

A Guide to the TurtleGraphics Package for R

A. Cena, M. Gagolewski, M. Kosiński,
N. Potocka, B. Żogała-Siudem



Contents

1	The TurtleGraphics Package Introduction	2
2	Installation And Usage of The Package	2
2.1	Installation of the Package	2
2.2	The Basics	2
2.2.1	Moving the Turtle	2
2.2.2	Additional Options	5
3	Introduction to R	12
4	Advanced Usage of the Package	14
4.1	Examples	14
4.1.1	Random Lines	15
4.1.2	The Spiral	16
4.1.3	The Turtle Rainbow Star	17
4.1.4	The Turtle Brownian	18
4.1.5	The Fractal Tree	19
4.1.6	The Koch Snowflake	21
4.1.7	The Sierpinski Triangle	22

1 The TurtleGraphics Package Introduction

The TurtleGraphics package offers to R-users functionality of the "turtle graphics" from Logo educational programming language. The main idea standing behind it is to encourage the children to learn programming and show that working with computer can be fun and creative.

The TurtleGraphics package allows to create either simple or more sophisticated graphics on the basis of lines. The main idea is that the Turtle, described by its location and orientation, moves with commands that are relative to its own position. The line that it leaves behind can be controlled, by disabling it or by setting its color and type.

The TurtleGraphics package offers functions to move forward or backward a given distance and to turn the Turtle in a chosen direction. The graphical parameter of the plot, for example the color, type or visibility of the line, can also be easily changed.

We strongly encourage you to try it yourself. Enjoy and have fun!

2 Installation And Usage of The Package

2.1 Installation of the Package

To install the package TurtleGraphics you should use following instructions.

```
> install_package("TurtleGraphics")
```

Then you have to load the package with the `require()` function, as it is shown below.

```
> require("TurtleGraphics")
```

2.2 The Basics

2.2.1 Moving the Turtle

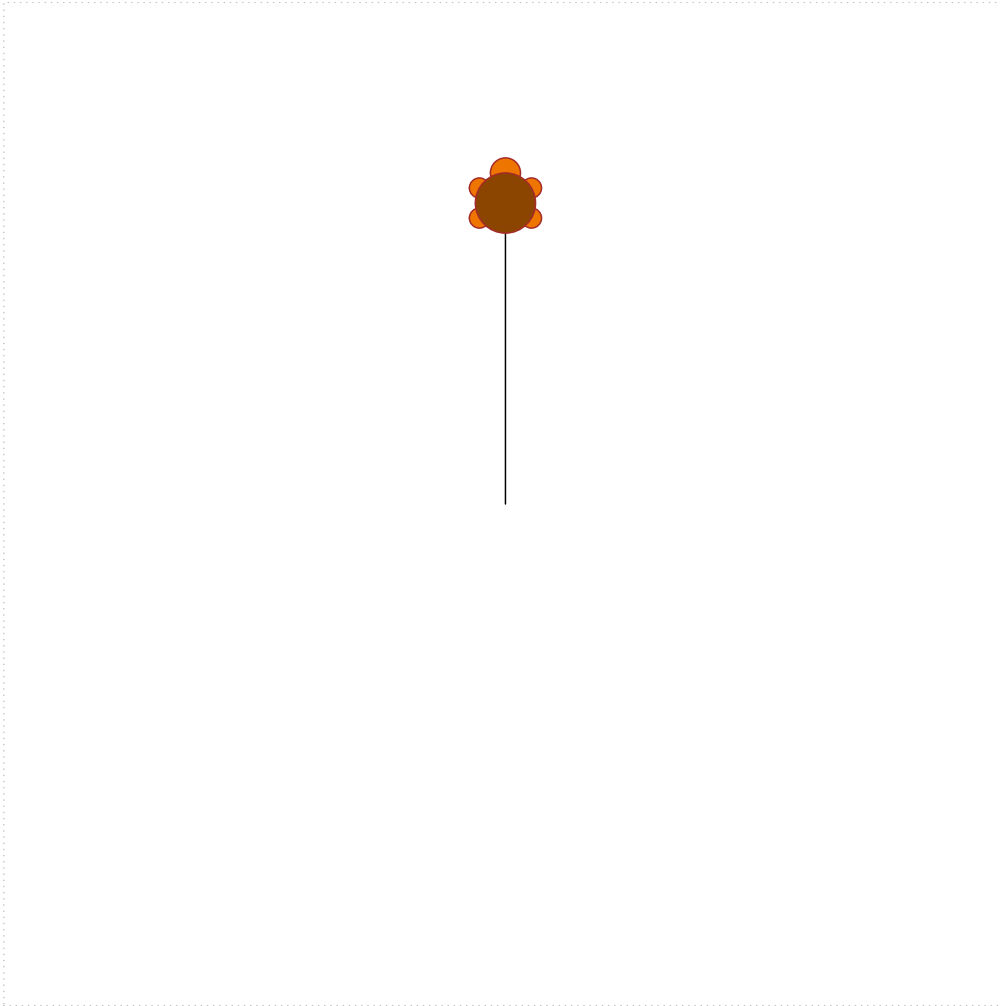
To start using the program call the `turtle_init()` function. It creates a plot region (sometimes called "Terrarium") and places the Turtle in the middle pointing north.

```
> turtle_init()
```

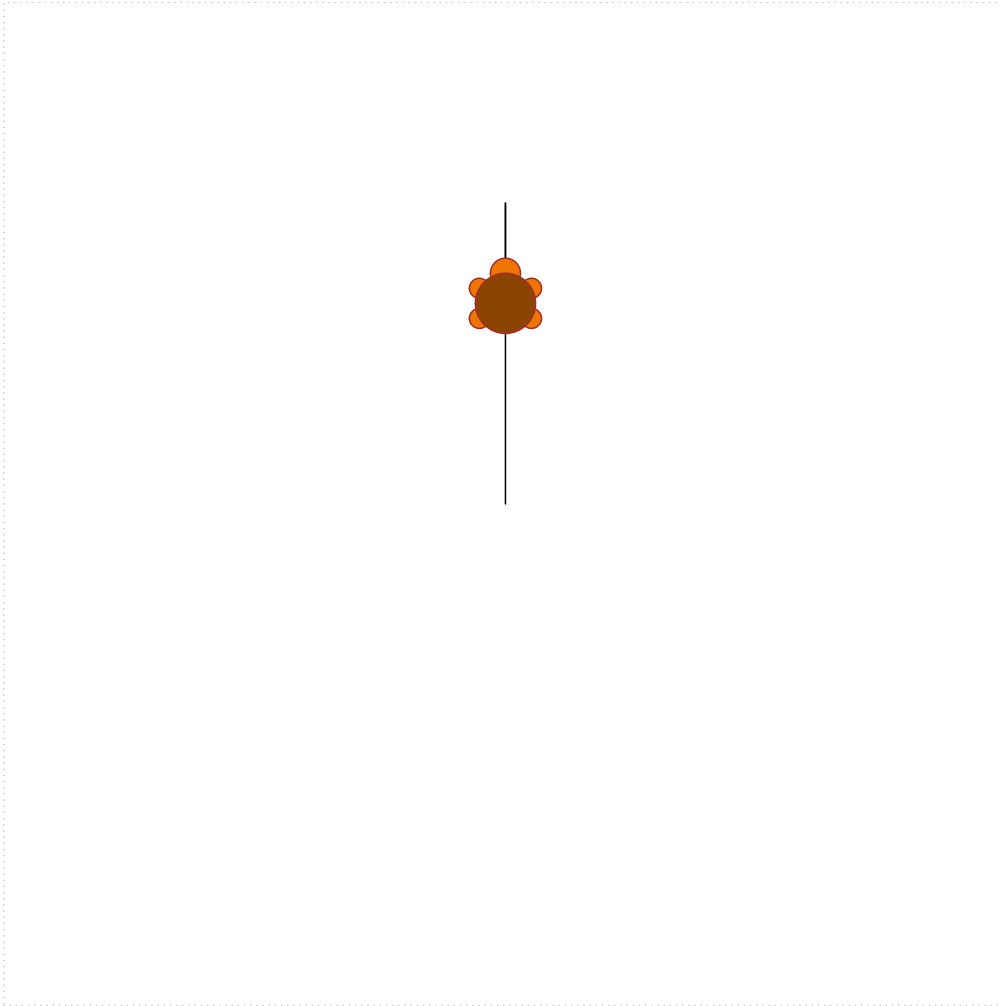
By default its size is 100 by 100 units. You can easily change it by passing as the arguments `width` and `height` (e.g. `turtle_init(width=200, height=200)`). There is one more argument that you can pass to this function. It defines the mode of the Terrarium and what happens if the Turtle moves outside the plot region. If you choose the "clip" option, which is default, then the Turtle can go outside the board but it (the Turtle) will not be seen. The "error" option does not let the Turtle out of the Terrarium and if you try to it will return an error. The third option – "cycle" – makes the Turtle come on the other side of the board in case of crossing its boarder.

There are two main group of functions used to move the Turtle.

The first one consists of the `turtle_forward()` and the `turtle_backward()` functions. In its argument you have to give the distance you desire the Turtle to move. For example, to move the Turtle forward for a distance of 10 units use the `turtle_forward()` function. To move the Turtle backwards you can use either the `turtle_forward()` function with the negative number as an argument or simply use the `turtle_backward()` function.

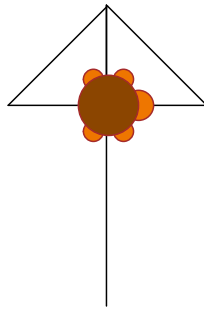


```
> turtle_init()  
> turtle_forward(dist=30)
```



```
> turtle_backward(dist=10)
```

The other tool that helps to move the Turtle are the `turtle_left` and the `turtle_right` functions. They change the Turtle's direction by a given angle.

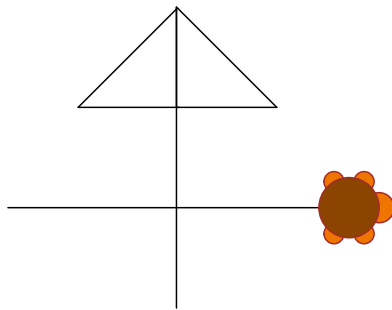


```
> turtle_right(angle=90)
> turtle_forward(dist=10)
> turtle_left(angle=135)
> turtle_forward(dist=14)
> turtle_left(angle=90)
> turtle_forward(dist=14)
> turtle_left(angle=135)
> turtle_forward(dist=10)
```

2.2.2 Additional Options

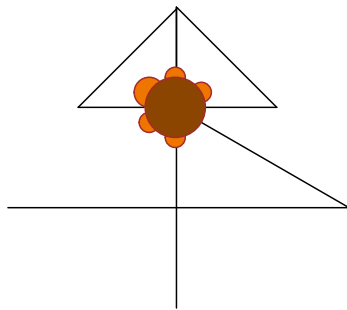
Now you know the basics and it is mostly it. Not very complicated, is it? Don't worry though, there are some additional parameters that you can play with.

To disable the path from being drawn you can simply use the `turtle_up()` function. Let's consider a simple example. Turn the Turtle to the right by 90 degrees and then use the `turtle_up()` function. Now, when you move forward the path is not visible. If you want the path to be drawn again you should call the `turtle_down()` function.



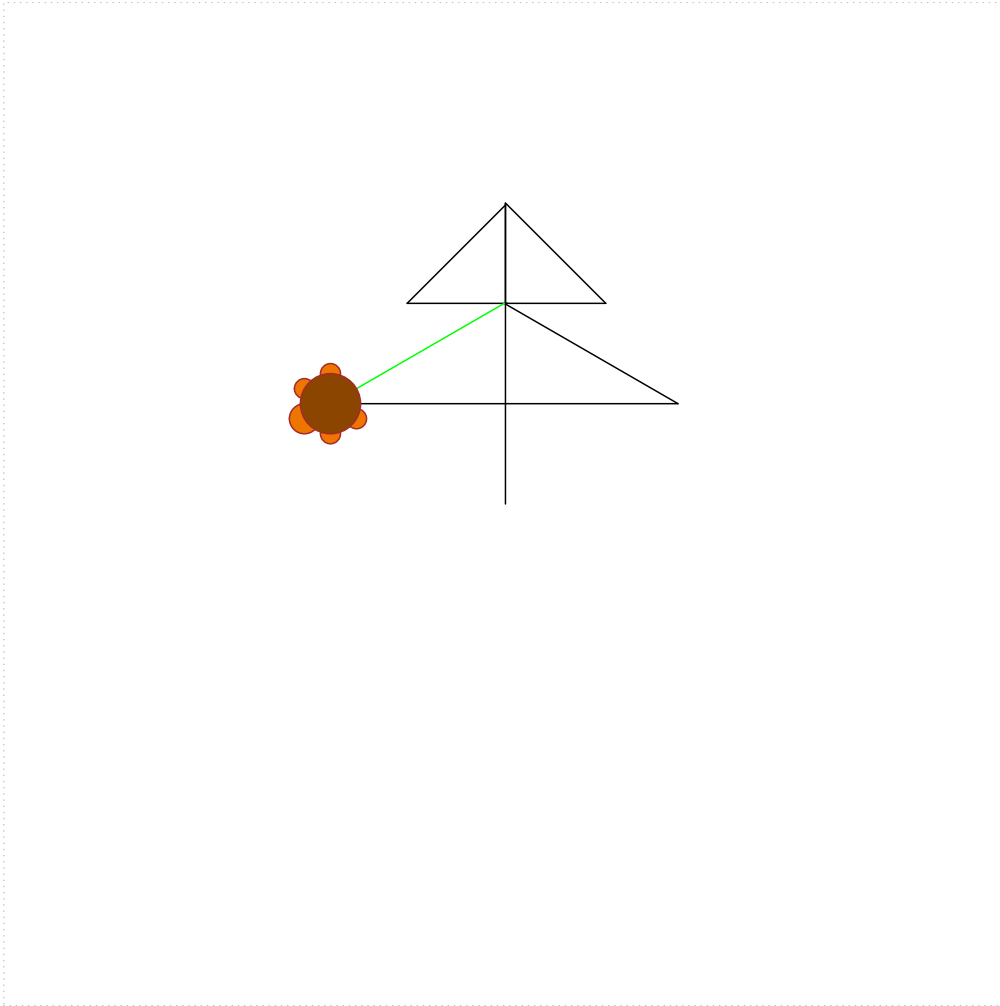
```
> turtle_right(90)
> turtle_up()
> turtle_forward(dist=10)
> turtle_right(angle=90)
> turtle_forward(dist=17)
> turtle_down()
> turtle_left(angle=180)
> turtle_forward(dist=34)
```

Similarly, you may show or hide the Turtle image, using the `turtle_show()` and `turtle_hide()` functions respectively. If you call a lot of functions it is strongly recommended to hide the Turtle first as it speeds up the process.



```
> turtle_hide()  
> turtle_left(angle=150)  
> turtle_forward(dist=20)  
> turtle_show()
```

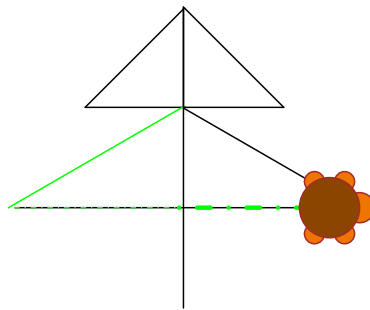
To change the nature of the Turtle's trace you can use the `turtle_col()`, `turtle_lty()` and `turtle_lwd()` functions. The first one, as you can easily guess, changes the color of the path the Turtle is making. For example, if you wish to change the trace into green try



```
> turtle_col(col="green")  
> turtle_left(angle=60)  
> turtle_forward(dist=20)
```

The full list of colors is available under the `colors()` function. Important! Remember that when passing an argument to this function you always have to use the quotation marks.

The `turtle_lty()` and `turtle_lwd()` functions change the type and the width of the line the Turtle is making. To change the type of the path as an argument pass a number from 0 to 6 – each means a different type of the line (0 =blank, 1 =solid (default), 2 = dashed, 3 = dotted, 4 = dotdash, 5 = longdash, 6 = twodash). To change the width of the line use the `turtle_lwd()` function. As an argument you pass a width you desire (but don't exaggerate!).



```
> turtle_left(angle=150)
> turtle_lty(lty=4)
> turtle_forward(dist=17)
> turtle_lwd(lwd=3)
> turtle_forward(dist=15)
```

If you got lost in the terrarium don't worry! There is a function `turtle_status()` which returns the parameters of your drawing. It tells you whether the Turtle and its path are visible, the width and height of the terrarium, where the Turtle is placed right now and at which angle.

```
> turtle_status()

$DisplayOptions
$DisplayOptions$col
[1] "green"

$DisplayOptions$lty
[1] 4

$DisplayOptions$lwd
[1] 3

$DisplayOptions$visible
[1] TRUE

$DisplayOptions$draw
[1] TRUE
```

```
$Terrarium
$Terrarium$width
[1] 100
```

```
$Terrarium$height
[1] 100
```

```
$TurtleStatus
$TurtleStatus$x
[1] 64.55999
```

```
$TurtleStatus$y
[1] 60
```

```
$TurtleStatus$angle
[1] -630
```

If you just want to know where the Turtle is or at which angle try `turtle_getpos()` and `turtle_getangle()` functions respectively.

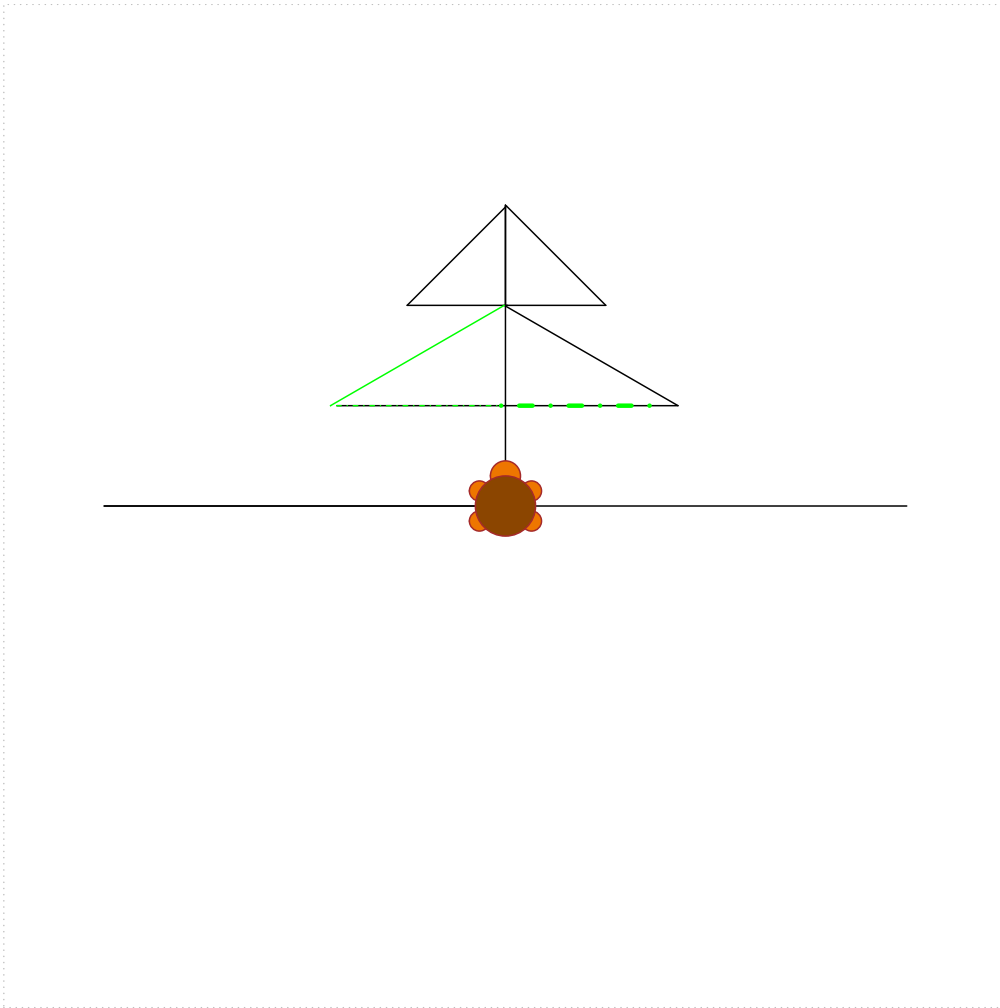
```
> turtle_getpos()
```

```
      x      y
64.55999 60.00000
```

```
> turtle_getangle()
```

```
angle
-630
```

If you wish to place the Turtle back at the starting position and delete all of the graphical parameters you set use the `turtle_reset()` function. The `turtle_goto()` function on the other hand, makes the Turtle go to the place you tell it to in its arguments passing the *x* and *y* coordinates. Mind that this function leaves the trace while using.



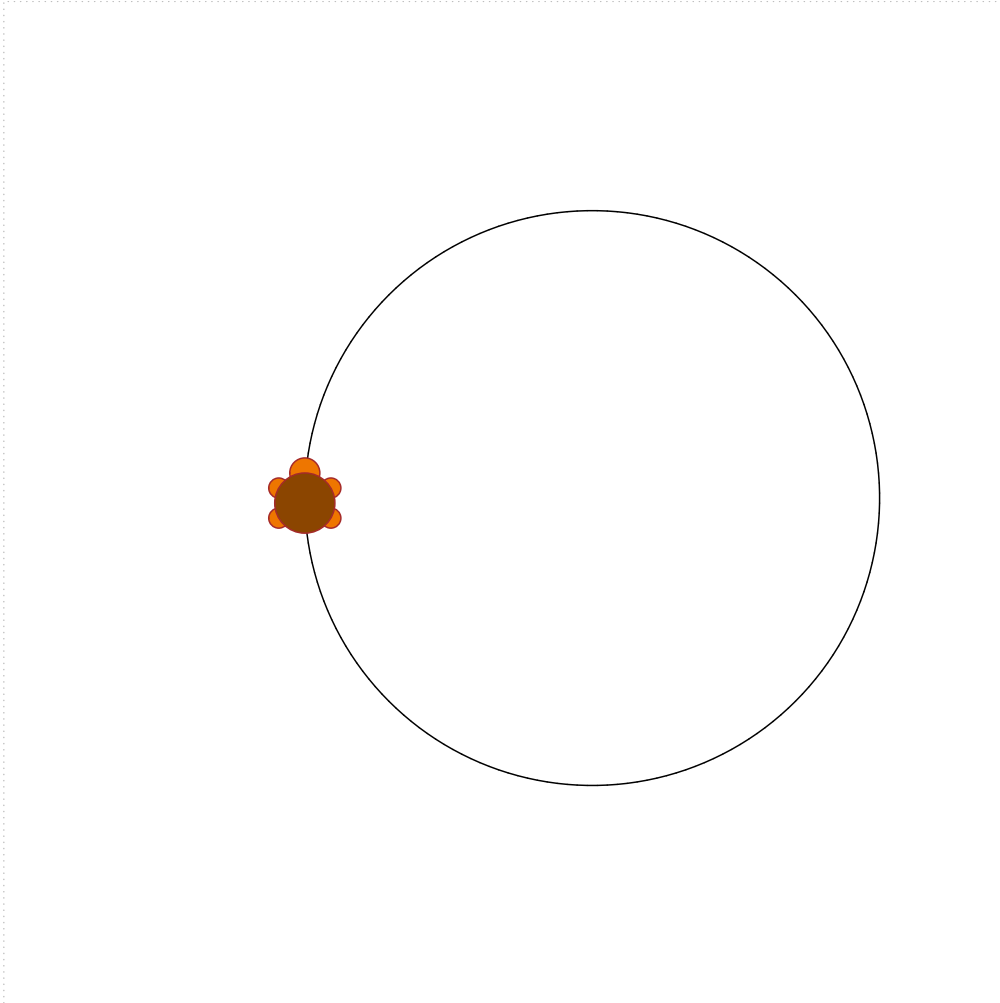
```
> turtle_reset()  
> turtle_goto(x=10,y=50)  
> turtle_goto(x=90,y=50)  
> turtle_reset()
```

3 Introduction to R

This section shows how to connect the functions listed above with the options that R gives us. For example, sometimes you would like to repeat some actions several times. If it is once or twice then it is easy to type it. But if you wish to use the same functions for, let's say, 360 times then it turns out to be quite difficult.

Do not worry though! R has the tools to deal with it. In the case you want to repeat actions several times we can use the so called loop. The syntax is following: `for(i in 1:100){}`. `i` is the counter that counts the turns of the loop. `1:100` defines when the counter should start counting and when it should stop. In this case the counter `i` starts in 1 and it increases by 1 each time the loop turns. The loop will stop turning when `i` reaches 100.

Between `{}` you write the functions you would like to repeat. For example:



```
> turtle_init()
> turtle_up()
> turtle_goto(x=30,y=50)
> turtle_down()
> turtle_hide()
> for(i in 1:180){
+   turtle_forward(dist=1)
+   turtle_right(angle=2)
+ }
> turtle_show()
```

Sometimes it takes time to make the loop, especially when it turns many times. As you can see by using the loop a circle was drawn.

Another way to use the functions listed above is to make a function, that uses them. For example, if you'd like to make a square you can write a function that will do that for you. The syntax is the following:

```
> turtle_square<-function(r=10){
```

```
+   for(i in 1:4){
+       turtle_forward(r)
+       turtle_right(90)
+   }
+ }
```

square is the name of the function. After the word function you list the parameters in brackets. If there is more than one parameter you list them separating by comma. You can make the parameters default as in the example: `r=10` but you can also leave it to the decision of the user. Then it would be `function(r)`. You have to compile the function and then you can use it as all the other functions:

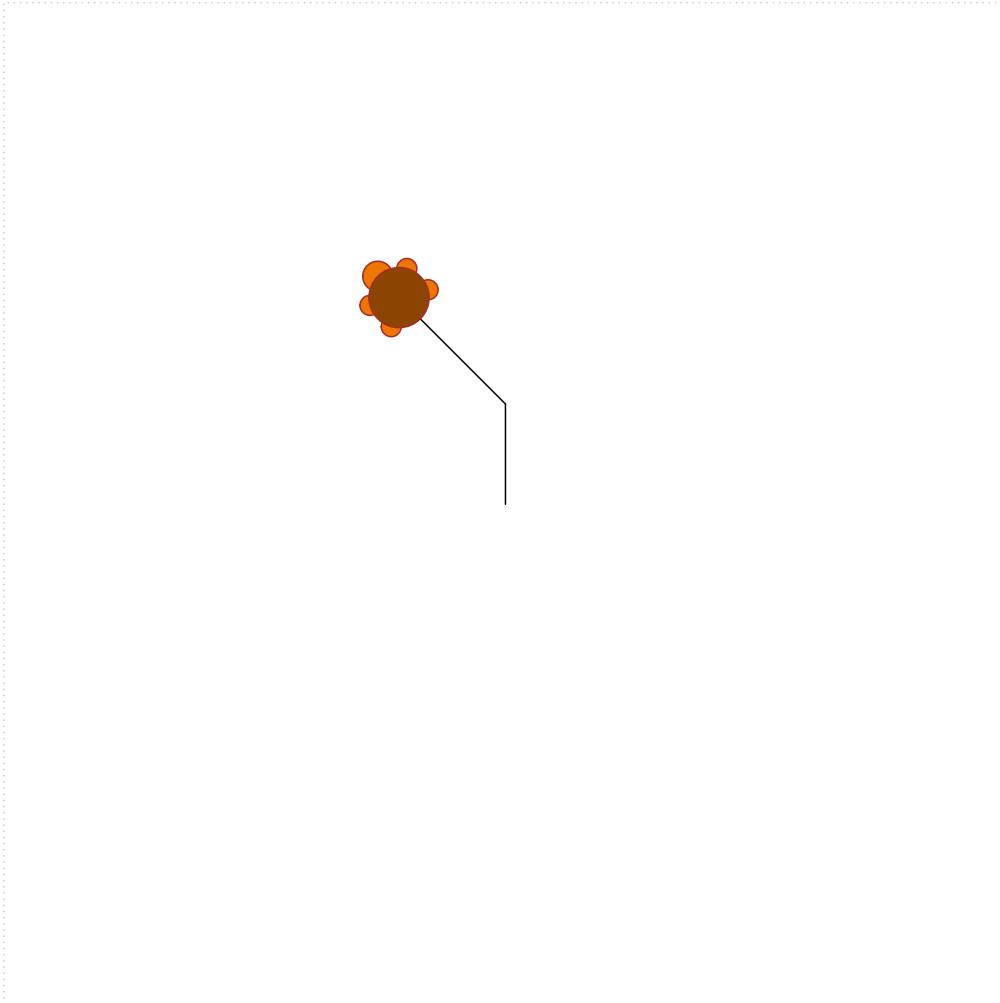


```
> turtle_init()
> turtle_square()
> turtle_left(angle=90)
> turtle_forward(dist=30)
> turtle_square(r=5)
```

4 Advanced Usage of the Package

Now you know the basics. There some more advanced methods of usage of the package. There is one more function (`turtle_do()`) in the package that is designed for more complicated sequence of commands. The usage is the following: in the argument you put `expr =` and between the curly brackets you give the sequence. Each command should be separated by a new line. Let's look at the example:

```
> turtle_init()
> turtle_do(expr = {
+   turtle_move(10)
+   turtle_turn(45)
+   turtle_move(15)
+ })
>
```



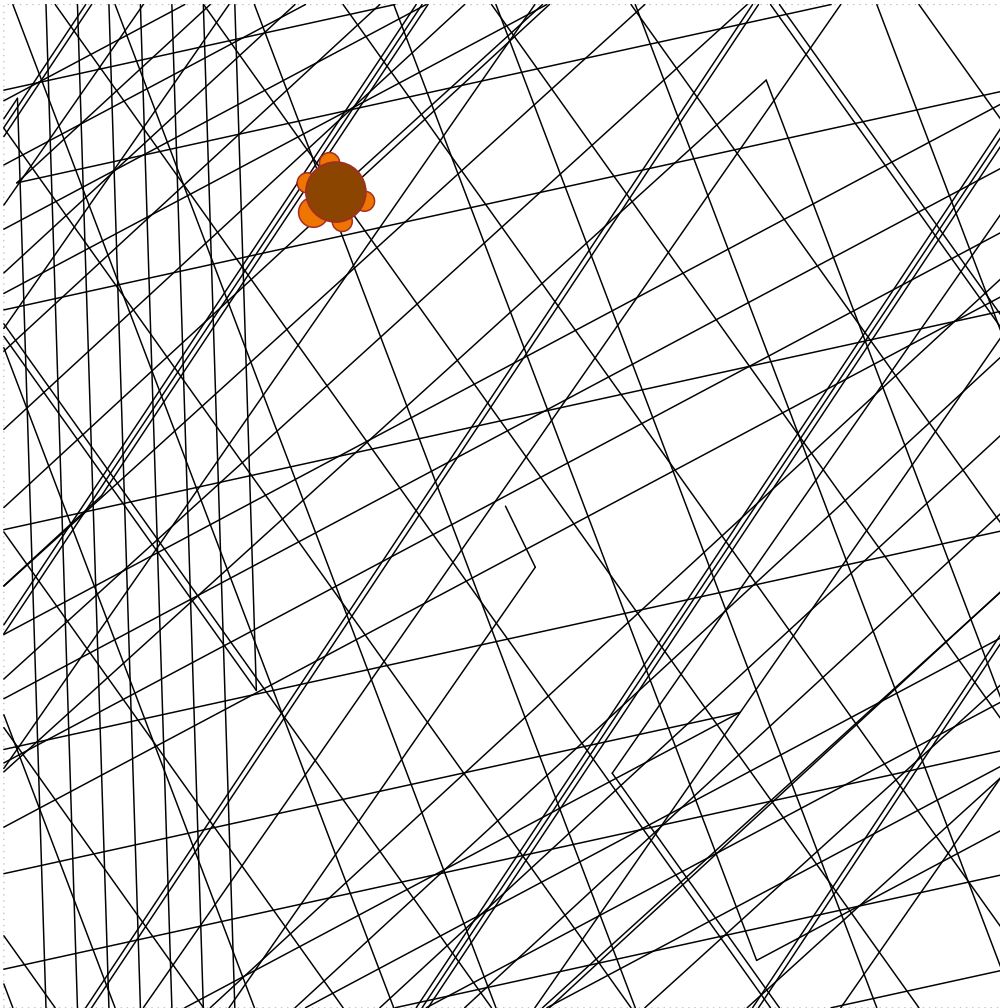
You may ask why bother using such a function if the result is the same as while using three separate commands (in this case this would be `turtle_move(10); turtle_turn(45); turtle_move(15)`). The thing is that this function hides the Turtle before performing the expression, thus the time that `expr` use is decreased.

4.1 Examples

At the end of this guide we would like to present some colorful and inspiring examples.

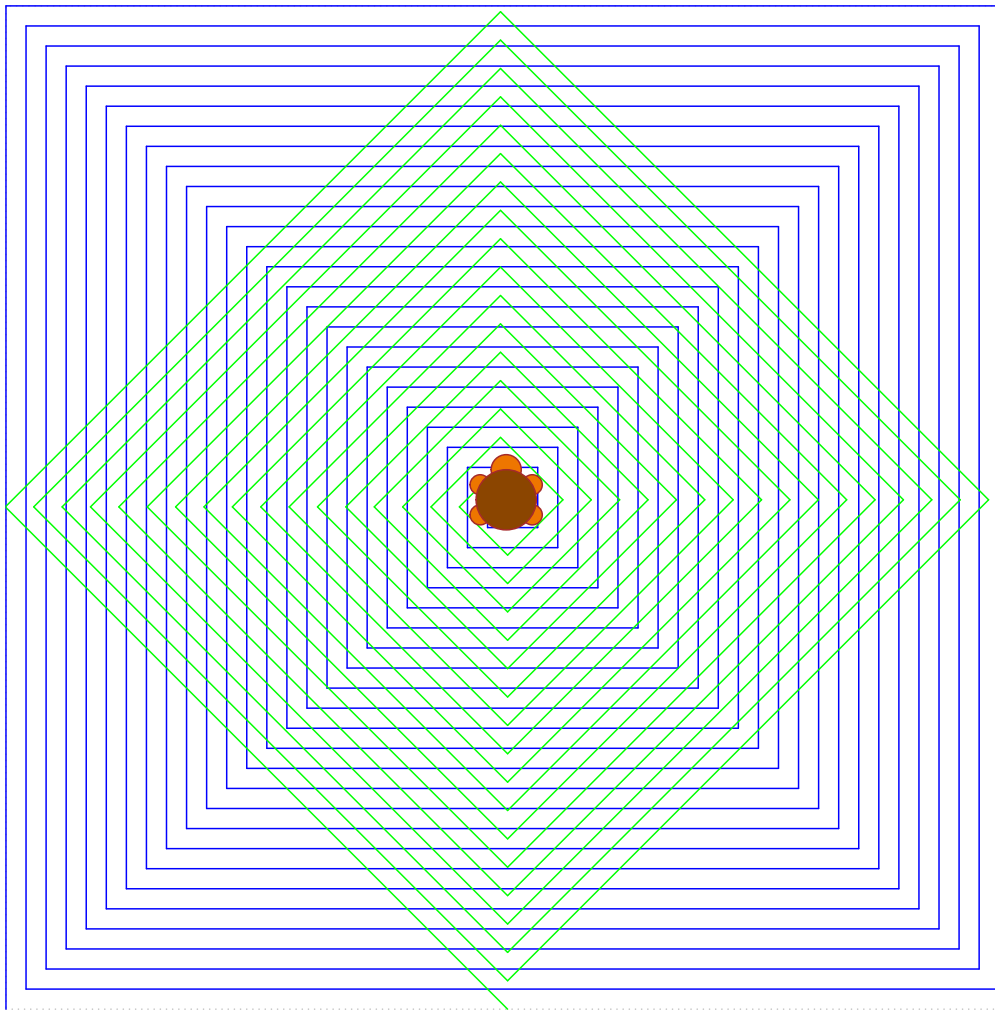
4.1.1 Random Lines

The first example is based on the random lines - every time the result is different.



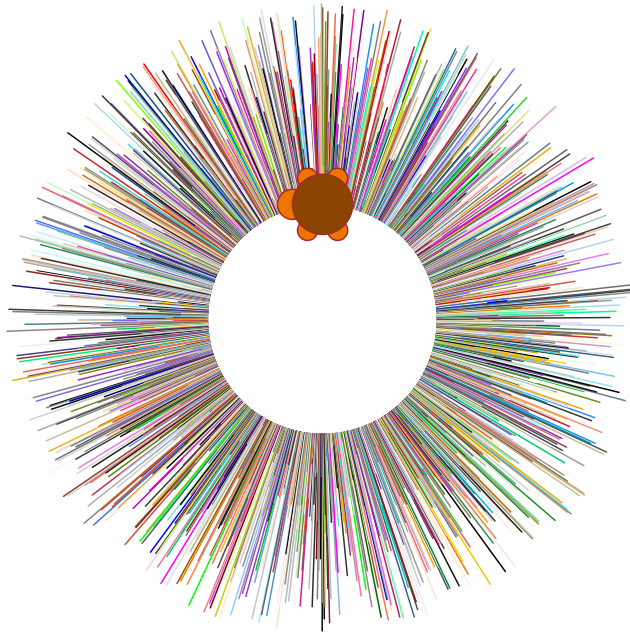
```
> turtle_init(100, 100, mode = "cycle")
> for(i in 1:10){
+   turtle_left(runif(1,0,360))
+   turtle_forward(runif(1, 0, 1000))
+ }
```

4.1.2 The Spiral



```
> drawSpiral <- function(lineLen){  
+   if (lineLen > 0){  
+     turtle_forward(lineLen)  
+     turtle_right(90)  
+     drawSpiral(lineLen-5)  
+   }  
+   invisible(NULL)  
+ }  
> turtle_init(500, 500, mode="clip")  
> turtle_setpos(x=0, y=0)  
> turtle_col("blue")  
> turtle_do(drawSpiral(500))  
> turtle_setpos(x=250, y=0)  
> turtle_left(45)  
> turtle_col("green")  
> turtle_do(drawSpiral(354))  
> turtle_setangle(0)
```

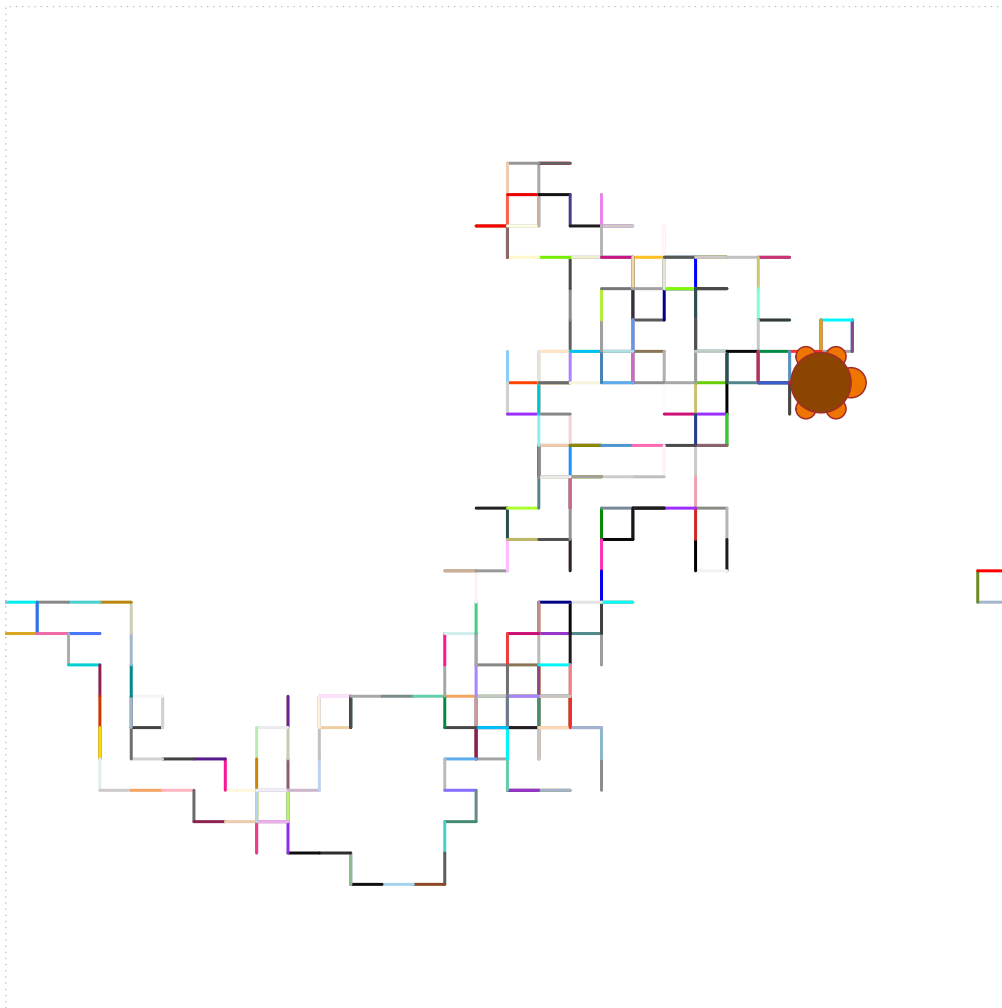

4.1.3 The Turtle Rainbow Star



```
> turtle_star <- function(intensity=1){  
+   y <- sample(1:657, 360*intensity, replace=TRUE)  
+   for (i in 1:(360*intensity)){  
+     turtle_right(90)  
+     turtle_col(colors()[y[i]])  
+     x <- sample(1:100,1)  
+     turtle_forward(x)  
+     turtle_up()  
+     turtle_backward(x)  
+     turtle_down()  
+     turtle_left(90)  
+     turtle_forward(1/intensity)  
+     turtle_left(1/intensity)  
+   }  
> turtle_init(500,500)  
> turtle_left(90)  
> turtle_hide()  
> turtle_star(7)  
> turtle_show()
```

4.1.4 The Turtle Brownian

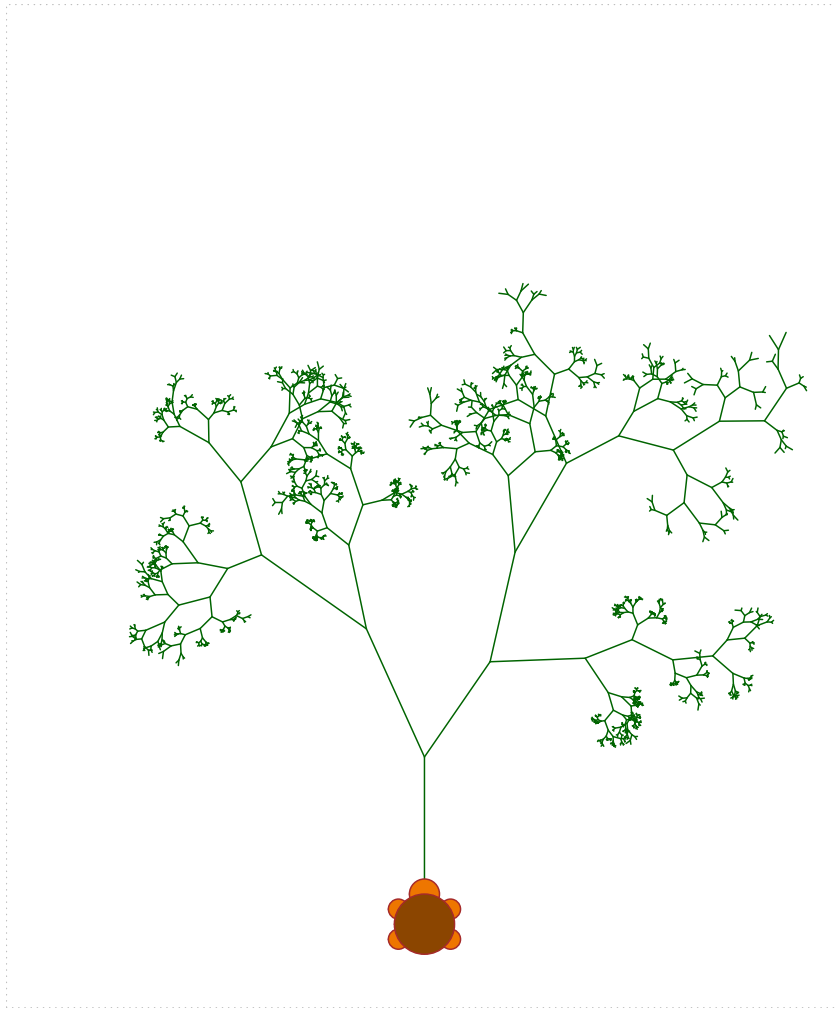
This example is inspired by Brownian motion.



```
> turtle_brownian <- function(steps=100, length=10){
+   turtle_lwd(2)
+   angles <- sample(c(90,270,180,0), steps,replace=TRUE)
+   coll <- sample(1:657, steps, replace=TRUE)
+   for (i in 1:steps){
+     turtle_left(angles[i])
+     turtle_col(colors()[coll[i]])
+     turtle_forward(length)
+   }
+ }
> turtle_init(800,800, mode="cycle")
> turtle_do(turtle_brownian(500, length=25))
```

4.1.5 The Fractal Tree

The next three examples are based on fractals - in the assumption they display self-similar pattern.

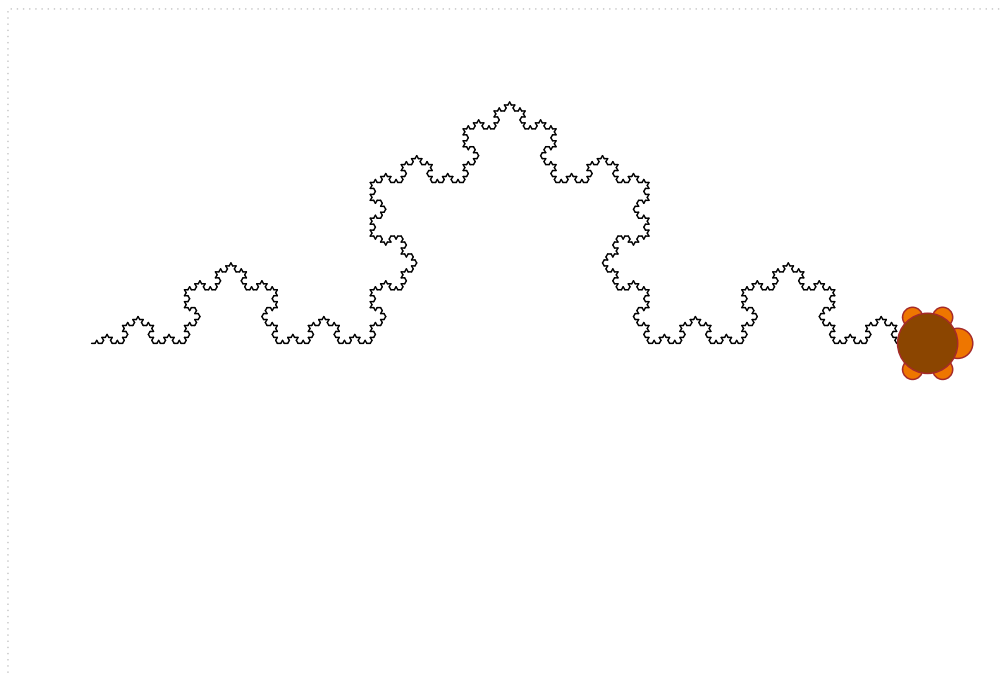


```

> fractal_tree <- function(s=100, n=2) {
+   if (n <= 1) {
+     turtle_forward(s)
+     turtle_up()
+     turtle_backward(s)
+     turtle_down()
+   }
+   else {
+     turtle_forward(s)
+     a1 <- runif(1, 10, 60)
+     turtle_left(a1)
+     fractal_tree(s*runif(1, 0.25, 1), n-1)
+     turtle_right(a1)
+     a2 <- runif(1, 10, 60)
+     turtle_right(a2)
+     fractal_tree(s*runif(1, 0.25, 1), n-1)
+     turtle_left(a2)
+     turtle_up()
+     turtle_backward(s)
+     turtle_down()
+   }
+ }
> set.seed(123)
> turtle_init(500, 600, clip)
> turtle_do({
+ turtle_up()
+ turtle_backward(250)
+ turtle_down()
+ turtle_col("darkgreen")
+ fractal_tree(100, 12)
+ })

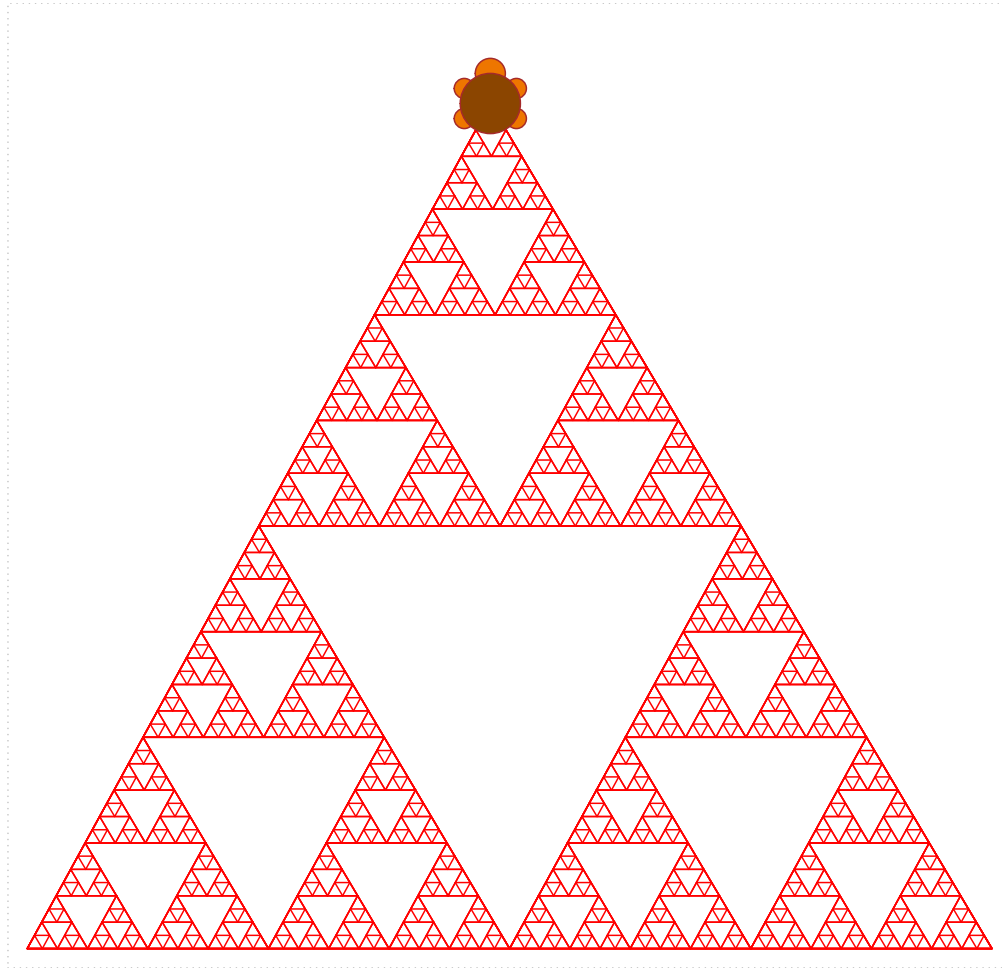
```

4.1.6 The Koch Snowflake



```
> turtle_init(600, 400, error)
> turtle_up()
> turtle_left(90)
> turtle_forward(250)
> turtle_right(180)
> turtle_down()
> koch <- function(s=50, n=6) {
+   if (n <= 1)
+     turtle_forward(s)
+   else {
+     koch(s/3, n-1)
+     turtle_left(60)
+     koch(s/3, n-1)
+     turtle_right(120)
+     koch(s/3, n-1)
+     turtle_left(60)
+     koch(s/3, n-1)
+   }
+ }
> turtle_hide()
> koch(500, 6)
> turtle_show()
```

4.1.7 The Sierpinski Triangle



```

> drawTriangle<- function(points){
+   turtle_setpos(points[1,1],points[1,2])
+   turtle_goto(points[2,1],points[2,2])
+   turtle_goto(points[3,1],points[3,2])
+   turtle_goto(points[1,1],points[1,2])
+ }
> getMid<- function(p1,p2) c((p1[1]+p2[1])/2, c(p1[2]+p2[2])/2)
> sierpinski <- function(points, degree){
+   drawTriangle(points)
+   if (degree > 0){
+     p1 <- matrix(c(points[1,], getMid(points[1,], points[2,]),
+                     getMid(points[1,], points[3,])), nrow=3, byrow=TRUE)
+
+     sierpinski(p1, degree-1)
+     p2 <- matrix(c(points[2,], getMid(points[1,], points[2,]),
+                     getMid(points[2,], points[3,])), nrow=3, byrow=TRUE)
+
+     sierpinski(p2, degree-1)
+     p3 <- matrix(c(points[3,], getMid(points[3,], points[2,]),
+                     getMid(points[1,], points[3,])), nrow=3, byrow=TRUE)
+     sierpinski(p3, degree-1)
+   }
+   invisible(NULL)
+ }
> turtle_init(520, 500, "clip")
> p <- matrix(c(10, 10, 510, 10, 250, 448), nrow=3, byrow=TRUE)
> turtle_col("red")
> turtle_do(sierpinski(p, 6))
> turtle_setpos(250, 448)

```