
import Rhino.Geometry as rg
import random

branches = []

def Branch(startPos, dir, levelRemained):
    ...

    if (levelRemained > 0):
        ...

        leftBranchDir = rg.Vector3d(dir)
        leftBranchDir.Rotate(-iSpreadAngle + random.uniform(0, iSpreadVariation), normalPlane.XAxis)
        leftBranchDir *= iShrink + random.uniform(0, iShrinkVariation)
        Branch(endPos, leftBranchDir, levelRemained - 1)

        rightBranchDir = rg.Vector3d(dir)
        rightBranchDir.Rotate(iSpreadAngle + random.uniform(0, iSpreadVariation), normalPlane.XAxis)
        rightBranchDir *= iShrink + random.uniform(0, iShrinkVariation)
        Branch(endPos, rightBranchDir, levelRemained - 1)

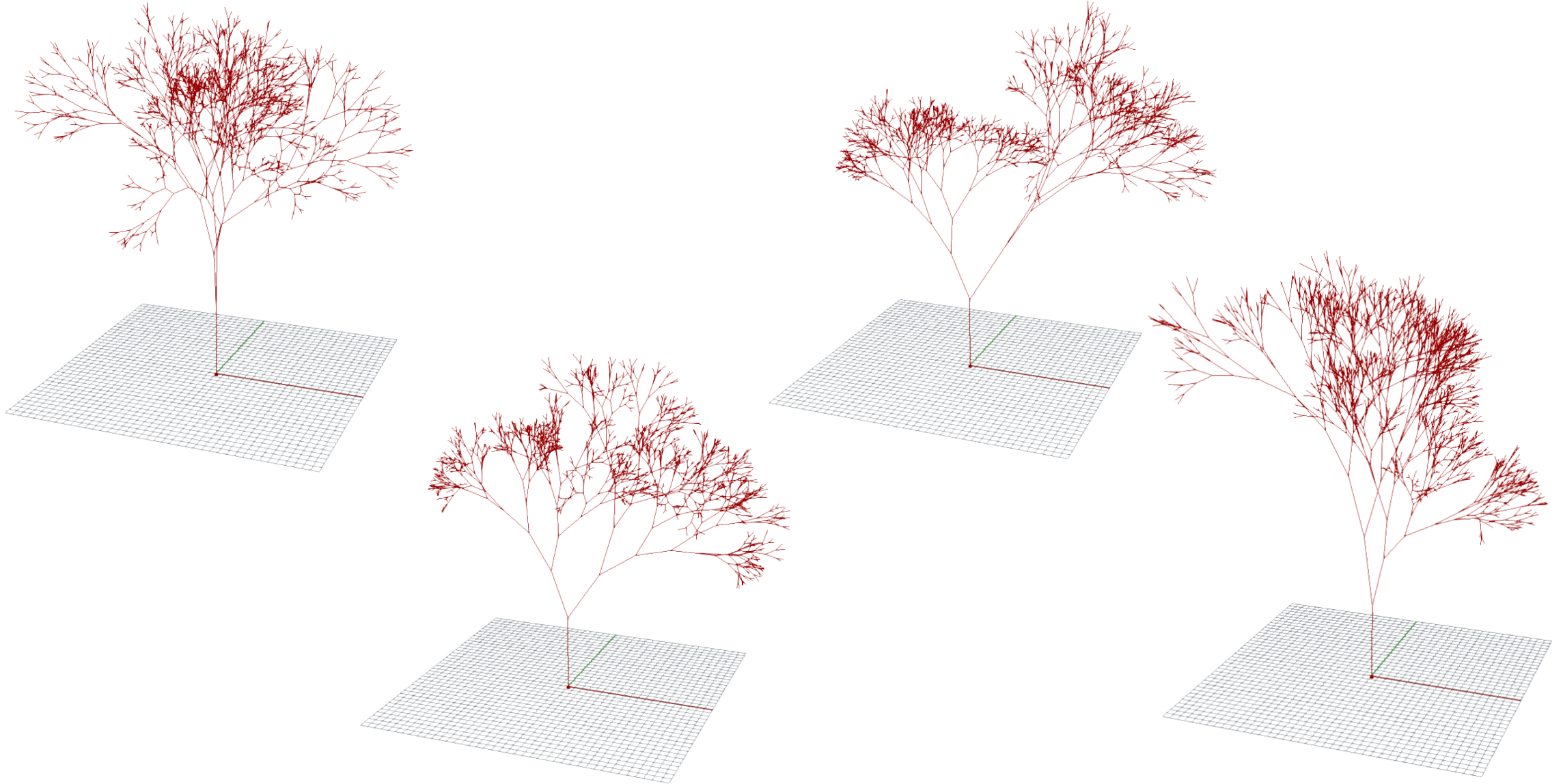
Branch(iRootPosition, iRootDirection, 10)
```



# Live Example: Tree growth

Step 3: Adding random variations to the spread angle and shrink factor

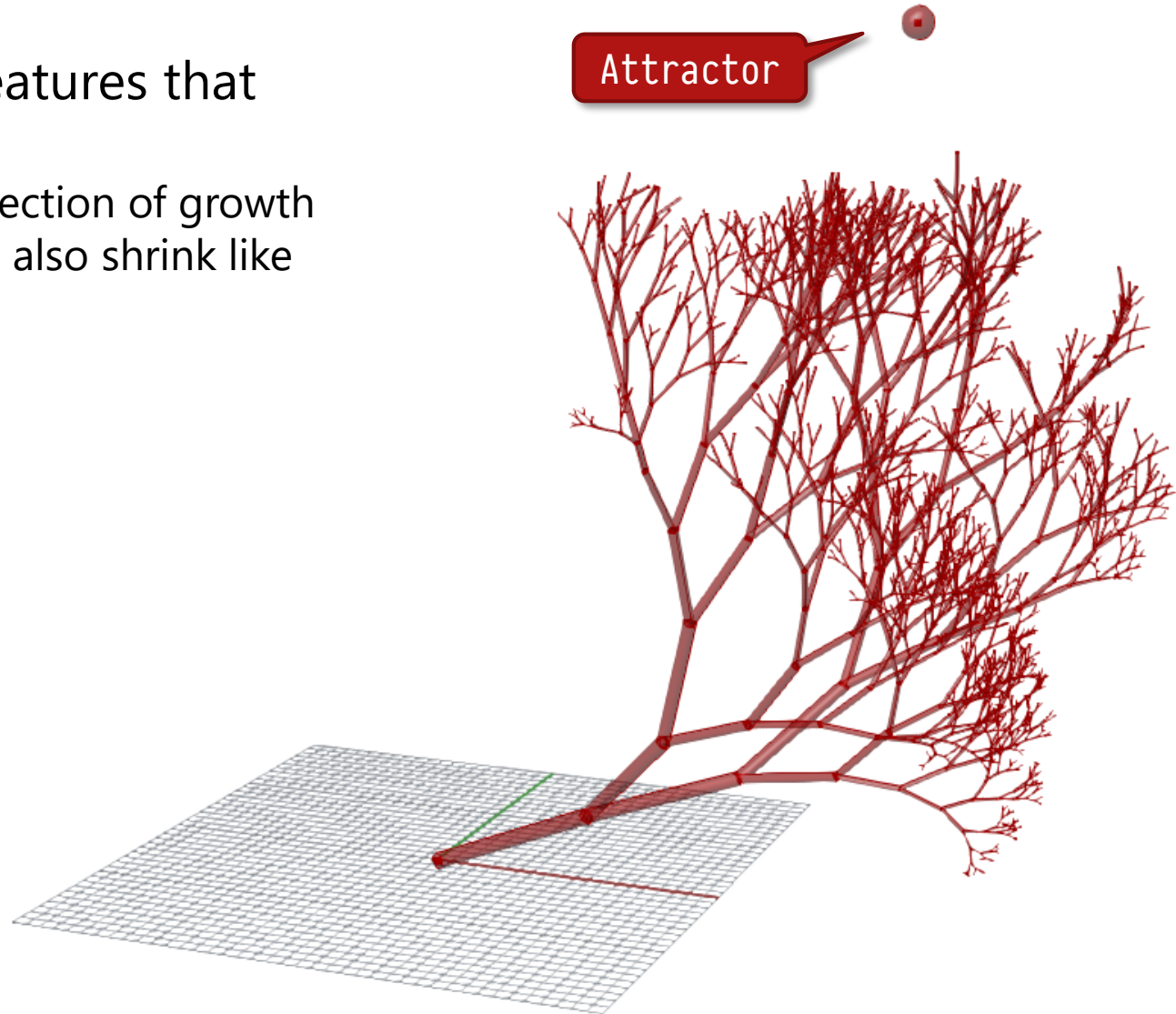
30



# Tree growth: let the fun go on !

There are so many interesting features that you can add:

- **Attractor** point that influence the direction of growth
- Branches with **thickness** (that should also shrink like the branch length)





# Tree growth: let the fun go on !

## Make a little forest

Each tree will have a different look thanks to the random variations

