Introduction to Generative Urban Models

Prof. Dr. Reinhard König
www.uni-weimar.de/computational-architecture
reinhard.koenig@uni-weimar.de
Winter Semester 2016

Computational Architecture

Junior Professor Dr. König Reinhard

Bauhaus-Universität Weimar

Chapter 01 // Introduction

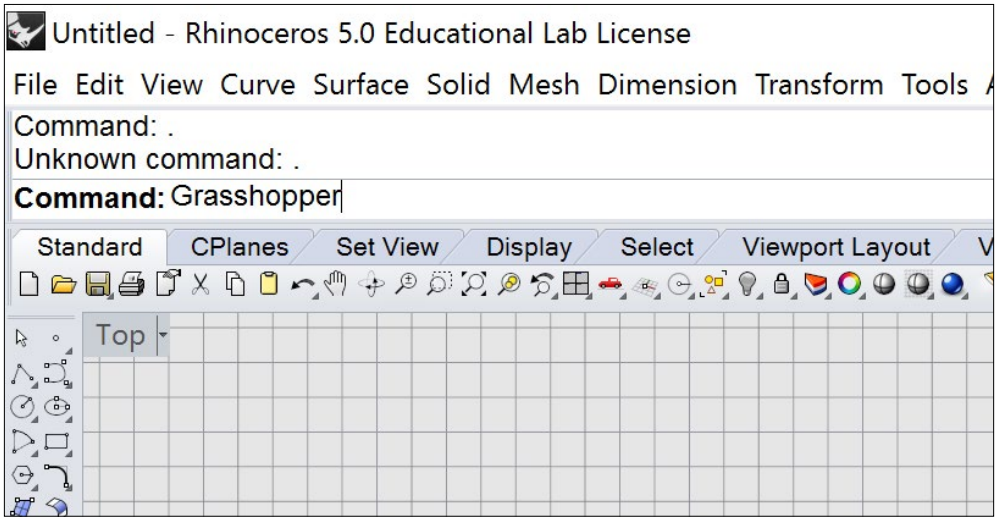
1. Introduction

This script accompanies the seminar Computational Urban Synthesis by thee Junior-Professorship Computational Architecture at the Bauhaus University Weimar. It helps you with your first steps into the world of programming.

As development environment we use the C# component in Grasshopper for Rhino3D. This allows us to link our code directly with other Grasshopper components and to work directly with the 3D capabilities of Rhino3D.

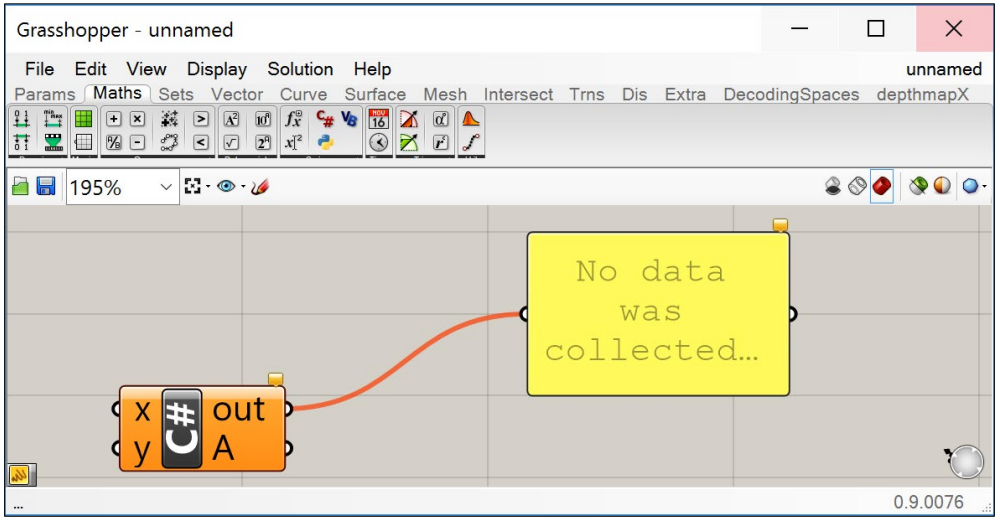
1. 1. Start Rhino and Grasshopper

We assume that you are familiar with the basic controls and functions in Rhino and Grasshopper. Just as a reminder, you start Grasshopper by typing “Grasshopper” into the Command line at the top right of the Rhino user interface (UI):

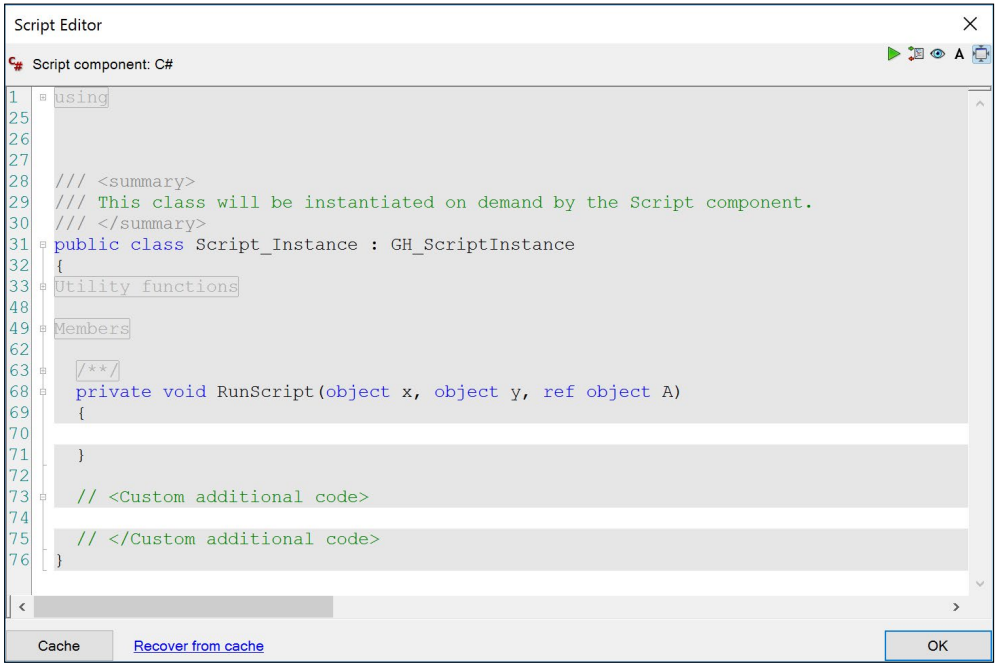


This command opens the Grasshopper UI. The C# component can be found at the Maths tab. Alternatively double click on the empty dashboard and enter “C#” and select the C# Script component. Additionally we need a Panel, which is connected to the out parameter of the C# component as shown in the following. The Panel is used later to show some output massages from our code.

With a double click on the C# component you open the integrated development environment (IDE), which we use to enter our code:



So far you managed the setup. Now let’s start with our first lines of code!



1. 2. Data types

Computer programs, including spreadsheets, text editors, calculators, or chat clients, work with data. Tools to work with various data types are essential part of a modern computer language. A data type is a set of values and the allowable operations on those values.

Data types are used everywhere in a programming language like C#. Because it's a strongly typed language, you are required to inform the compiler about which data types you wish to use every time you declare a variable. In this chapter we will take a look at some of the most used data types and how they work.

Boolean values

`bool` is one of the simplest data types. It can contain only 2 values - false or true. The `bool` type is important to understand when using logical operators like the if statement.

Strings

`string` is used for storing text, that is, a number of chars. In C#, strings are immutable, which means that strings are never changed after they have been created. When using methods which changes a string, the actual string is not changed - a new string is returned instead.

Integers

`int` is short for integer, a data type for storing numbers without decimals. When working with numbers, `int` is the most commonly used data type. Integers have several data types within C#, depending on the size of the number they are supposed to store.

Floating point numbers

`float` or `double` are one of the data types used to store numbers which may or may not contain decimals.

There is much more to say about data types and there are many more types in each category, but for the beginning it's fine if you remember the listed ones.

1. 3. Variables

A variable can be compared to a storage room, and is essential for the programmer. In C#, a variable is declared like this:

```
<data type> <name>;
```

An example could look like this:

```
string myName;
```

That's the most basic version. If we use the variable as a global variable, we have to assign a visibility to the variable, and perhaps assign a value to it at the same time. It can be done like this:

```
<visibility> <data type> <name> = <value>;
```

And with an example:

```
private string myName = "John Doe";
```

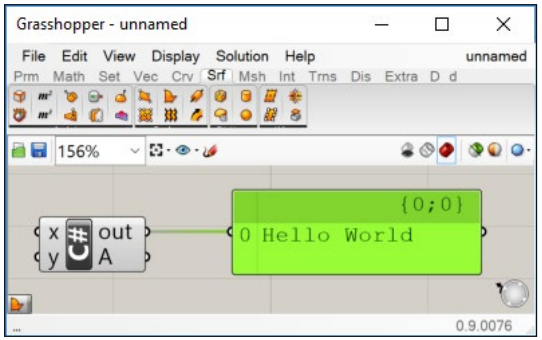
Now, we have enough theory to start with a first simple example.

1. 4. Hello World

In our first example we declare a variable and assign a value to it:

```
private void RunScript(object x, object y, ref object A)
  {
    string myString = "Hello World";
    Print(myString);
  }
```

With this example we also see the idea of the Panel, which shows the printed values. This means the `Print` command sends its values to the out parameter of the component. We also get all kinds of error messages to this out parameter and our panel.



This value in our example is the string “Hello World”, which is send to the print command and finally shown on the panel. Of cause this in not a really exiting example and there is no geometry created in Rhino. Therefore next we look into an example to use a simple parametrized line geometry.

1. 5. Create lines

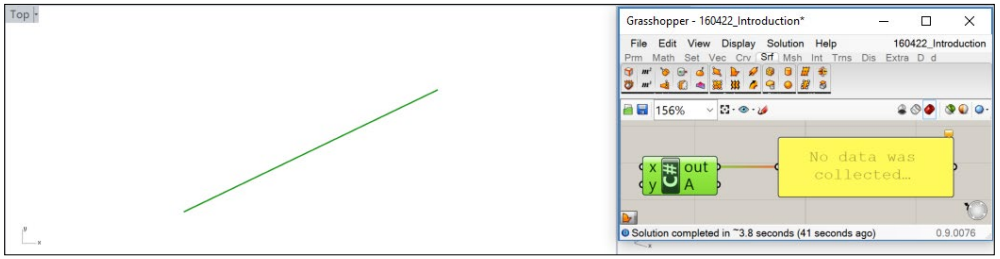
In the next example we create our first Rhino geometry from our C# component. The following code create a line from a starting point `fromPt` to a destination point `toPt`. The line itself is represented by the variable `myLine` in line number 8. It is initialized by using the two points as arguments. In this code example the two lines of our first example are deactivated (comment out) by the `//` at the beginning of a line of code:

```
1       private void RunScript(object x, object y, ref object A)
2       {
3           //string myString = “Hello World”;
4           //Print(myString);
5
6           Point3d fromPt = new Point3d(0, 0, 0);
7           Point3d toPt = new Point3d(25, 12, 0);
8           Rhino.Geometry.Line myLine = new Line(fromPt, toPt);
9           A = myLine;
10      }
```

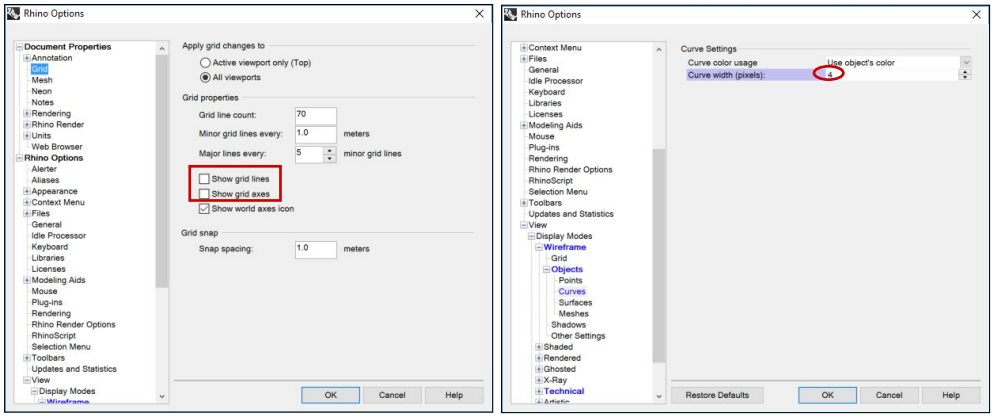
The created line is only drawn to Rhino after we assign it to the output parameter `A` in line 9.

We introduced in this example classes or objects, which comes from Rhino. The Point3d is an object from the namespace `Rhino.Geometry`, which you can find in the using list at the top of our C# IDE component. If we include the namespace in the using, we don’t need to write it in front of the object we want to use. Whereas in line 8 you can see that we used the namespace. Delete the `.Line` after `Rhino.Geometry.` and add a new point after `Rhino.Geometry`. This opens a list of optional classes that are offered by `. Rhino.Geometry`. This list of possible features is called IntelliSense.

After clicking the OK button in the C# editor you should see your first line in Rhino! If you change now the coordinates for the `fromPt` and `toPt`, the line is drawn correspondingly. This is the core idea of parametric modeling.



In the preset of Rhino you have a gray background with a grid. You may change this to a white background without grid as I’ve done it for the examples. Therefore go to *View / Display Options* and look into the *Grid* options of the dialog. You can also increase the line width (for all lines in the display) in this dialog box for each display mode. We use Wireframe and change the widths for the Objects Curves to 4, so that we see a thicker line in the display.



In a next step we can use the parameters `x` an `y`, which are the predefined input parameters for our C# component. Therefore we change the `toPt` declaration as follows:

```
Point3d toPt = new Point3d(x, y, 0);
```

If you click OK, you receive an error message in the panel. This is because we need to define the input parameters of the C# components first. This can be done by right click on the `x` and `y` parameter and selecting *Type hint / double*. The simplest way to assign a parameter value is by right click on he `x` and `y` parameter of the component again and entering a value at *Set*

Data Item. If you set 25 for x and 12 for y we get the same line as in the beginning. You can also add a Slider for each input parameter for a more comfortable control of the values.

1. 6. Experiment!

The best way for starting your exploration in the world of coding is to change existing code. Therefore try to play around with different Rhino objects and parameter settings! In this section we can only help you with your first steps, but to learn how to walk confident in this world depends on your engagement.

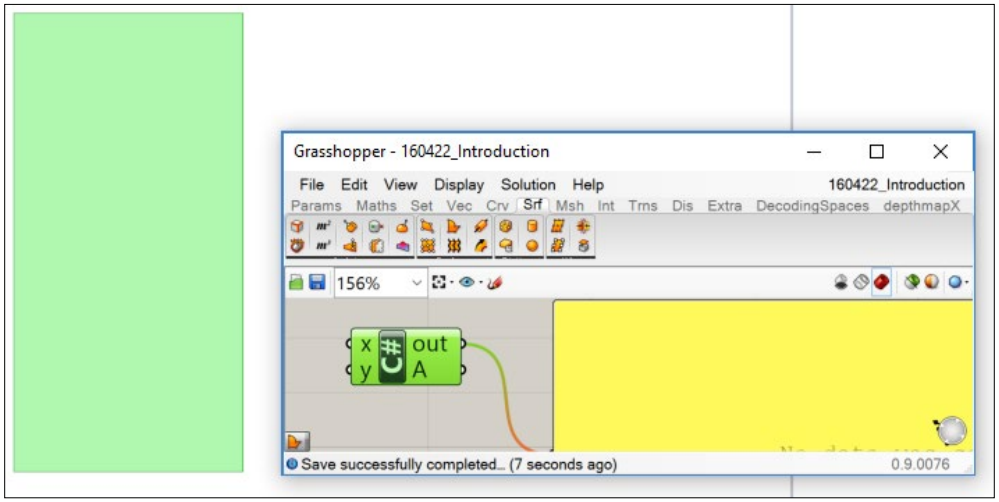
The following example draws a rectangle on a defined plane in Rhino:

```
1   private void RunScript(double x, double y, ref object A)
2     {
3       /*
4       Point3d fromPt = new Point3d(0, 0, 0);
5       Point3d toPt = new Point3d(x, y, 0);
6       Rhino.Geometry.Line myLine = new Line(fromPt, toPt);
7       A = myLine;
8       */
9
10      // -- Define the drawing plane
11      Rhino.Geometry.Point3d firstPt = new Point3d(1, 0, 0);
12      Rhino.Geometry.Point3d secondPt = new Point3d(0, 1, 0);
13      Rhino.Geometry.Point3d origPt = new Point3d(0, 0, 0);
14      Rhino.Geometry.Plane myPlane = new Plane(origPt, firstPt,
        secondPt);
15
16      // -- create a rectangle
17      double width = 10;
18      double length = 20;
19      Rhino.Geometry.Rectangle3d myRect = new Rectangle3d(myPlane,
        width, length);
20
21      // -- convert the recangle to a nurbs and create a planar Brep
        Surface in Rhino
22      NurbsCurve myCurve = myRect.ToNurbsCurve();
23      Brep myBrepSurface = Rhino.Geometry.Brep.
        CreatePlanarBreps(myCurve)[0];
24      A = myBrepSurface;//myRect;
25    }
```

You find new comments elements in the code example. Our old code is out commented by using a slash-star star-slash syntax `/* ... */`. Using this the code from line 3 to line 8 is deactivated.

Another important, but often forgotten element are comments in your code, which explains what happens. In the example you find them in line 10, 16, and 21. You can't add enough of them. Belief me, you will forget what your code means very soon and wish you would have added more comments!

From line 11 we define a plane `myPlane` on which the rectangle is drawn. The rectangle `myRect` itself is created from line 17, where we use separate parameters for `width` and `length`, which are used to create the rectangle in line 19. The code from line 22 is used to make a surface `myCurve` out of our rectangle `myRect` so that we get a better visualization. In line 24 we send the created surface to the output parameter A of our component. You should see a simple rectangular surface in your Rhino views:



For the ones who are familiar with Grasshopper: We could also output the `myRect` rectangle and use the *Boundary Surfaces* component to create a surface out of it. But the idea of this introduction is to learn how things can be done by code.

1. 7. Loops

So far we have not touched one of the most powerful aspects of programming: Looping, the ability to do repetitions in an efficient and elegant way. In C#, looping can be done in 4 different variants, and we will have a look at each one of them.

The while loop

The while loop is probably the most simple one, so we will start with that. The while loop simply executes a block of code as long as the condition you give it is true:

```
private void RunScript(double x, double y, ref object A)
  {
    int number = 0;

    while(number < 5)
    {
      Print(number.ToString());
      number = number + 1;
    }
  }
```

You will get a nice listing of numbers, from 0 to 4. The number is first defined as 0, and each time the code in the loop is executed, it's incremented by one. But why does it only get to 4, when the code says 5? For the condition to return true, the number has to be less than 5, which in this case means that the code which outputs the number is not reached once the number is equal to 5. This is because the condition of the while loop is evaluated before it enters the code block.

The do loop

The opposite is true for the do loop, which works like the while loop in other aspects through. The do loop evaluates the condition after the loop has executed, which makes sure that the code block is always executed at least once.

```
private void RunScript(double x, double y, ref object A)
  {
    int number = 0;

    do
    {
      Print(number.ToString());
      number = number + 1;
    } while(number < 5);
  }
```

The output is the same though - once the number is more than 5, the loop is exited.

The for loop

The for loop is a bit different. It's preferred when you know how many iterations you want, either because you know the exact amount of iterations, or because you have a variable containing the amount. Here is an example on the for loop.

```
private void RunScript(double x, double y, ref object A)
  {
    int number = 5;

    for(int i = 0; i < number; i++)
    {
      Print(i.ToString());
    }
  }
```

This produces the exact same output, but as you can see, the for loop is a bit more compact. It consists of 3 parts - we initialize a variable for counting, set up a conditional statement to test it, and increment the counter (++ means the same as "variable = variable + 1").

The first part, where we define the i variable and set it to 0, is only executed once, before the loop starts. The last 2 parts are executed for each iteration of the loop. Each time, i is compared to our number variable - if i is smaller than number, the loop runs one more time. After that, i is increased by one.

Try running the program, and afterwards, try changing the number variable to something bigger or smaller than 5. You will see the loop respond to the change.

The foreach loop

The last loop we will look at, is the `foreach` loop. It operates on collections of items, for instance arrays or other built-in list types. In our example we will use one of the simple lists, called an `List`.

```
private void RunScript(double x, double y, ref object A)
{
    List<string> list = new List<string>();
    list.Add("First string");
    list.Add("Second string");
    list.Add("Third string");

    foreach(string name in list)
    {
       Print(name);
    }
}
```

1. 8. Geometry in the Loop

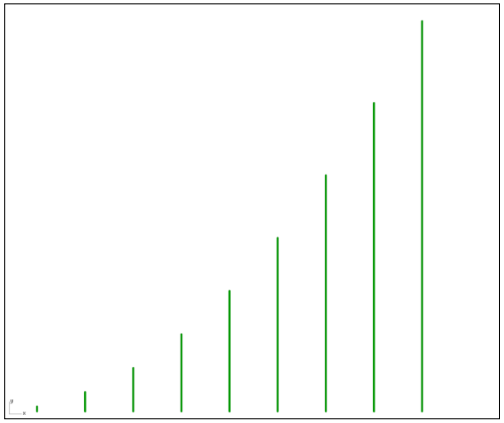
We create an instance of a List that can contain data of the type `string`, and then we add some string items to it. We use the `foreach` loop to run through each item, setting the `name` variable to the item we have reached each time. That way, we have a named variable to print to our panel.

When working with collections, you are very likely to be using the `foreach` loop most of the time, mainly because it's simpler than any of the other loops for these kind of operations.

Okay, now you know how loops work. Let's use it for drawing! We use a for loop to repeat the creation of a line and change the coordinates by using the variable `i` that is incremented per iteration. The created lines are collected by a list, which can contain Line objects `List<Line>`. This list is returned to the output parameter `A` at the end:

```
1   private void RunScript(double x, double y, ref object A)
2   {
3     int iterations = 10;
4
5     // -- create a bunch of lines
6     List<Line> collectLines = new List<Line>();
7     for(int i = 1; i < iterations; i++)
8     {
9       // -- create a line in Rhino
10      Point3d fromPt = new Point3d(i, 0, 0);
11      Point3d toPt = new Point3d(i, i*(i/10.0), 0);
12      Line myLine = new Line(fromPt, toPt);
13      collectLines.Add(myLine);
14    }
15
16    A = collectLines;
17  }
```

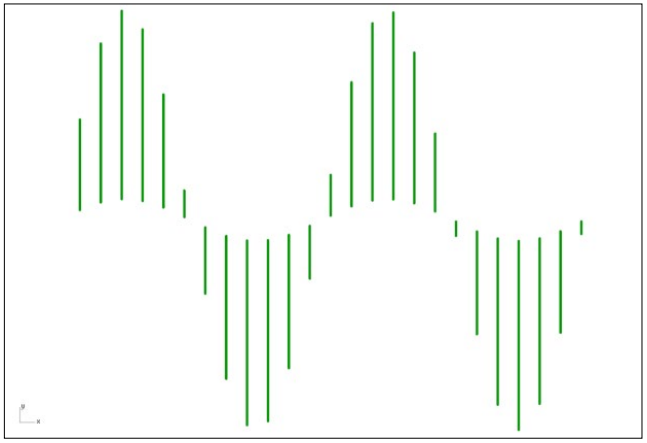
Be aware that it is important to use the explicit floating point number 10.0 in line 11. If you use 10 only, the resulting number is rounded to an `int` and you wont see any line. The code above draw this lines:



Play around with this code! For example, change the number of iterations and the way, how `i` is used for defining the start and end points of the line. An important class for calculations is `Math`. Exchange lines 10 and 11 by the following:

```
Point3d fromPt = new Point3d(i, Math.Sin(i / 2.0), 0);
Point3d toPt = new Point3d(i, Math.Sin(i / 2.0) * 10, 0);
```

The result will change to a sinus wave:

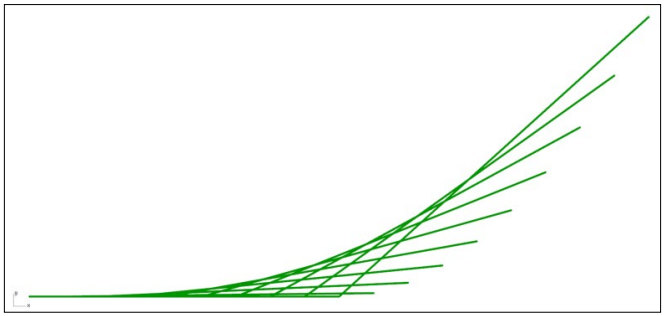


You can access items in a list and change them. In the following we change the end point of all lines. Therefore we replace the point by a new one:

```
for(int i = 0; i < collectLines.Count; i++)
{
  Line curLine = collectLines[i];
  Point3d tempPt = curLine.To;
  tempPt.X += 9;
  curLine.To = tempPt;
  collectLines[i] = curLine;
}
```

```
A = collectLines;
```

We could use the foreach loop to iterate through the list, but if we want to change items in the collection as we do it here, the forech loop don't allow changes of its items. The result of our small experiment looks like this:

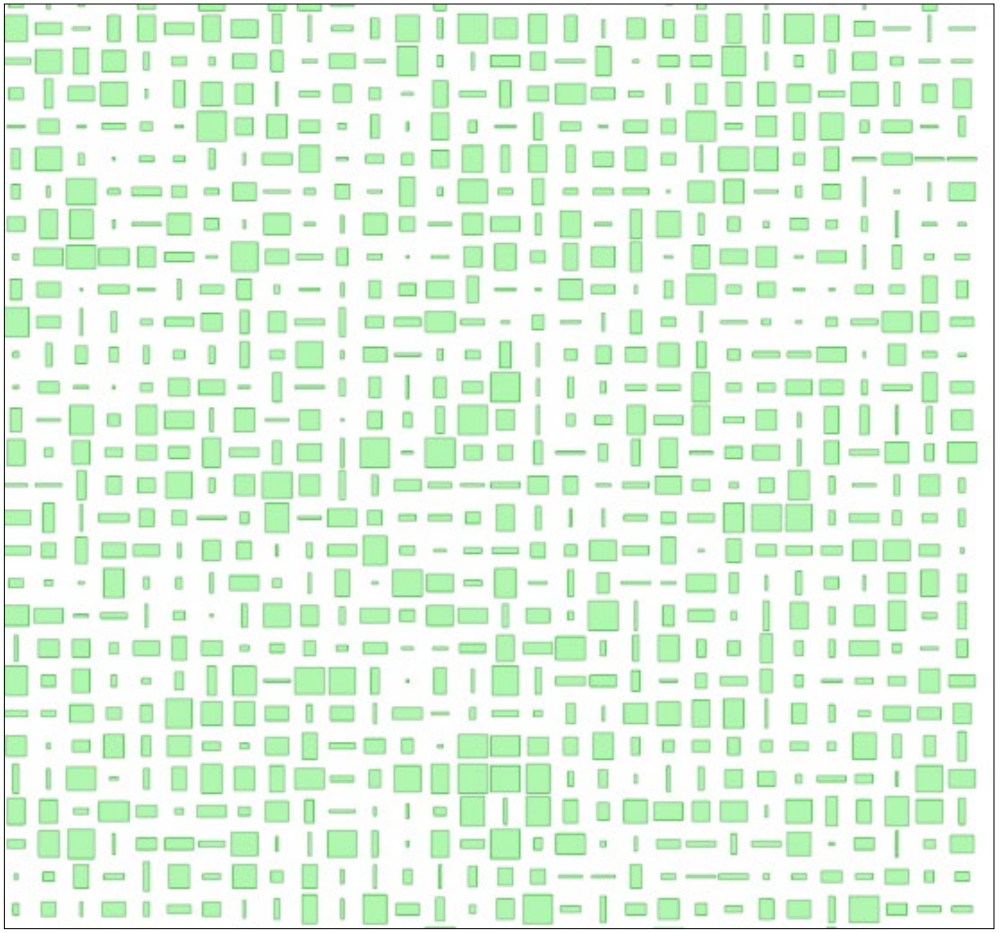


1. 9. Nested Loops

The concept of a loop can be extended to nested loops by placing one loop into the other. We also use a random function in the following example:

```
1   private void RunScript(int nr, double y, ref object A)
2   {
3     int iterations = nr;
4     Random rnd = new Random();
5     List<Brep> collectBreps = new List<Brep>();
6
7     // -- Define the drawing plane
8     Point3d firstPt = new Point3d(1, 0, 0);
9     Point3d secondPt = new Point3d(0, 1, 0);
10    Point3d origPt = new Point3d(0, 0, 0);
11    Plane myPlane = new Plane(origPt, firstPt, secondPt);
12
13    // -- first loop used for x coordinates
14    for(int i = 0; i < iterations; i++)
15    {
16      // -- second loop used for y coordinates
17      for(int k = 0; k < iterations; k++)
18      {
19        // -- create a rectangle
20        int maxRnd = 10;
21        double width = 1.0 + rnd.Next(0, maxRnd);
22        double heigth = 1.0 + rnd.Next(0, maxRnd);
23        Point3d refPt = new Point3d(i * (maxRnd + 1.1),
              k * (maxRnd + 1.1), 0);
24        Point3d cornerA = new Point3d(refPt.X - (0.5 * width),
              refPt.Y - (0.5 * heigth), 0);
25        Point3d cornerB = new Point3d(refPt.X + (0.5 * width),
              refPt.Y + (0.5 * heigth), 0);
26
27        Rectangle3d myRect = new Rectangle3d(myPlane, cornerA, cornerB);
28
29        // -- convert the recangle to a nurbs and create a planar Brep
               Surface in Rhino
30        NurbsCurve myCurve = myRect.ToNurbsCurve();
31        Brep myBrepSurface = Brep.CreatePlanarBreps(myCurve)[0];
32        collectBreps.Add(myBrepSurface);
33      }
34    }
35
36    A = collectBreps;
37  }
```

The second, nested loop is added in line 17. This means that per iteration of the outer loop in line 17, all iterations of the inner loop are executed. Inside the inner loop we draw a rectangle with random width and height values. Therefore we use the `Random` class to declare the `rnd` variable in line 4. The function `rnd.Next(1, 10)` returns a random integer value between 1 and 10. As a result we get a grid of randomly sized rectangles:



It's time for your own variations!

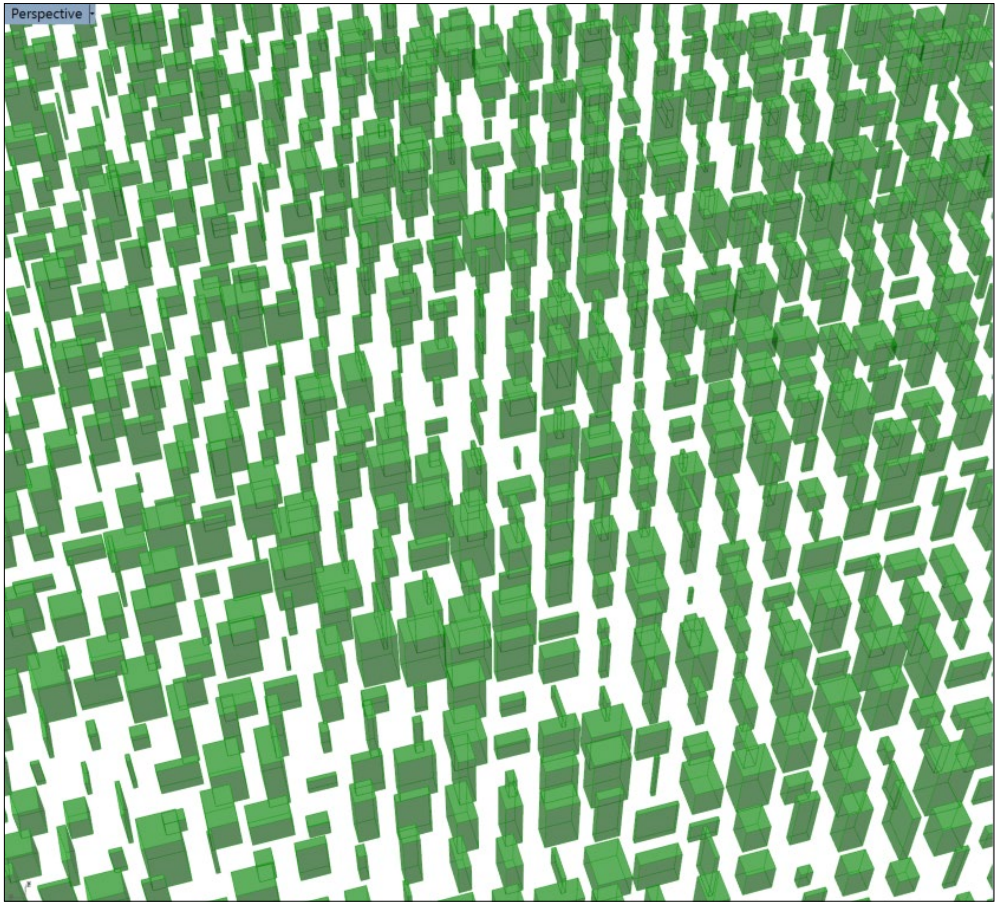
1. 10. Functions

To define your own function is very useful if the code becomes more complex. Functions allows us to combine logical code parts into one separate entity. In the following we use a variant of the last example, where we put the code to create a surface `CreateRandomSurface` in a function, which returns the created surface. In the main `RunScript` procedure our new function is called from the inner loop in line 12:

```
1    private void RunScript(int nr, double y, ref object A)
2    {
3      int iterations = nr;
4      List<Extrusion> collectBreps = new List<Extrusion>();
6      // -- first loop used for x coordinates
7      for(int i = 0; i < iterations; i++)
8      {
9        // -- second loop used for y coordinates
10       for(int k = 0; k < iterations; k++)
11       {
12         Extrusion myExtrudedObject = CreateRandomSurface(i, k);
13         collectBreps.Add(myExtrudedObject);
14       }
15     }
17     A = collectBreps;
18   }
19
20   // <Custom additional code>
21   private Random rnd = new Random();
22
23   private Extrusion CreateRandomSurface(int i, int k)
24   {
25     // -- create a rectangle
26     int maxRnd = 10;
27     double width = 1.0 + rnd.Next(0, maxRnd);
28     double heigth = 1.0 + rnd.Next(0, maxRnd);
29     Point3d refPt = new Point3d(i * (maxRnd + 1.1),
             k * (maxRnd + 1.1), 0);
30     Point3d cornerA = new Point3d(refPt.X - (0.5 * width),
             refPt.Y - (0.5 * heigth), 0);
31     Point3d cornerB = new Point3d(refPt.X + (0.5 * width),
             refPt.Y + (0.5 * heigth), 0);
33     Rectangle3d myRect = new Rectangle3d(Plane.WorldXY, cornerA,
             cornerB);
34     myRect.Transform(Transform.Rotation(0.3, myRect.Center));
35
```

```
36      // -- convert the recangle to a nurbs and create a planar Brep
                Surface in Rhino
37      NurbsCurve myCurve = myRect.ToNurbsCurve();
38      Rhino.Geometry.Extrusion myExtrusion = Extrusion.Create(myCurve,
                rnd.Next(5, 20), true);
39      return myExtrusion;
40    }
```

We changed the created object from a planar surface to an extruded rectangle. Therefore we use the `Extrusion` class from `Rhino.Geometry`. Additionally we introduce the `Transform` method, which is used to rotate our rectangle: `myRect.Transform(Transform.Rotation(0.3, myRect.Center));` There are many other transformations available for your explorations! Here is the result of our example code above:



1. 11. Conditions - The if statement

One of the single most important statements in every programming language is the if statement. Being able to set up conditional blocks of code is a fundamental principal of writing software. In C#, the if statement is very simple to use. The if statement needs a boolean result, that is, true or false. We start with a very simple example:

```
private void RunScript(int nr, ref object A)
 {
   int value = 10 / 2;
   if (value == 5)
   {
     Print(value.ToString() + " = 5 :: true");
   }
   else
   {
     Print(value.ToString() + " = 5 :: false");
   }
 }
```

We make a basic calculation and compare the resulting `value` if it's 5 or not. The program is branching into the first bracket, if the `if` condition is true and it goes into the `else` branch otherwise. Remember that we use int variables, so that in case you change the calculation to `int value = 11 / 2;` the condition will still be true, since the result is rounded.

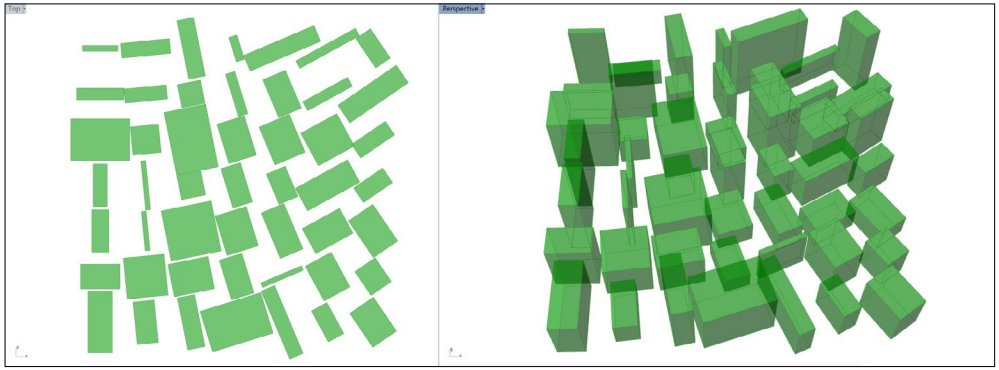
A more complex example follows on the last pages.

Growing Rectangles

Finally we combine everything we have learned so far into one example. We create a grid of rectangles again, but this time we grow the rectangles until they intersect with another one. With the first part of the code we are familiar with. The section where we grow the rectangles starts at line 20. We use some additional lists to manage the growth process. In line 26 a `do` loop starts, which is executed until all rectangles are grown to a size that they would intersect with another one if they grow another step.

Since there is no direct copy or duplicate function for `Rectangle3d`, we used a `GH_Rectangle`. Both are very similar, but the second one is implemented in the Grasshopper namespace. The copied rectangle `copyRect` is now scaled by the factor 1.2 in line 35.

Afterwards we test if the existing rectangles intersect with the new `copyRect`. Therefore we use a `foreach` look in line 40. The conditional check in line 42 ensures that we don't compare the same rectangles with each other. The intersection computation is done in line 44 and returns an `events` variable of the type `Rhino.Geometry.Intersect.CurveIntersections`. We just use it to test if there is an intersection in line 45.



```
1   private void RunScript(int nr, ref object A)
2   {
3     int iterations = nr;
4     List<Extrusion> collectBreps = new List<Extrusion>();
5     List<Rectangle3d> collectRectangles = new List<Rectangle3d>();
6
7     // -- first loop used for x coordinates
8     for(int i = 0; i < iterations; i++)
9     {
10      // -- second loop used for y coordinates
11      for(int k = 0; k < iterations; k++)
12      {
13        // -- create the rectangles
14        Rectangle3d myRectangle = CreateRectangles(i, k);
15        collectRectangles.Add(myRectangle);
16      }
17    }
18
19    // -- grow rectangles
20    List<Rectangle3d> growCadidates = new
            List<Rectangle3d>(collectRectangles); // copy our list
21    List<Rectangle3d> toRemoveRects = new List<Rectangle3d>();
22    List<Rectangle3d> grownRectangles = new List<Rectangle3d>();
23    int counter = 0; // security counter to avoid endless looing!
24    int maxIterations = 10;
25
26    do{
27      for(int i = 0; i < growCadidates.Count; i++)
28      {
29        Rectangle3d curRect = growCadidates[i];
30        // -- bit complicate copy of our rectangle...
31        GH_Rectangle ghRect = new GH_Rectangle(curRect);
32        Rectangle3d copyRect;
33        ghRect.CastTo(out copyRect);
34        // -- scale the rectangle
35        copyRect.Transform(Transform.Scale(copyRect.Center, 1.2));
36
37        // -- check for intersection
38        bool intersect = false;
39        int origID = -1;
40        foreach(Rectangle3d curTestRect in collectRectangles)
41        {
42          if (curTestRect.Center != curRect.Center)
43          {
44            var events = Rhino.Geometry.Intersect.Intersection.
                  CurveCurve(curTestRect.ToNurbsCurve(), copyRect.
                  ToNurbsCurve(), 0.001, 0);
```

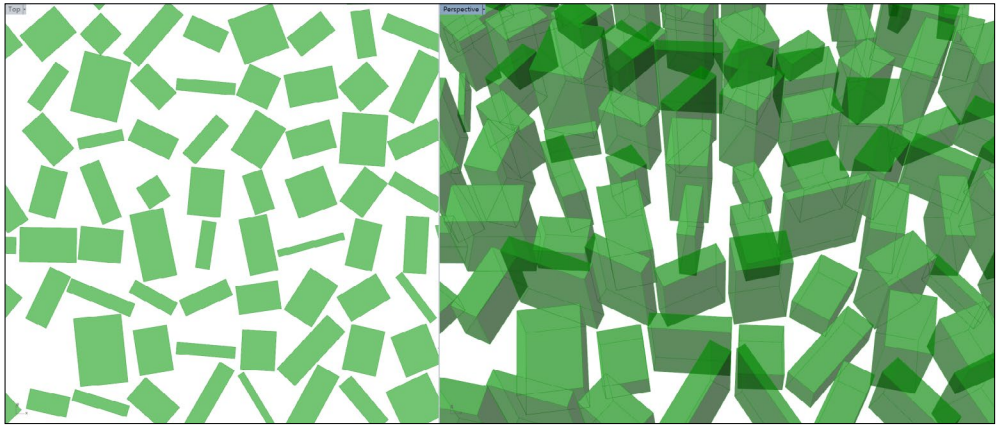
From line 55 we implement the rules, what to do if there is an intersection or not. In the list `toRemoveRects` we collect all rectangles which have reached their maximum size and shall be removed from the `growCadidates` list after the for loop. It is important to know that we can not remove elements from a list which we currently iterate through. Therefore the elements are removed afterwards in the `foreach` loop starting at line 70.

The assignment of the `copyRect` with the new size to the original list `collectRectangles` in line 65 is important for comparing the new sized rectangles with the further growing ones in the next iteration.

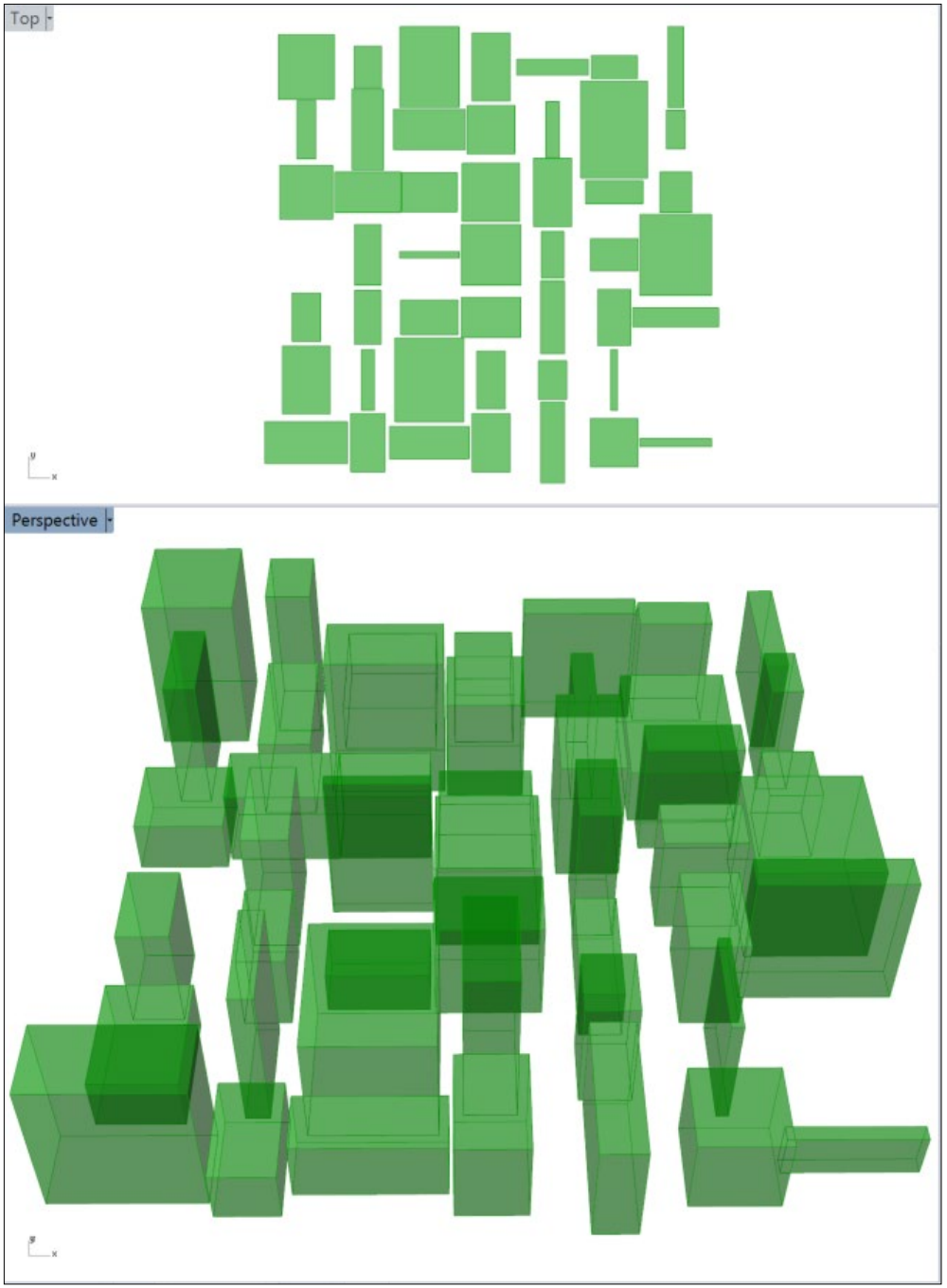
The do loop is repeated until the condition `while(growCadidates.Count > 0 && counter < maxIterations);` is met. This means we repeat the loop as long as all items from the `growCadidates` list are removed or until a counter reaches a certain maximum value. The counter is just used for security to avoid endless looping in case the `growCadidates` list will never be empty for some reasons.

With the rest of the code you are already familiar with. The resulting geometry is shown in the figure on the next page.

Now it's time for your explorations!



```
45            if (events.Count > 0)
46            {
47              intersect = true;
48            }
49          }
50          else // -- remember the index of the original rectangle
51            origID = collectRectangles.IndexOf(curTestRect);
52        }
53
54        // -- decide what to do if there is an intersection
55        if (intersect)
56        {
57          Print("intersect");
58          toRemoveRects.Add(curRect);  // if it can't grow further,
                                            remove element
59          grownRectangles.Add(curRect);// add element to the final list
60        }
61        else
62        {
63          Print("grow on");
64          growCadidates[i] = copyRect; // assign new size for
                                            intersection test
65          collectRectangles[origID] = copyRect; // assign new size for
                                                     imtersection test
66        }
67      }
68
69      // -- remove rectangles which can not grow anymore
70      foreach(Rectangle3d delRect in toRemoveRects)
71      {
72        growCadidates.Remove(delRect);
73      }
74
75      Print("iteration " + counter.ToString());
76      counter++;
77    } while(growCadidates.Count > 0 && counter < maxIterations);
78
79    // -- extrude the rectangles
80    foreach(Rectangle3d curRect in grownRectangles)
81    {
82      // -- convert the recangle to a nurbs and create a planar Brep
            Surface in Rhino
83      NurbsCurve myCurve = curRect.ToNurbsCurve();
84      Rhino.Geometry.Extrusion myExtrusion = Extrusion.Create(myCurve,
            rnd.Next(5, 20), true);
85      collectBreps.Add(myExtrusion);
86    }
```




```
87
88      A = collectBreps;
89    }
90
91    // <Custom additional code>
92    private Random rnd = new Random();
93
94    private Rectangle3d CreateRectangles(int i, int k)
95    {
96      // -- create a rectangle
97      int maxRnd = 10;
98      double width = 1.0 + rnd.Next(0, maxRnd);
99      double heigth = 1.0 + rnd.Next(0, maxRnd);
100     Point3d refPt = new Point3d(i * (maxRnd + 1.1),
              k * (maxRnd + 1.1), 0);
101     Point3d cornerA = new Point3d(refPt.X - (0.5 * width),
              refPt.Y - (0.5 * heigth), 0);
102     Point3d cornerB = new Point3d(refPt.X + (0.5 * width),
              refPt.Y + (0.5 * heigth), 0);
103
104     Rectangle3d myRect = new Rectangle3d(Plane.WorldXY,
              cornerA, cornerB);
105     return myRect;
106   }
```