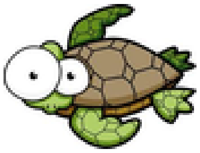
A Guide to the TurtleGraphics Package for R

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



Contents

1 The TurtleGraphics Package Introduction

The TurtleGraphics package offers R users the so-called "turtle graphics" facilities known from the Logo programming language. The key idea behind the package is to encourage children to learn programming and demonstrate that working with computers can be fun and creative.

The TurtleGraphics package allows to create either simple or more sophisticated graphics on the basis of lines. The Turtle, described by its location and orientation, moves with commands that are relative to its 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 by a given distance and to turn the Turtle by a chosen angle. The graphical options 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 TurtleGraphics package use the following calls.

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

Then you 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, call the turtle_init() function. It creates a plot region and places the Turtle in the Terrarium's center, facing north.

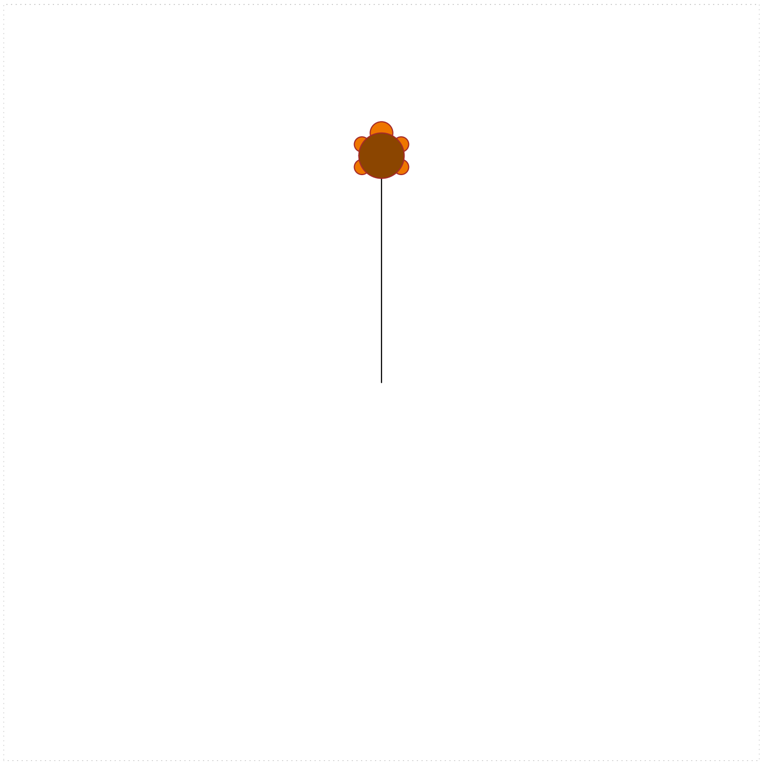
```
> turtle_init()
```

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)).

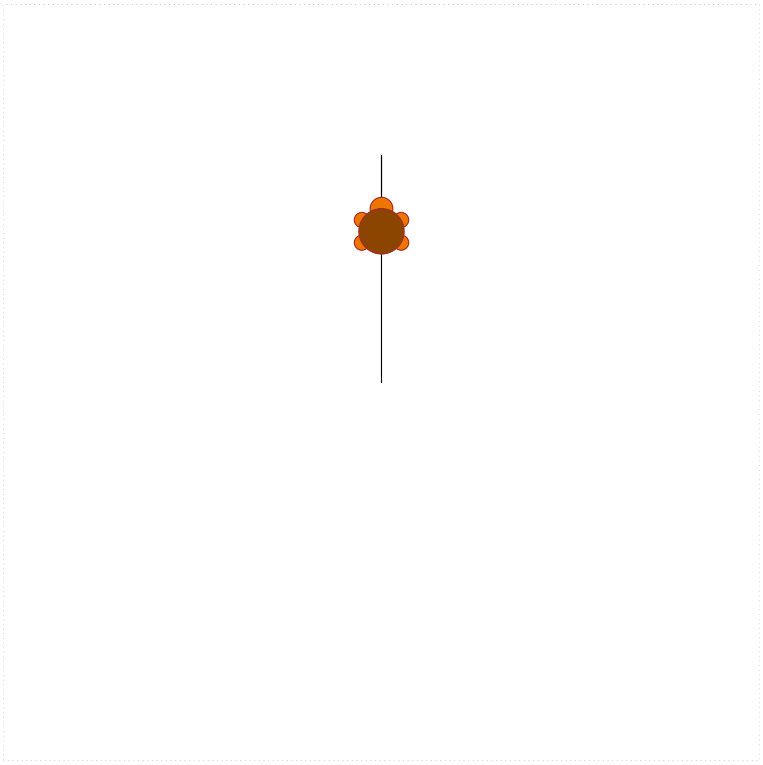
To define what happens if the Turtle moves outside the plot region, you can set the mode option. The default value, "clip", means that the Turtle can freely go outside the board (but it will not be seen). The "error" option does not let the Turtle out of the Terrarium – if the Turtle tries to escape, an error is thrown. The third value, "cycle", makes the Turtle come out on the other side of the board in case of crossing its border.

turtle_forward and turtle_backward. There are two main groups 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 by 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.

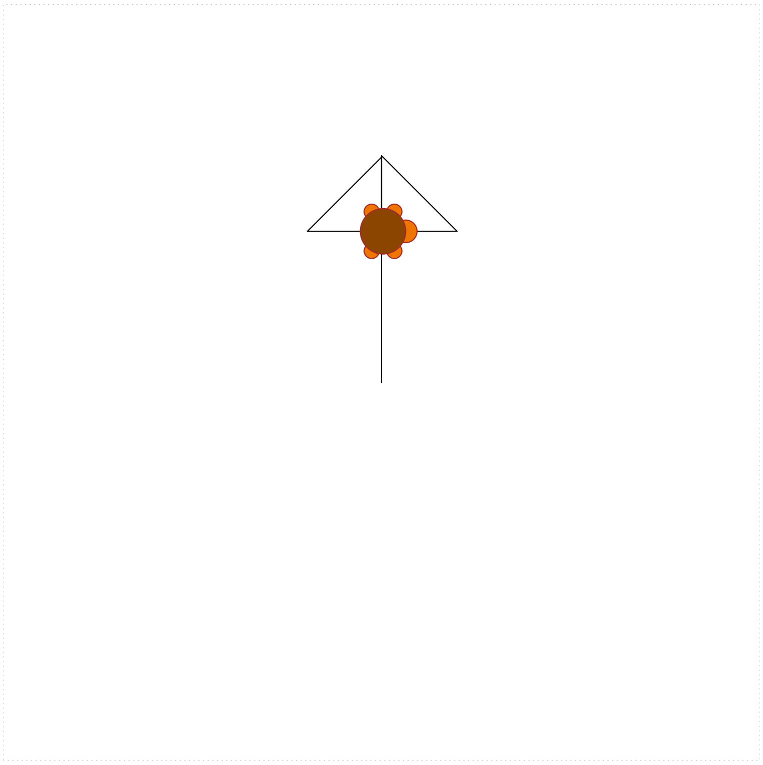


```
> turtle_init()
> turtle_forward(dist=30)
```



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

turtle_right and turtle_left. The other group of functions deal with the Turtle's rotation. turtle_left and the turtle_right change the Turtle's direction by a given angle.

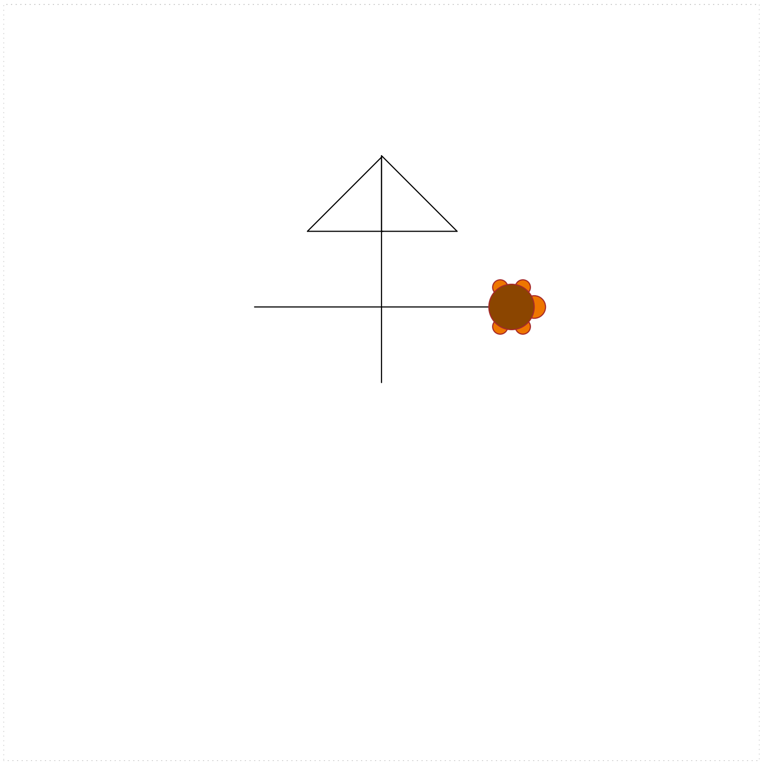


```
> turtle_right(angle=90)
> turtle_forward(dist=10)
> turtle_left(angle=135)
> turtle_forward(dist=14)
> turtle_left(angle=90)
> turtle_forward(dist=14)
> turtle_left(angle=135)
> turtle_forward(dist=10)
```

2.2.2 Additional Options

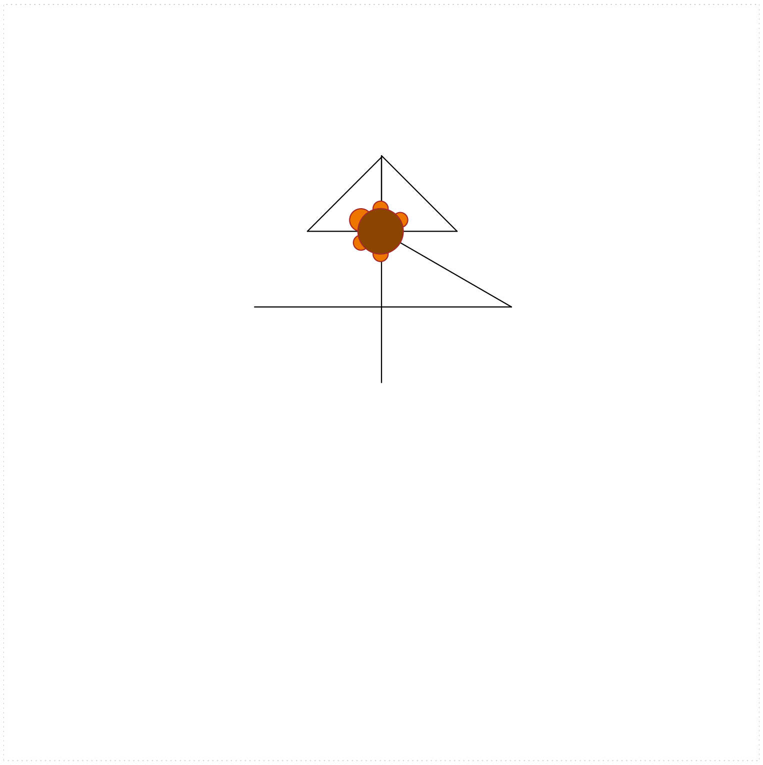
Let's discuss some additional features that you can play with.

turtle_up and turtle_down. To disable the path from being drawn you can 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.



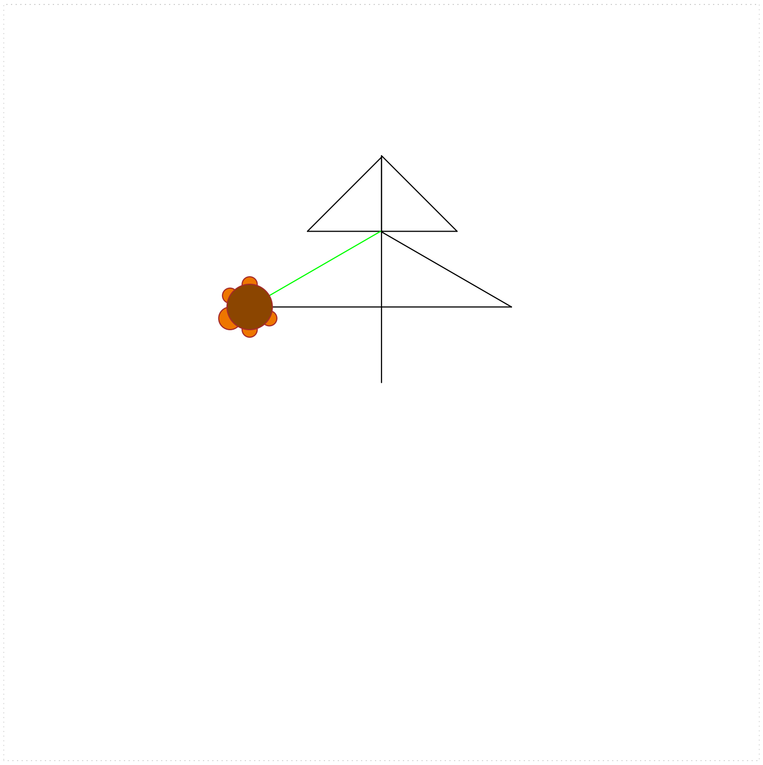
```
> turtle_right(90)
> turtle_up()
> turtle_forward(dist=10)
> turtle_right(angle=90)
> turtle_forward(dist=17)
> turtle_down()
> turtle_left(angle=180)
> turtle_forward(dist=34)
```

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 are calling a lot of functions it is strongly recommended to hide the Turtle first as it speeds up the drawing process, see also turtle_do().



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

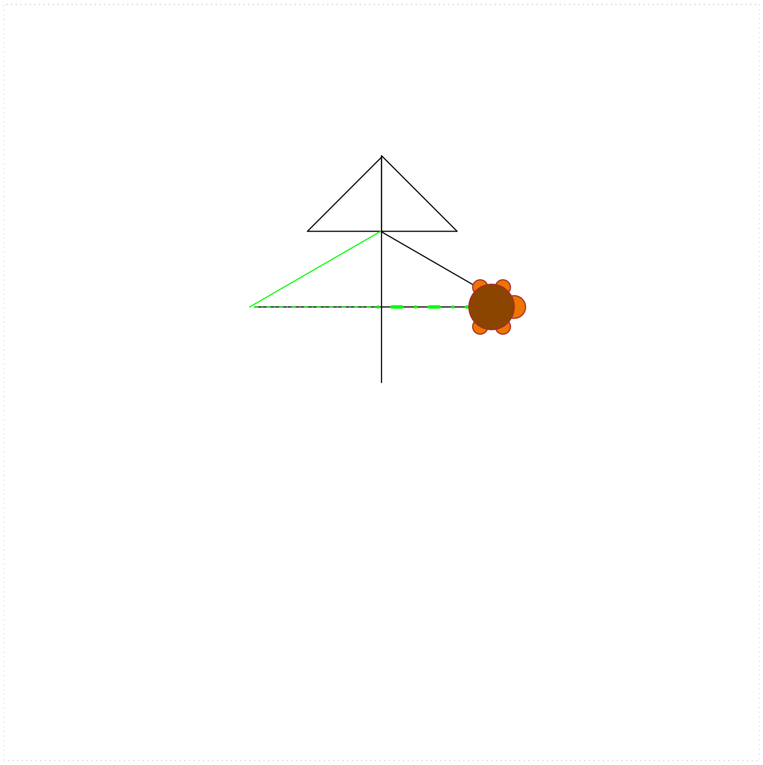
turtle_col, turtle_lty and turtle_lwd. To change the style of the Turtle's trace you can use the turtle_col(), turtle_lty(), and turtle_lwd() functions. The first one, as you can easily guess, changes the color of the path. For example, if you wish to change the trace into green, try:



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

The full list of colors is available under the colors() function.

The turtle_lty() and turtle_lwd() functions change the type and the width of the line. To change the type of the path pass as an argument a number from 0 to 6 – each denotes 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.



```
> 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! Just use the turtle_status() function: it returns the current drawing settings. 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 it points try turtle_getpos() and turtle_getangle() functions, respectively.

```
> turtle_getpos()
```

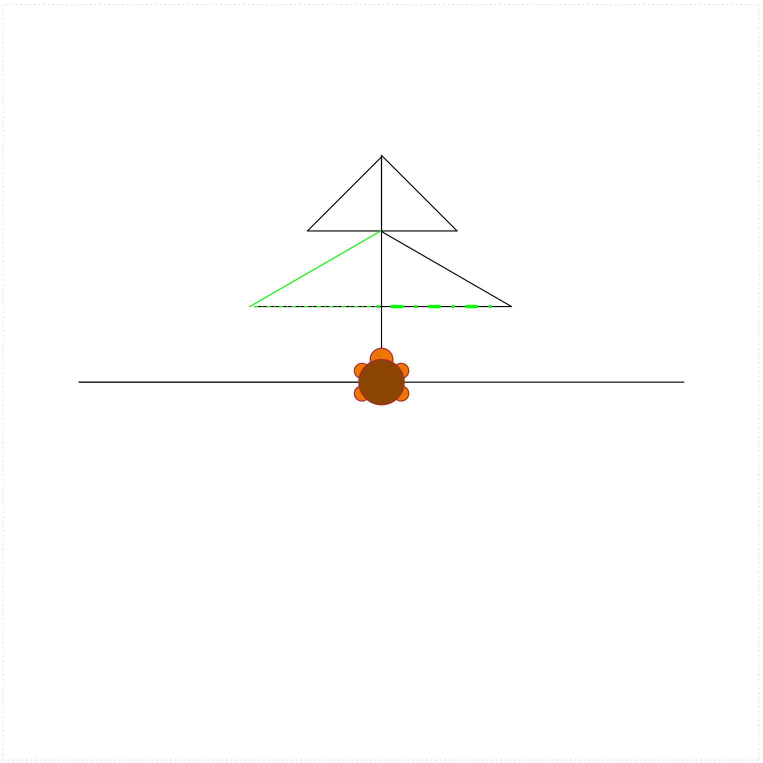
```
       x        y
64.55999 60.00000
```

```
> turtle_getangle()
```

```
angle
 -630
```

turtle_reset and turtle_goto. If you wish to place the Turtle back at the starting position and reset all of the graphical parameters, call the turtle_reset() function.

The turtle_goto() function, on the other hand, asks the Turtle to go to the desired position.

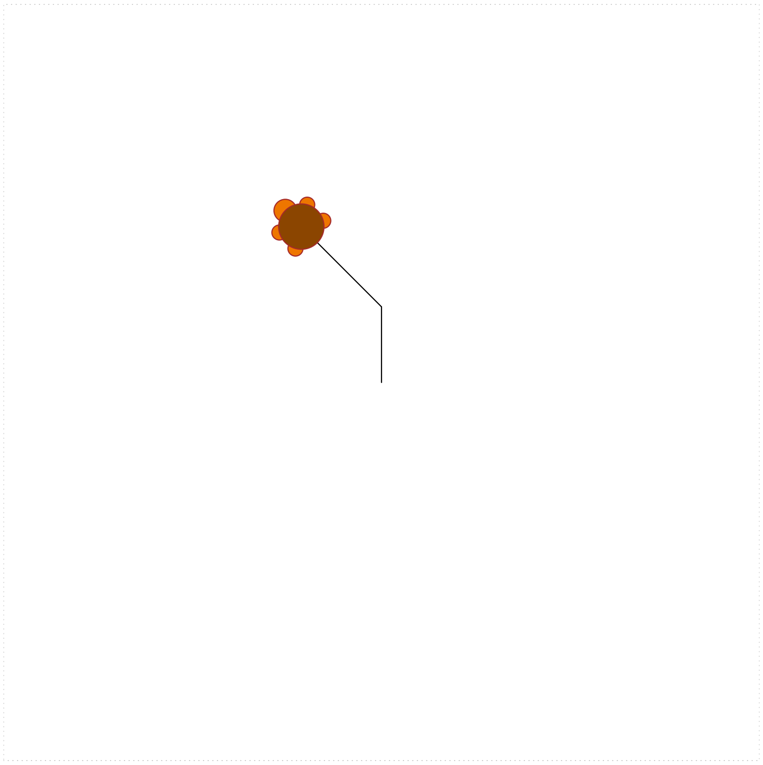


```
> turtle_reset()
> turtle_goto(x=10,y=50)
> turtle_goto(x=90,y=50)
> turtle_reset()
```

2.3 Advanced Usage of the Package

Now you familiar with the basics. There are some more advanced ways to use the package. The (turtle_do()) function is designed for calling more complicated plot expressions.

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

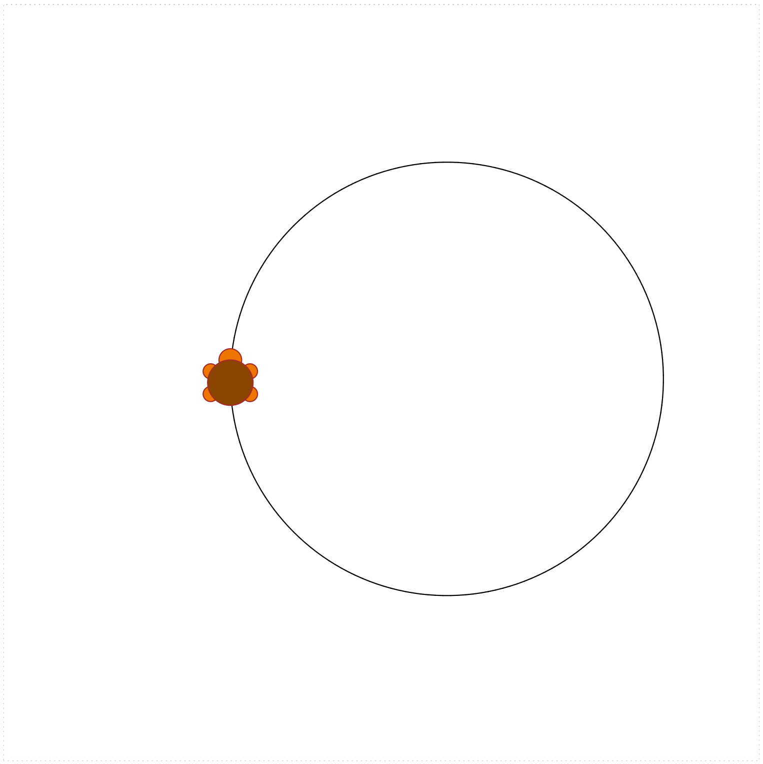


You may ask why bother using such a function if the result is the same as while using three separate commands (in this case this would be turtle_move(10); turtle_turn(45); turtle_move(15)). The thing is that this function hides the Turtle before evaluating the expression, thus the time that expr uses 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 action several times. In such a case, we can use the for loop. The syntax is as follows: for(i in 1:100){ expr}. Such an expression will evaluate expr 100 times. For example:

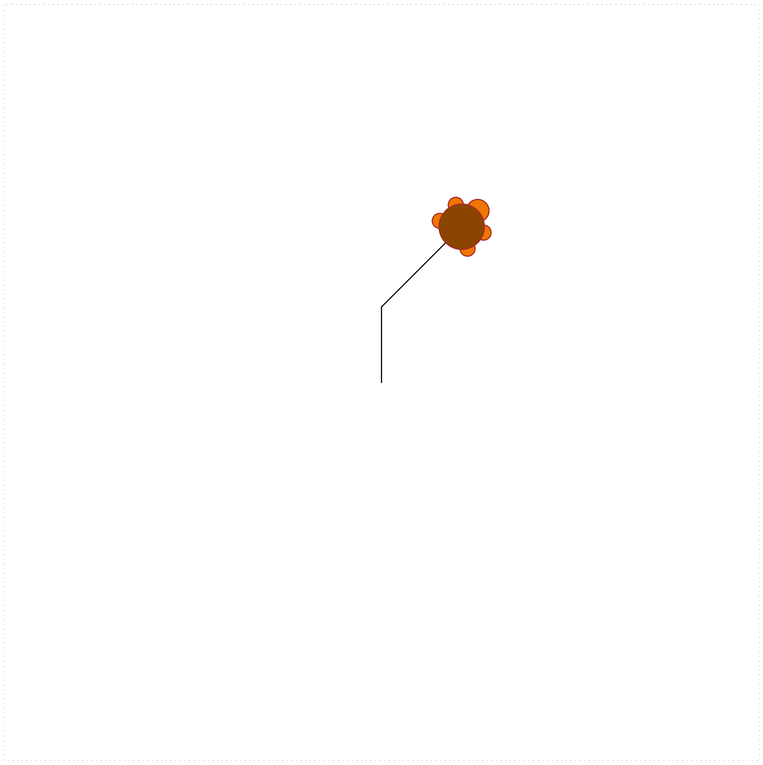


```
> turtle_init()
> turtle_up()
> turtle_goto(x=30,y=50)
> turtle_down()
> turtle_hide()
> for(i in 1:180){
+     turtle_forward(dist=1)
+     turtle_right(angle=2)
+ }
> turtle_show()
```

We strongly recommend to call each for loop always within turtle_do().

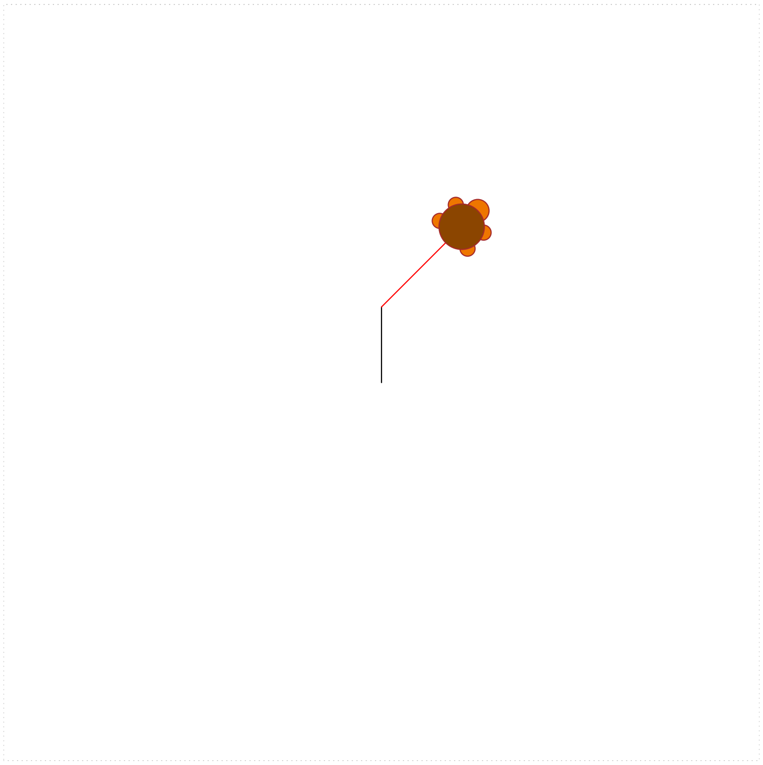
3.2 If

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) {}. When the condition is fulfilled the sequence of actions you put between the curly bracket is called. Let's see an example.



```
> turtle_init()
> turtle_forward(dist=10)
> x <- 1 # or e.g. runif(1) - a random number
> 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 otherwise? There is an answer to it too. After the closing curly braces you write `else` {} and again between the brackets there are the functions you would like to call.



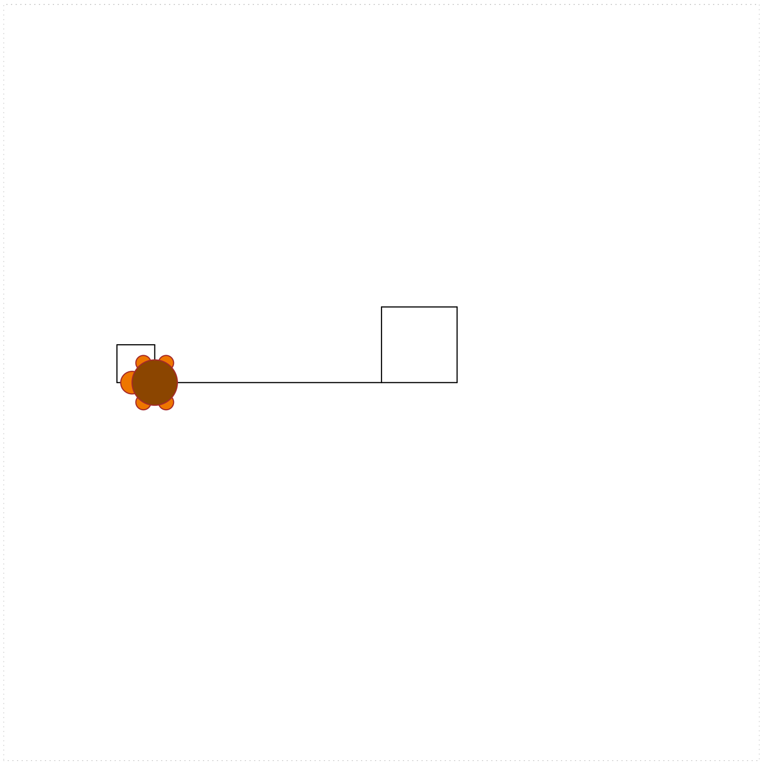
```
> turtle_init()
> turtle_forward(dist=10)
> x<-runif(1) #this function returns a random value between 0 and 1, see ?runif
> if(x>0.5){
+    turtle_right(angle=45)
+    turtle_col(col="red")
+ } else {
+    turtle_left(angle=45)
+    turtle_lwd(lwd=3)
+    turtle_col(col="purple")
+ }
> turtle_forward(dist=15)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

3.3 Functions

Sometimes it is desirable to "store" a sequence of expressions for further use. For example, if you'd like to draw many squares, you can write a custom function so that it can be called many times. For example:

```
> turtle_square <- function(r){
+    for(i in 1:4){
+       turtle_forward(r)
+       turtle_right(90)
+    }
+ }
```

turtle_square is the name of the function. The parameters of the function are listed within the round brackets (separated by commas).



```
> turtle_init()
> turtle_square()
> turtle_left(90)
> turtle_forward(30)
> turtle_square(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.

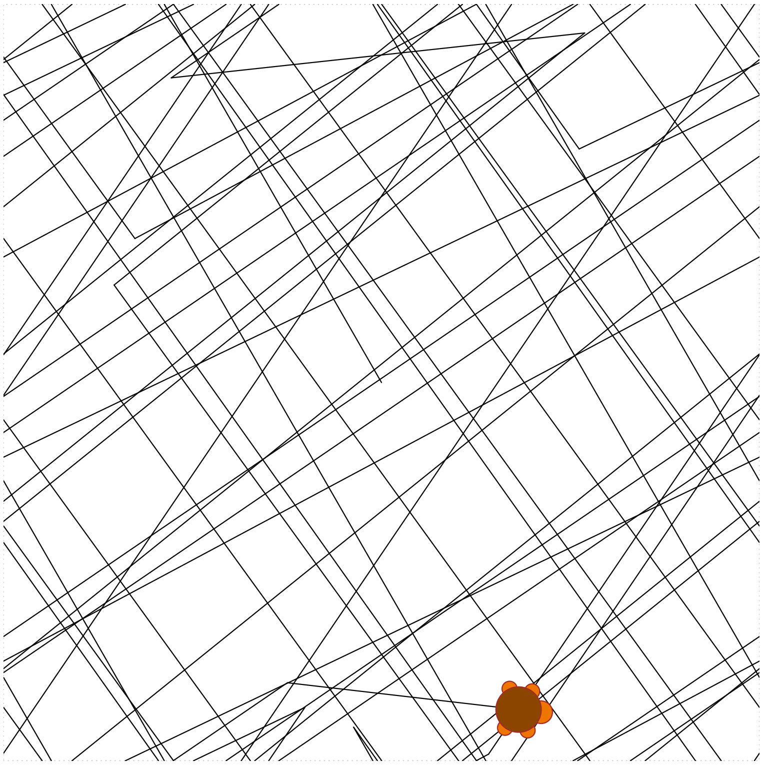
Fractals make perfect examples of the power of recursion. Usually, a fractal is an image which at every scale exhibits the same (or very similar) structure. In Section 4 you have some typical examples of fractals – the fractal tree, the Koch snowflake and the Sierpiński triangle.

4 Examples

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

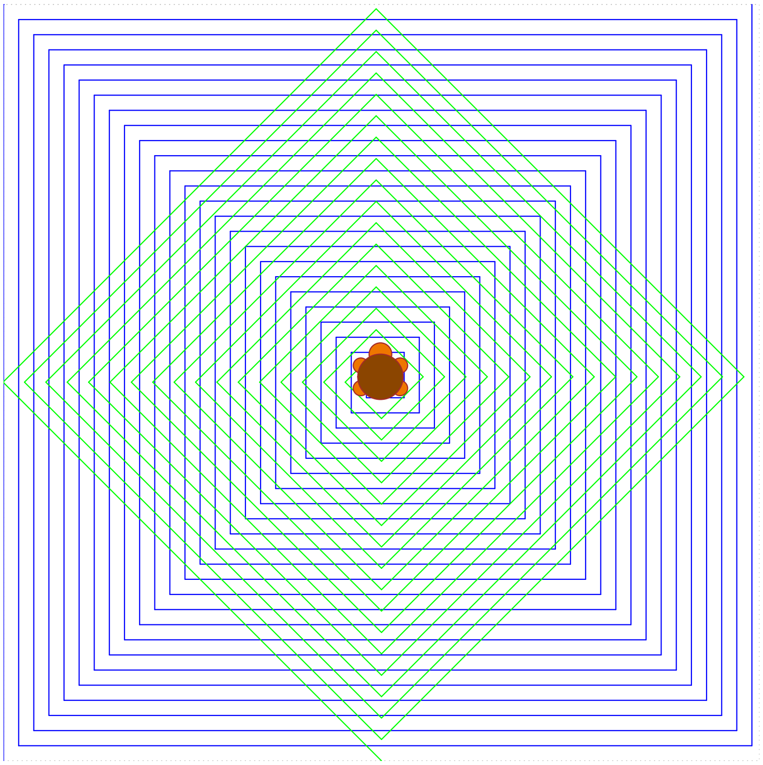
4.1 Random Lines

The first example generates random lines.



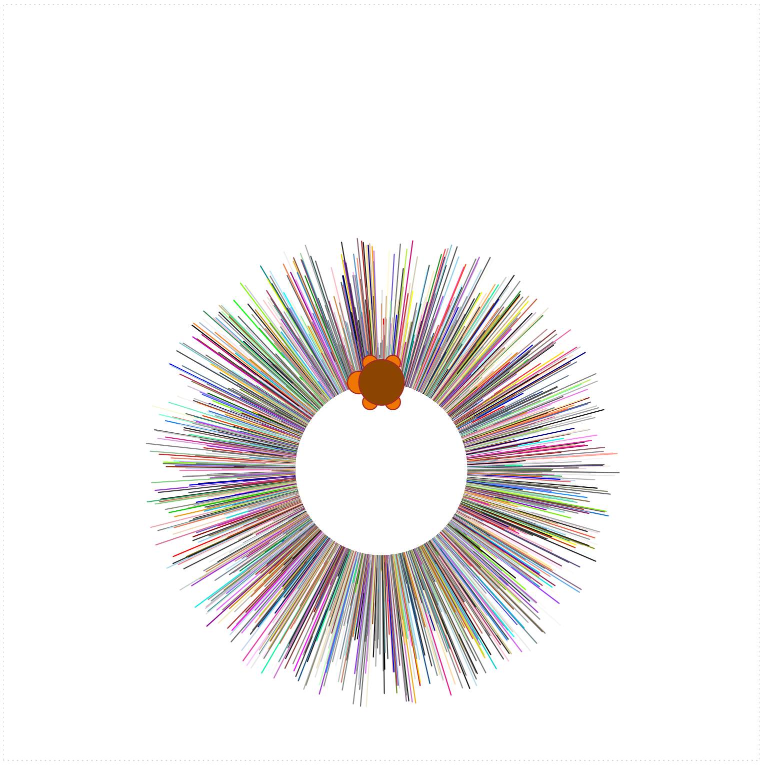
```
> turtle_init(100, 100, mode = "cycle")
> for(i in 1:10){
+   turtle_left(runif(1,0,360))
+   turtle_forward(runif(1, 0, 1000))
+ }
```

4.2 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)
```

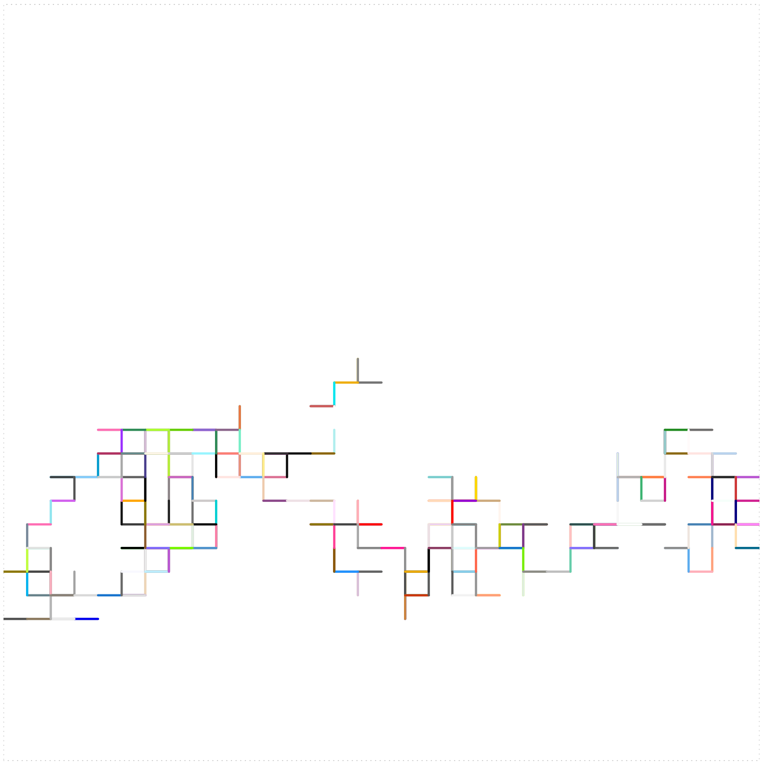
4.3 The Turtle Rainbow Star



```
> 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()
```

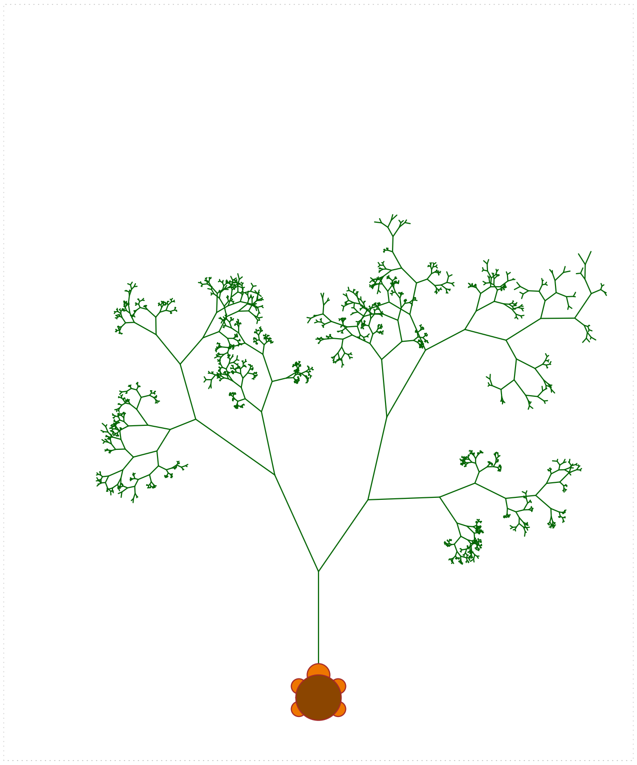
4.4 The Turtle Brownian

This example is inspired by Brownian motion.



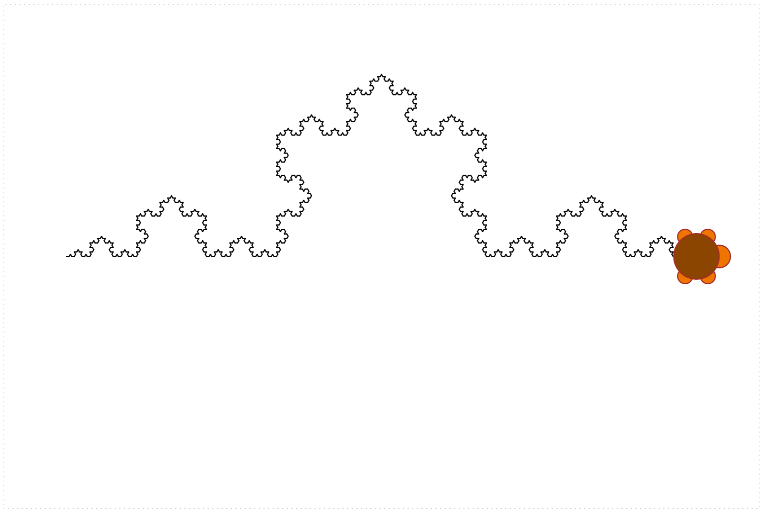
```
> turtle_brownian <- function(steps=100, length=10){
+    turtle_lwd(2)
+    angles <- sample(c(90,270,180,0), steps,replace=TRUE)
+    coll <- sample(1:657, steps, replace=TRUE)
+    for (i in 1:steps){
+       turtle_left(angles[i])
+       turtle_col(colors()[coll[i]])
+       turtle_forward(length)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

4.5 The Fractal Tree



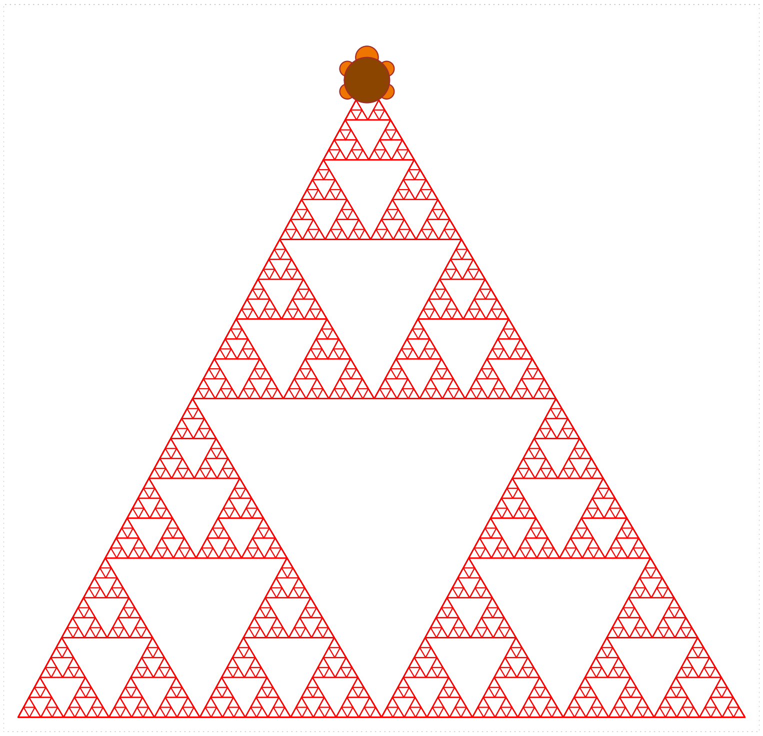
```
> fractal_tree <- function(s=100, n=2) {
+    if (n <= 1) {
+       turtle_forward(s)
+       turtle_up()
+       turtle_backward(s)
+       turtle_down()
+    }
+    else {
+       turtle_forward(s)
+       a1 <- runif(1, 10, 60)
+       turtle_left(a1)
+       fractal_tree(s*runif(1, 0.25, 1),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

4.6 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()
```

4.7 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)
```