A Guide to the TurtleGraphics Package for R

A. Cena, M. Gągolewski, 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 . . . 4
2.3 Advanced Usage of the Package . . . 9

3 Introduction to R **11**
3.1 The for loop. . . . 11
3.2 If . . . 11
3.3 Functions . . . 13
3.4 Recursion . . . 14

4 Examples **15**
4.0.1 Random Lines . . . 15
4.0.2 The Spiral . . . 16
4.0.3 The Turtle Rainbow Star . . . 17
4.0.4 The Turtle Brownian . . . 18
4.0.5 The Fractal Tree . . . 19
4.0.6 The Koch Snowflake . . . 21
4.0.7 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 etiher 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 choosen 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

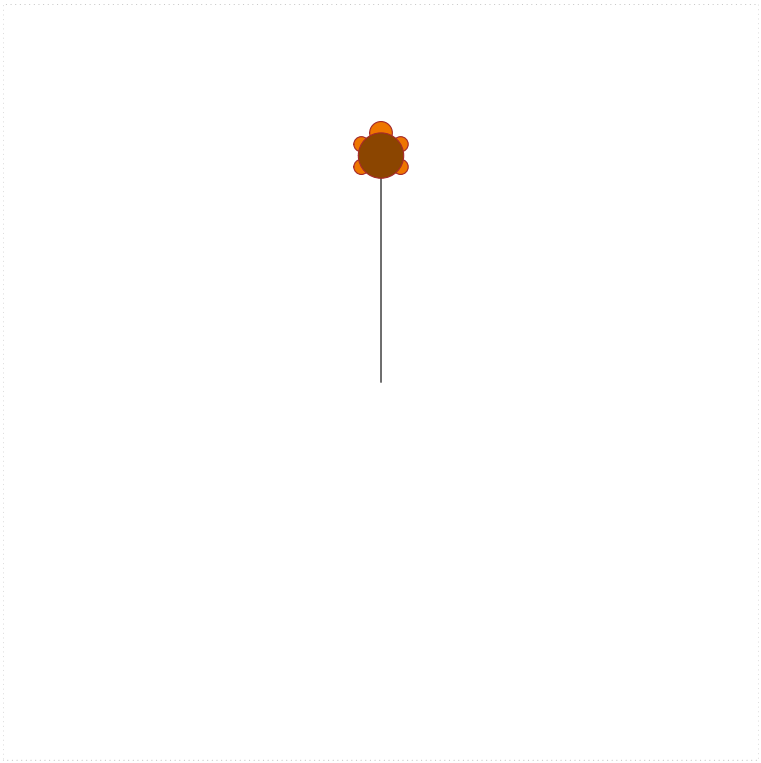
turtle_init. To start using the program call the turtle_init() function. It creates a plot region (sometimes called "Terrarium") and places the Turtle in the middle pointing north.

```
> turtle_init()
```

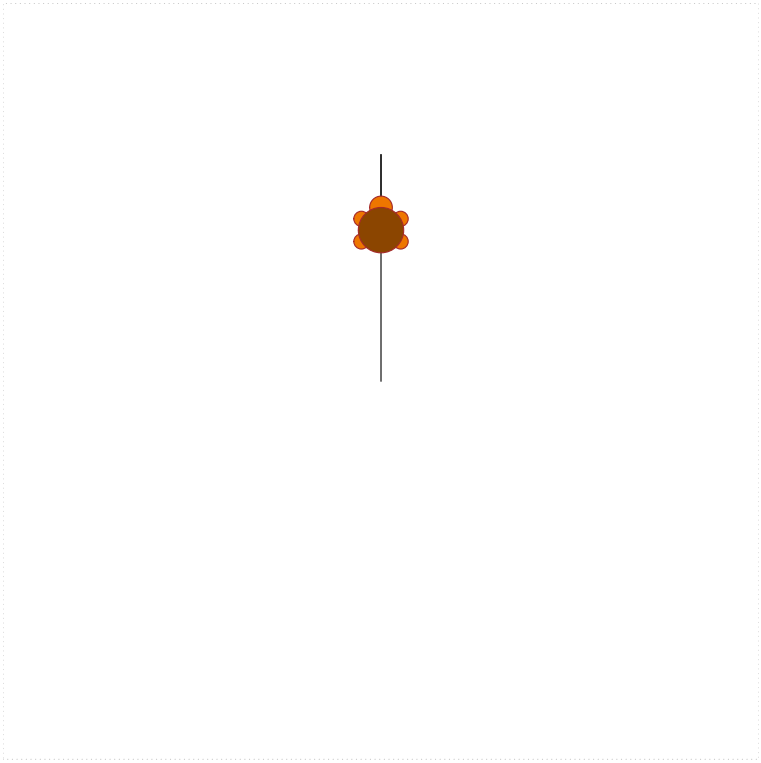
By default its size is 100 by 100 units. You can easily change it by passing as the arguments width and height (e.g. turtle_init(width=200, height=200)). There is one more argument that you can pass to this function. It defines the mode of the Terrarium and what happens if the Turtle moves outside the 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.

turtle_forward and turtle_backward. 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.

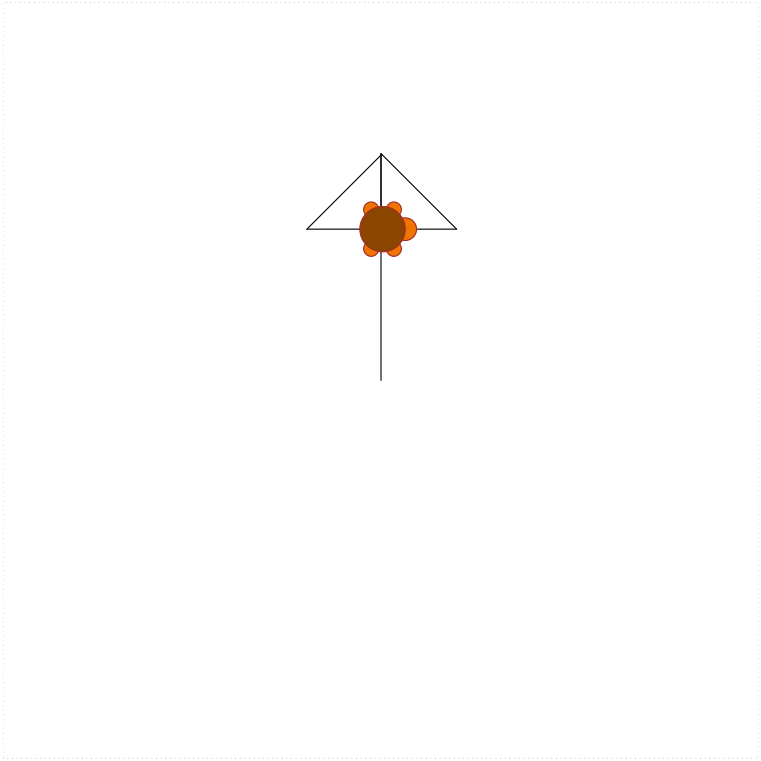


```
> turtle_init()
> turtle_forward(dist=30)
```



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

turtle_right and turtle_left. 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.

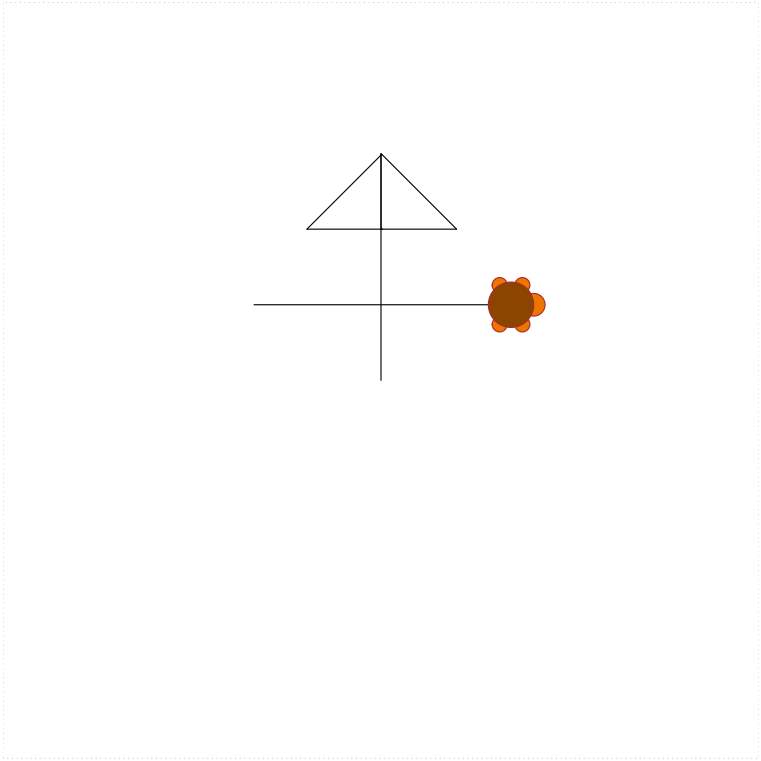


```
> turtle_right(angle=90)
> turtle_forward(dist=10)
> turtle_left(angle=135)
> turtle_forward(dist=14)
> turtle_left(angle=90)
> turtle_forward(dist=14)
> turtle_left(angle=135)
> turtle_forward(dist=10)
```

2.2.2 Additional Options

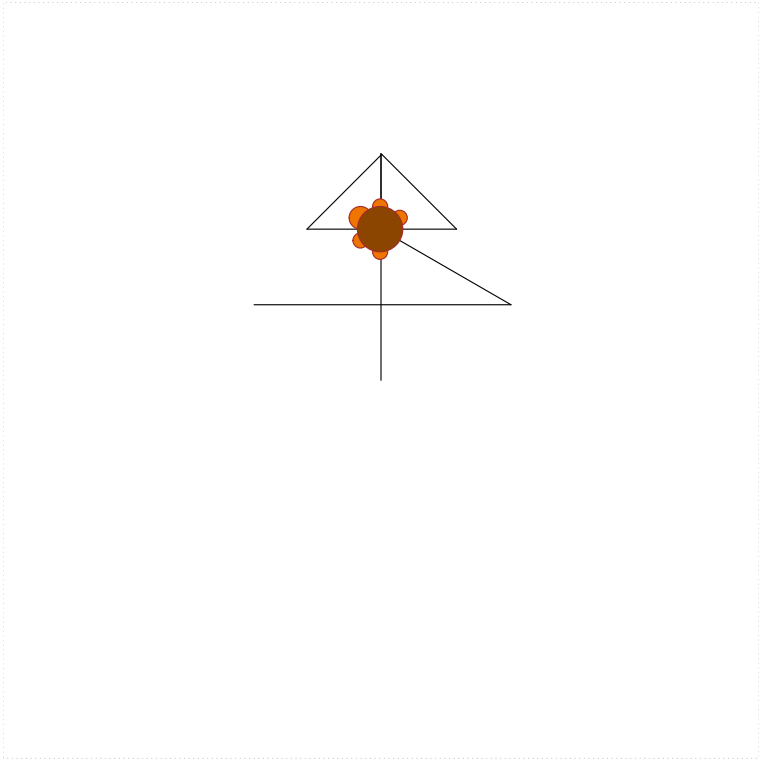
Now you know the basics and it is mostly it. Not very complicated, is it? Don't worry though, there are some additional parameters that you can play with.

turtle_up and turtle_down. 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.



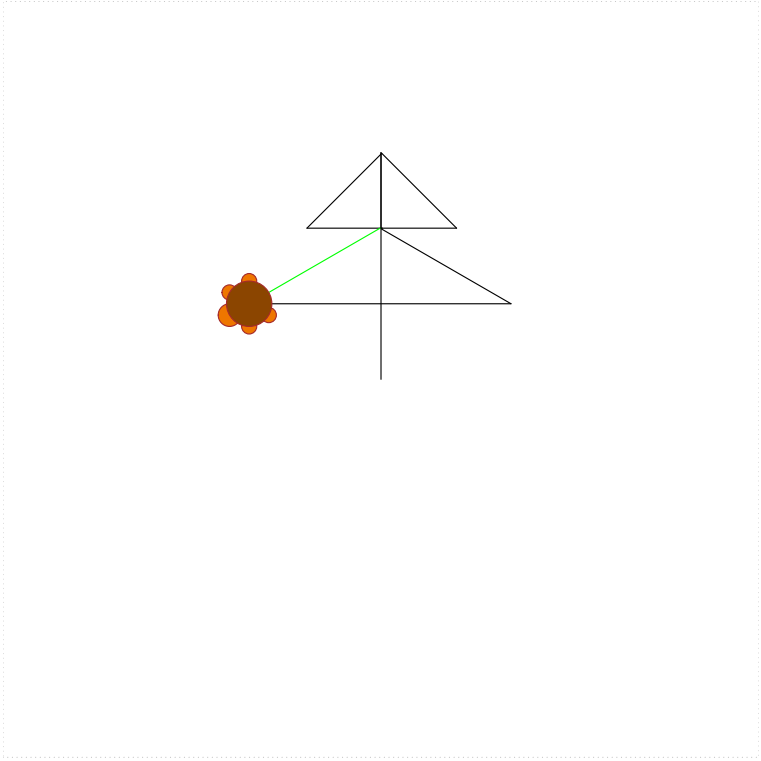
```
> turtle_right(90)
> turtle_up()
> turtle_forward(dist=10)
> turtle_right(angle=90)
> turtle_forward(dist=17)
> turtle_down()
> turtle_left(angle=180)
> turtle_forward(dist=34)
```

turtle_hide and turtle_show. 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()
```

turtle_col, turtle_lty and turtle_lwd. 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

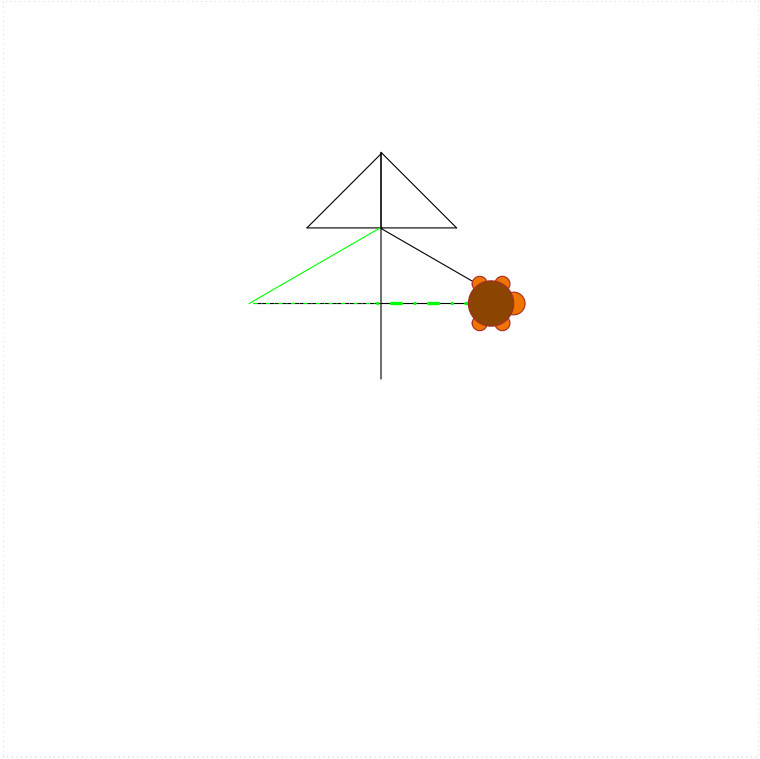


```
> turtle_col(col="green")
> turtle_left(angle=60)
```

```
> 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 Turle 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)
```

turtle_status, turtle_getpos and turtle_getangle. 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 Turle 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.

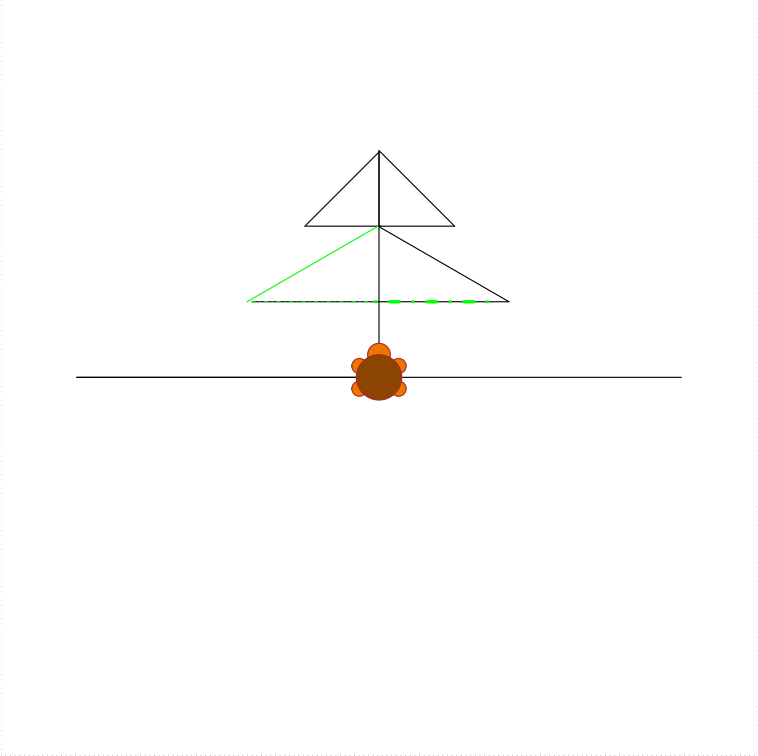
```
> turtle_getpos()

       x        y
64.55999 60.00000

> turtle_getangle()

angle
 -630
```

turtle_reset and turtle_goto. 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.

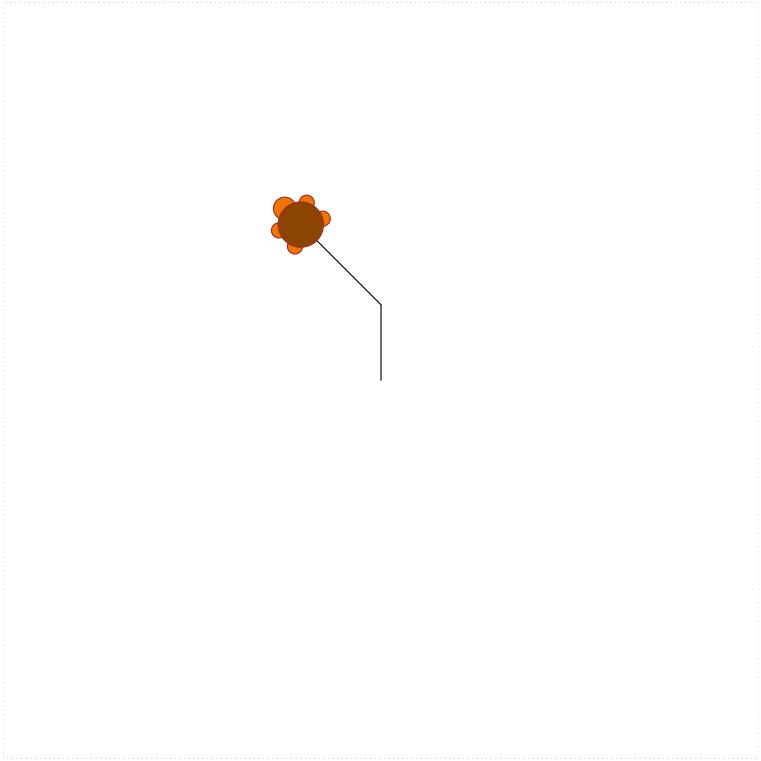


```
> turtle_reset()
> turtle_goto(x=10,y=50)
> turtle_goto(x=90,y=50)
> turtle_reset()
```

2.3 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 separted 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.

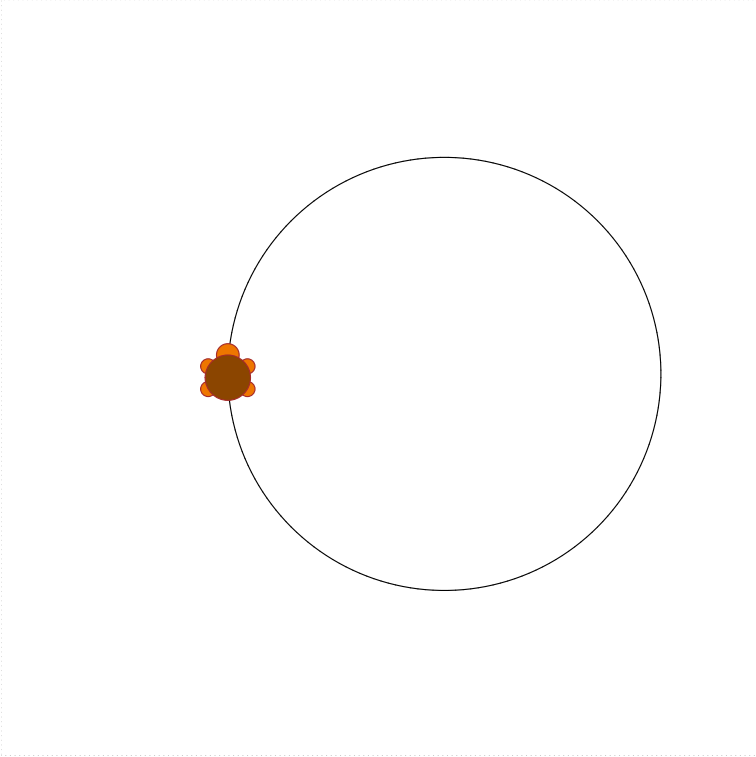
3 Introduction to R

3.1 The for loop.

This section shows how to connect the functions listed above with the options that R provides us with. 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.

Fortunately, R has the tools to deal with it. In case you want to repeat actions several times we can use the so called loop. The syntax is the 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 the curly rackets ({}) you write the functions you would like to repeat, each separated by a new line. For example:

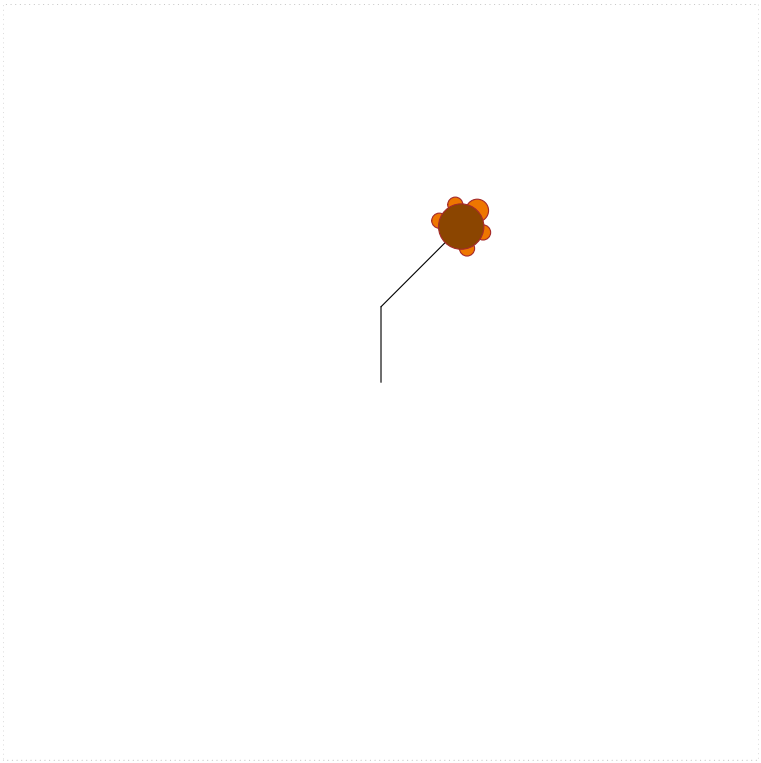


```
> turtle_init()
> turtle_up()
> turtle_goto(x=30,y=50)
> turtle_down()
> turtle_hide()
> for(i in 1:180){
+    turtle_forward(dist=1)
+    turtle_right(angle=2)
+ }
> turtle_show()
```

Sometimes it takes time to make the loop, especially when it turns many times. That is why we strongly recommend to hide the Turtle beforehand. As you can see thanks to usage of the loop a circle was drawn.

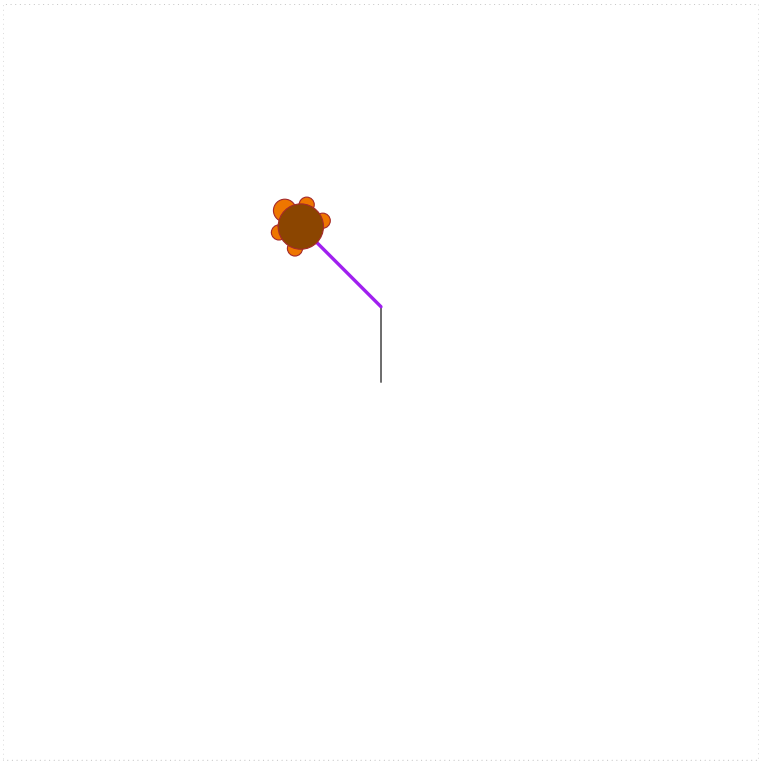
3.2 If

There are some cases when you would like to call a function when some condition is fulfilled. The if expression enables you to do it. The syntax is the following: if(condition) {}. The condition is always logical. When the condition is fulfilled the sequance of actions you put between the curly bracket is called. Let's see an example.



```
> turtle_init()
> turtle_forward(dist=10)
> x <- 1
> if(x>0.5){
+    turtle_right(angle=45)
+    turtle_col(col="red")
+ }
> turtle_forward(dist=15)
```

As you can see the condition is fulfilled so the Turtle turns right. What if you would like to perform some actions when the condition is fulfilled and some other actions in case it is not? There is an answer to it too. After the closing curly bracket you write else {} and again between the brackets there are the functions you would like to call.



```
> turtle_init()
> turtle_forward(dist=10)
> x<-runif(1) #this function returns a random value between 0 and 1, see ?runif
> if(x>0.5){
+    turtle_right(angle=45)
+    turtle_col(col="red")
+ } else {
+    turtle_left(angle=45)
+    turtle_lwd(lwd=3)
+    turtle_col(col="purple")
+ }
> turtle_forward(dist=15)
```

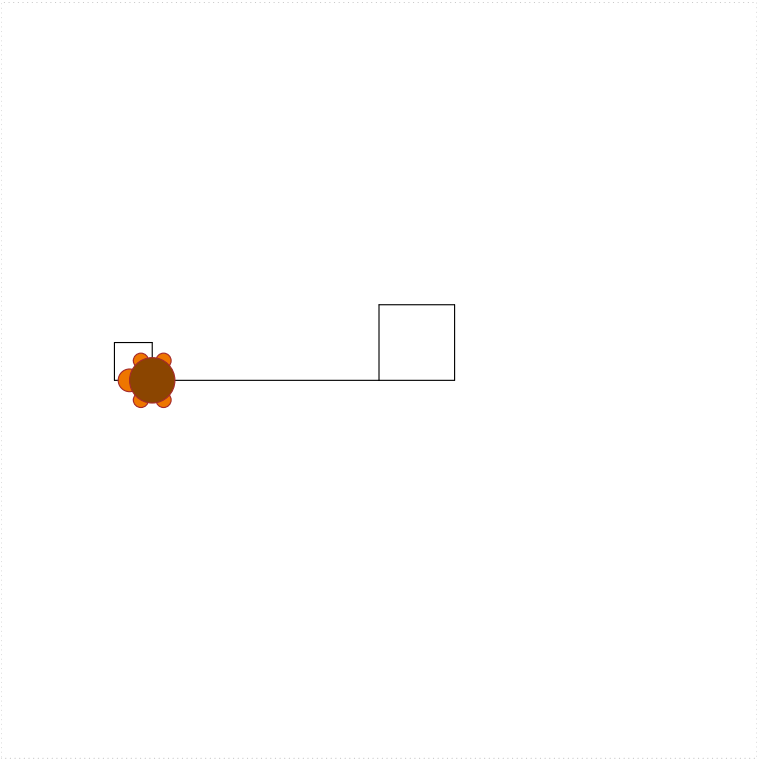
Now each time you call the above synax the result can be different.

3.3 Functions

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 sqaure you can write a functions 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)
+    }
+ }
```

turtle_square is the name of the function. After the word function you list the parameters in brackets. If there are 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)
```

3.4 Recursion

The last thing you should know while using this package is recursion. It is a process of repeating actions in the self-similar pattern. You have a base case and all the other cases are reduced to the base case that is defined by some rules (they are called recursive step).

Let's consider a simple example. Your base step is to walk 1 meter (so you know only how to walk 1 meter). The recursive step would be walking half way than you have to. So if your task is to walk, let's say, 16 meters than by recursion you have to walk 8 meters (but you don't do that, you don't know how). Then by recursion your task is to walk 4 meters. And again by recursion you have to walk 2 meters. And now by recursion you have to walk 1 meter. But you know how to do it! Great! Now you walk 1 meter. But if you walked 1 meter you can do another one and now you know how to walk 2 meters. If you know that, you can walk 2 meters twice and so you can walk 4 meters. And again you know how to do 4 meters, so you can walk it again and now you can move by 8 meters. And now you can do that one more time so you walk the entire 16 meters. Congratulations you have accomplished the task!

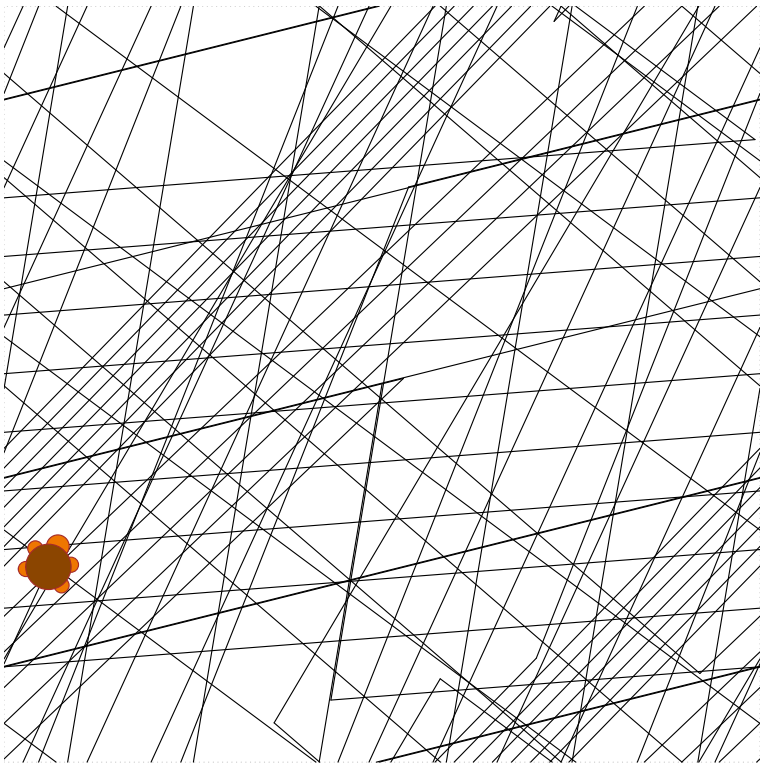
The other example of recursion is a fractal. Usually, a fractal is an image which at every scale shows the same pattern. In the example section (4) you have 3 characteristic examples of the fractals – the fractal tree, the Koch snowflake and the Sierpinski triangle.

4 Examples

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

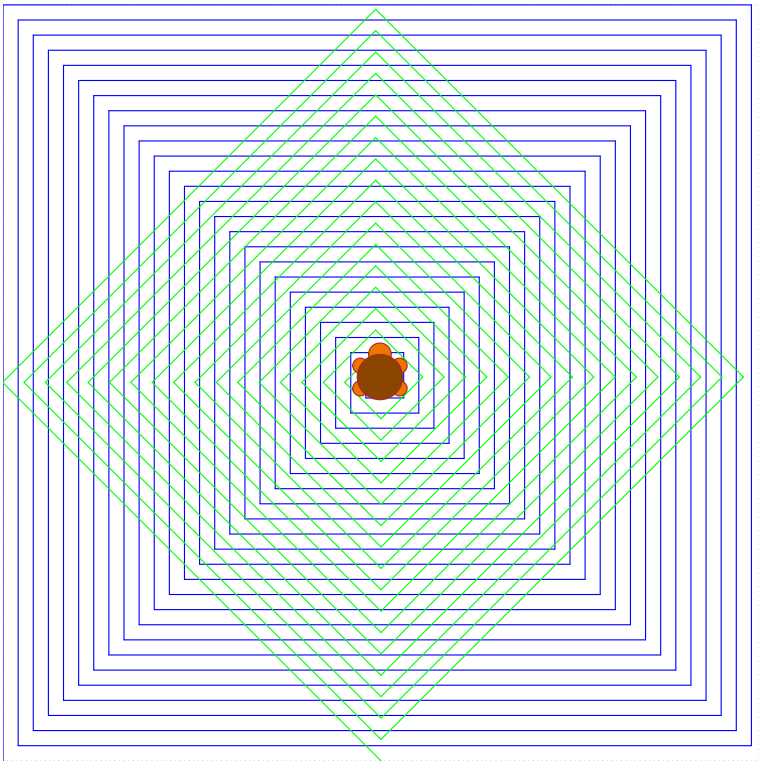
4.0.1 Random Lines

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



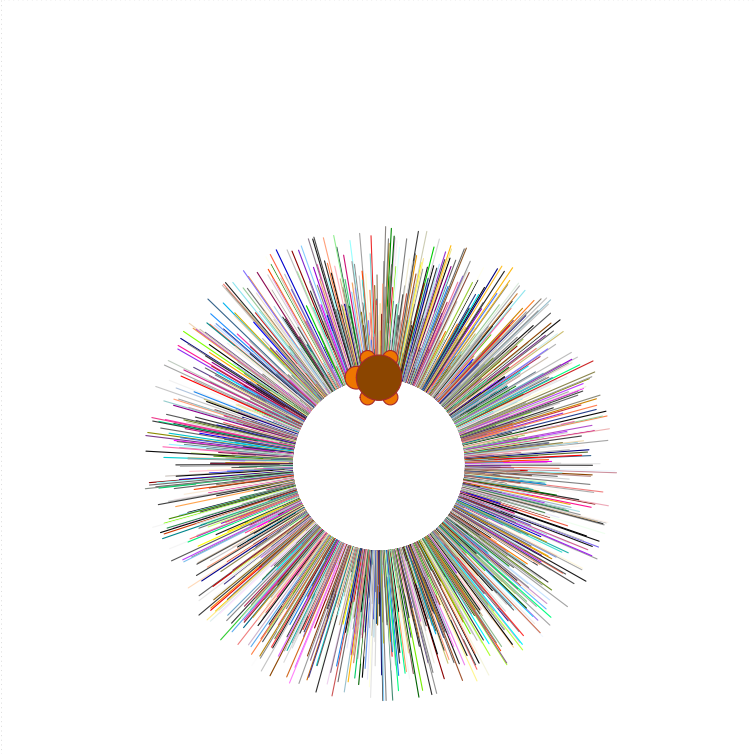
```
> turtle_init(100, 100, mode = "cycle")
> for(i in 1:10){
+   turtle_left(runif(1,0,360))
+   turtle_forward(runif(1, 0, 1000))
+ }
```

4.0.2 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)
```

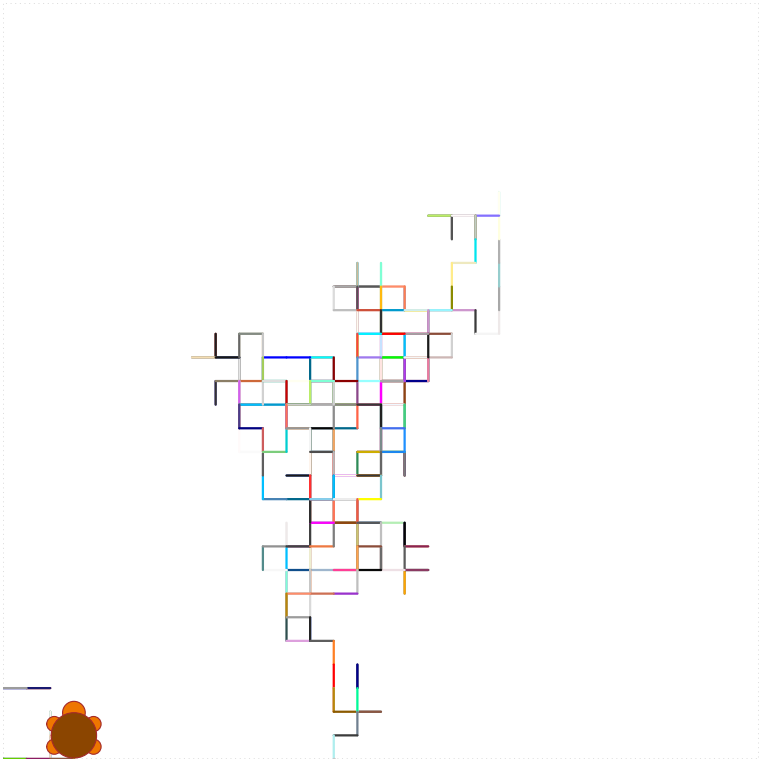
4.0.3 The Turtle Rainbow Star



```
> 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()
```

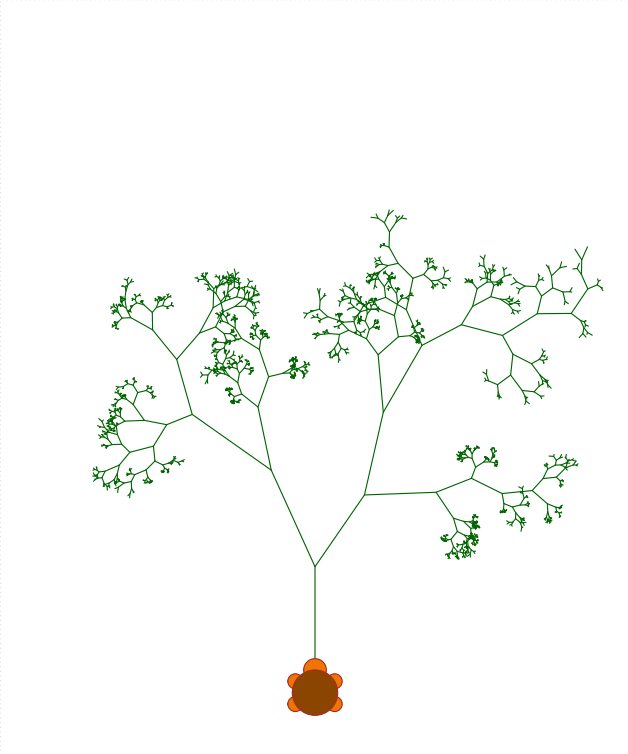
4.0.4 The Turtle Brownian

This example is inspired by Brownian motion.



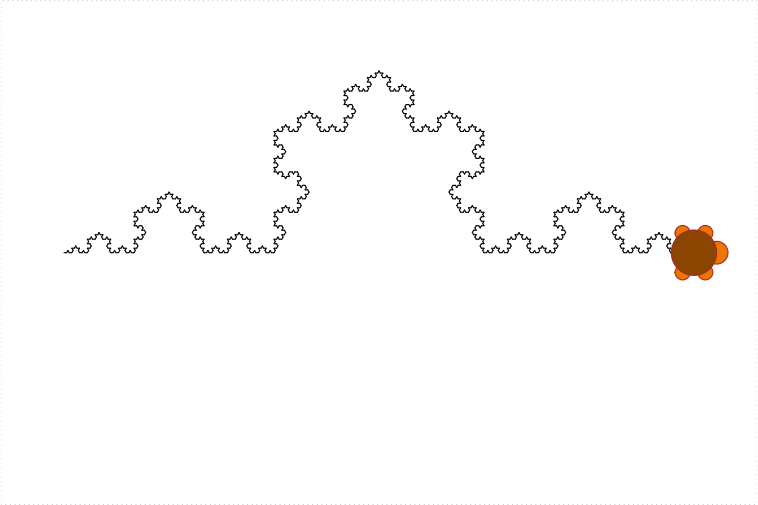
```
> turtle_brownian <- function(steps=100, length=10){
+    turtle_lwd(2)
+    angles <- sample(c(90,270,180,0), steps,replace=TRUE)
+    coll <- sample(1:657, steps, replace=TRUE)
+    for (i in 1:steps){
+       turtle_left(angles[i])
+       turtle_col(colors()[coll[i]])
+       turtle_forward(length)
+    }
+ }
> turtle_init(800,800, mode="clip")
> turtle_do(turtle_brownian(1000, length=25))
```

4.0.5 The Fractal Tree



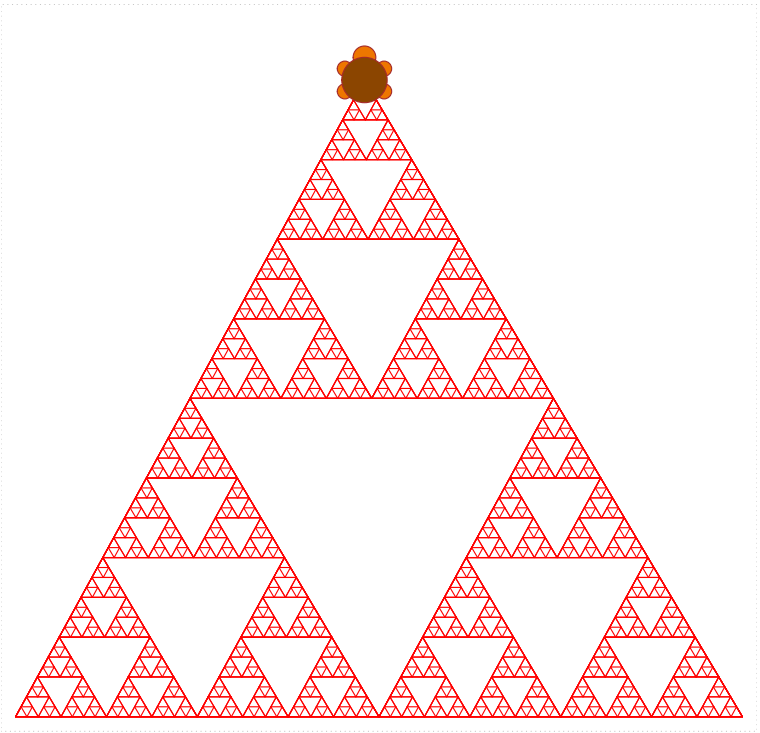
```
> fractal_tree <- function(s=100, n=2) {
+    if (n <= 1) {
+       turtle_forward(s)
+       turtle_up()
+       turtle_backward(s)
+       turtle_down()
+    }
+    else {
+       turtle_forward(s)
+       a1 <- runif(1, 10, 60)
+       turtle_left(a1)
+       fractal_tree(s*runif(1, 0.25, 1), 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)
+ })
```

4.0.6 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()
```

4.0.7 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)
```