# Turtle Graphics

Version 5.3.2

January 31, 2013

Turtle graphics are available in two forms: traditional imperative turtle operations that draw into a fixed window, and functional turtle operations that consume and produce a turtle picture.

# Contents

# 1 Traditional Turtles

```
(require graphics/turtles)
```

To use any of the turtle drawing functions, you first need to initialize the turtles by calling `(turtles #t)`.

```
(turtles on?) → void?
  on? : any/c
(turtles) → void?
```

Shows and hides the turtles window based on *on?*. If *on?* is not supplied, the state is toggled.

```
(move n) → void?
  n : real?
```

Moves the turtle *n* pixels without drawing.

```
(draw n) → void?
  n : real?
```

Moves the turtle *n* pixels and draws a line on the path.

```
(erase n) → void?
  n : real?
```

Moves the turtle *n* pixels and erase along the path.

```
(move-offset h v) → void?
  h : real?
  v : real?
(draw-offset h v) → void?
  h : real?
  v : real?
(erase-offset h v) → void?
  h : real?
  v : real?
```

Like `move`, `draw`, and `erase`, but using a horizontal and vertical offset from the turtle's current position.

```
(turn theta) → void?
  theta : real?
```

Turns the turtle *theta* degrees counter-clockwise.

```
(turn/radians theta) → void?
  theta : real?
```

Turns the turtle *theta* radians counter-clockwise.

```
(clear) → void?
```

Erases the turtles window.

```
(home) → void?
```

Leaves only one turtle, in the start position.

```
(split expr ...)
```

Spawns a new turtle where the turtle is currently located. In order to distinguish the two turtles, only the new one evaluates *expr*. For example, if you start with a fresh turtle-window and evaluate:

```
(split (turn/radians (/ pi 2)))
```

you will have two turtles, pointing at right angles to each other. Continue with

```
(draw 100)
```

You will see two lines. Now, if you evaluate those two expression again, you will have four turtles, etc.

```
(split* expr ...)
```

Like `(split expr ...)`, except that one turtle is created for each *expr*.

For example, to create two turtles, one pointing at $\pi/2$ and one at $\pi/3$, evaluate

```
(split* (turn/radians (/ pi 3)) (turn/radians (/ pi 2)))
```

```
(tprompt expr ...)
```

Limits the splitting of the turtles. Before *expr* is evaluated, the state of the turtles (how many, their positions and headings) is "checkpointed." Then *expr* is evaluated, and then the state of the turtles is restored, but all drawing that may have occurred during execution of *expr* remains.

For example

```
(tprompt (draw 100))
```

moves a turtle forward 100 pixel while drawing a line, and then moves the turtle be immediately back to its original position. Similarly,

```
(tprompt (split (turn/radians (/ pi 2))))
```

splits the turtle into two, rotates one 90 degrees, and then collapses back to a single turtle.

The fern functions below demonstrate more advanced use of `tprompt`.

```
(save-turtle-bitmap name kind) → void?
  name : (or/c path-string? output-port?)
  kind : (or/c 'png 'jpeg 'xbm 'xpm 'bmp)
```

Saves the current state of the turtles window in an image file.

```
turtle-window-size : exact-positive-integer?
```

The size of the turtles window.

## 1.1 Examples

```
(require graphics/turtle-examples)
```

The `graphics/turtle-examples` library's source is meant to be read, but it also exports the following examples. To display these examples, first initialize the turtle window with `(turtles #t)`.

```
(regular-poly sides radius) → void?
  sides : exact-nonnegative-integer?
  radius : real?
```

Draws a regular poly centered at the turtle with *sides* sides and with radius *radius*.

```
(regular-polys n s) → void?
  n : exact-nonnegative-integer?
  s : any/c
```

Draws *n* regular polys each with *n* sides centered at the turtle.

```
(radial-turtles n) → void?
  n : exact-nonnegative-integer?
```

Places $2^n$ turtles spaced evenly pointing radially outward.

```
(spaced-turtles n) → void?
  n : exact-nonnegative-integer?
```

Places $2^n$ turtles evenly spaced in a line and pointing in the same direction as the original turtle.

```
(spokes) → void?
```

Draws some spokes, using `radial-turtles` and `spaced-turtles`.

```
(gapped-lines) → void?
```

Draw a bunch of parallel line segments, using `spaced-turtles`.

```
(spyro-gyra) → void?
```

Draws a spyro-grya reminiscent shape.

```
(neato) → void?
```

As the name says...

```
(graphics-bexam) → void?
```

Draws a fractal that came up on an exam given at Rice in 1997 or so.

```
sierp-size : real?
```

A constant that is a good size for the `sierp` procedures.

```
(sierp sierp-size) → void?
  sierp-size : real?
(sierp-nosplit sierp-size) → void?
  sierp-size : real?
```

Draws the Sierpinski triangle in two different ways, the first using `split` heavily. After running the first one, try executing `(draw 10)`.

```
koch-size : real?
```

A constant that is a good size for the `koch` procedures.

```
(koch-split koch-size) → void?
  koch-size : real?
(koch-draw koch-size) → void?
  koch-size : real?
```

Draws the same Koch snowflake in two different ways.

```
(lorenz a b c) → void?
  a : real?
  b : real?
  c : real?
```

Watch the Lorenz attractor (a.k.a. butterfly attractor) initial values `a`, `b`, and `c`.

```
(lorenz1) → void?
```

Calls `lorenz` with good initial values.

```
(peano peano-size) → void?
  peano-size : real?
```

Draws the Peano space-filling curve.

```
(peano-position-turtle) → void?
```

Moves the turtle to a good place to prepare for a call to `peano`.

```
peano-size : exact-nonnegative-integer?
```

One size to use with `peano`.

```
fern-size : exact-nonnegative-integer?
```

A good size for the `fern1` and `fern2` functions.

```
(fern1 fern-size) → void?
  fern-size : exact-nonnegative-integer?
(fern2 fern-size) → void?
  fern-size : exact-nonnegative-integer?
```

Draws a fern fractal.

For `fern1`, you will probably want to point the turtle up before running this one, with something like:

```
(turn/radians (- (/ pi 2)))
```

For `fern2`, you may need to backup a little.

# 2 Value Turtles

```
(require graphics/value-turtles)
```

The value turtles are a variation on traditional turtles. Rather than having just a single window where each operation changes the state of that window, in the graphics/value-turtles library, the entire turtles window is treated as a value. This means that each of the primitive operations accepts, in addition to the usual arguments, a turtles-window value; instead of returning nothing, each returns a turtles-window value.

```
(turtles  width
          height
         [init-x
          init-y
          init-angle]) → turtles?
  width : real?
  height : real?
  init-x : real? = (/ width 2)
  init-y : real? = (/ height 2)
  init-angle : real? = 0
```

Creates a new turtles window with the given *width* and *height*. The remaining arguments specify position of the initial turtle and the direction in radians (where 0 is to the right).

```
(turtles? v) → boolean?
  v : any/c
```

Determines if *v* is a turtles drawing.

```
(move n turtles) → turtles?
  n : real?
  turtles : turtles?
```

Moves the turtle *n* pixels, returning a new turtles window.

```
(draw n turtles) → turtles?
  n : real?
  turtles : turtles?
```

Moves the turtle *n* pixels and draws a line along the path, returning a new turtles window.

```
(erase n turtles) → turtles?
  n : real?
  turtles : turtles?
```

Moves the turtle *n* pixels and erases a line along the path, returning a new turtles window.

```
(move-offset h v turtles) → turtles?
  h : real?
  v : real?
  turtles : turtles?
(draw-offset h v turtles) → turtles?
  h : real?
  v : real?
  turtles : turtles?
(erase-offset h v turtles) → turtles?
  h : real?
  v : real?
  turtles : turtles?
```

Like `move`, `draw`, and `erase`, but using a horizontal and vertical offset from the turtle's current position.

```
(turn theta turtles) → turtles?
  theta : real?
  turtles : turtles?
```

Turns the turtle *theta* degrees counter-clockwise, returning a new turtles window.

```
(turn/radians theta turtles) → turtles?
  theta : real?
  turtles : turtles?
```

Turns the turtle *theta* radians counter-clockwise, returning a new turtles window.

```
(merge turtles1 turtles2) → turtles?
  turtles1 : turtles?
  turtles2 : turtles?
```

The `split` and `tprompt` forms provided by `graphics/turtles` aren't needed for `graphics/value-turtles`, since the turtles window is a value.

Instead, the `merge` accepts two turtles windows and combines the state of the two turtles windows into a single window. The new window contains all of the turtles of the previous two windows, but only the line drawings of the first turtles argument.

```
(clean turtles) → turtles?
  turtles : turtles?
```

Produces a turtles like *turtles*, but with only a single turtle, positioned in the center.

## 2.1 Examples

```
(require graphics/value-turtles-examples)
```

The graphics/turtle-examples library's source is meant to be read, but it also exports the following examples.

```
(radial-turtles n turtles) → turtles?
  n : exact-nonnegative-integer?
  turtles : turtles?
```

Places $2^n$ turtles spaced evenly pointing radially outward.

```
(spaced-turtles n turtles) → turtles?
  n : exact-nonnegative-integer?
  turtles : turtles?
```

Places $2^n$ turtles evenly spaced in a line and pointing in the same direction as the original turtle.

```
(neato turtles) → turtles?
  turtles : turtles?
```

As the name says...

```
(regular-poly sides radius turtles) → turtles?
  sides : exact-nonnegative-integer?
  radius : real?
  turtles : turtles?
```

Draws a regular poly centered at the turtle with *sides* sides and with radius *radius*.

```
(regular-polys n s turtles) → turtles?
  n : exact-nonnegative-integer?
  s : any/c
  turtles : turtles?
```

Draws *n* regular polys each with *n* sides centered at the turtle.

```
(spokes turtles) → turtles?
  turtles : turtles?
```

Draws some spokes, using radial-turtles and spaced-turtles.

```
(spyro-gyra turtles) → turtles?
  turtles : turtles?
```

Draws a spyro-grya reminiscent shape.