TinyTurtle API Documentation

v0.90
by Piotr Kowalewski (komame), December 2025.

1. Architecture & Concept

`tinyturtle` is a lightweight implementation of the classic **turtle** module, specifically optimized for the HP Prime graphing calculator Python environment.

This module is built directly on top of the `hpprime` graphic primitives. It utilizes **double buffering** (rendering to an off-screen graphic object first) to ensure flicker-free animations and implements a "dirty rectangle" system to update only changed parts of the screen, maximizing speed on the calculator's hardware.

Some methods that usually belong to Screen (like bgcolor, tracer, update) are methods of the Turtle instance.

2. Initialization

`Turtle(screen_g=0, buffer_g=1, visible=True, viewport=False, grid=0)`

Initializes the turtle and the graphics environment.

- **screen_g** *(int)*: Target graphic object (usually 0 for the main screen).
- **buffer_g** *(int)*: Off-screen buffer for double-buffering (default 1).
- **viewport** *(bool)*: If True, enables "recording mode". Commands are not drawn immediately but saved to a display list for later viewing/zooming via show().
- **visible** *(bool)*: Sets the initial visibility of the turtle cursor. If True (default), the turtle starts visible. It can be toggled later using hideturtle() or showturtle().
- **grid** *(int or tuple)*: Configures the background grid.
    - **As an Integer (Mode):**
        - 0: No grid.
        - 1: Dot grid (10px spacing).
        - 2: Square grid (20px spacing).
        - 3: Triangular grid (20px spacing)
    - **As a Tuple (mode, color):**
      Allows setting the mode and line color simultaneously.
        - Example: grid=(2, 'red') draws a large square grid with red lines.
        - Example: grid=(3, 0x00FF00) draws a triangular green grid.

Note: If more than one turtle is created, then at this stage of the implementation all turtles must have unique assignments for both `screen_g` and `buffer_g`. Using shared buffers may lead to individual turtles overwriting each other’s graphical content.

3. Movement (Relative & Absolute)

`forward(distance)` | **`fd(distance)`**

Moves the turtle forward by the specified `distance`.

`backward(distance)` | `bk(distance)` | `back(distance)`

Moves the turtle backward by `distance` (opposite to its heading). Do not change the turtle's heading.

`right(angle)` | `rt(angle)`

Turns the turtle clockwise by `angle` degrees.

`left(angle)` | `lt(angle)`

Turns the turtle counter-clockwise by `angle` degrees.

`goto(x, y=None)` | `setpos(x, y)` | `setposition(x, y)`

Moves to absolute coordinates `(x, y)`.

- Draws a line if the pen is down.
- Accepts two arguments `goto(10, 20)` or a tuple `goto((10, 20))`.

`setheading(angle)` | `seth(angle)`

Sets the turtle's absolute orientation (0=East, 90=North, 180=West, 270=South).

`home()`

Moves the turtle to `(0, 0)` and sets heading to `0`.

`circle(radius, extent=None, steps=None)`

Draws a circle.

- radius: Positive = counter-clockwise (center is to the left). Negative = clockwise.
- extent: Angle to draw (default 360).
- steps: Number of polygon edges to approximate the circle (default is calculated automatically).

4. Pen Control & Appearance

`penup()` | `pu()` | `up()`

Lifts the pen. Moving will not draw lines.

`pendown()` | `pd()` | `down()`

Lowers the pen. Moving will draw lines.

`width(w=None)` | `pensize(w=None)`

Sets or gets the line thickness.

- Set: `t.width(5)` sets thickness to 5px.
- Get: `t.width()` returns current thickness (int).

`isvisible()`

Returns `True` if the turtle cursor (the shape) is visible.

`hideturtle()` | `ht()`

Hides the turtle cursor. Increases drawing speed significantly.

`showturtle()` | `st()`

Makes the turtle cursor visible.

5. Color Control

Return values depend on `colormode()`.

- If `colormode(1.0)`: RGB tuples are floats `(0.0, 0.5, 1.0)`.
- If `colormode(255)`: RGB tuples are integers `(0, 128, 255)`.

`colormode(mode)`

Sets the RGB scale.

- Input: `1.0` or `255`.
- Returns: The newly set mode.

`pencolor(*args)`

Sets or gets the outline color.

- Set: `pencolor("red")`, `pencolor('#FF0000')`, `pencolor(255, 0, 0)`.
- Get: `pencolor()` returns current pen color tuple.

`fillcolor(*args)`

Sets or gets the fill color.

- Set: `fillcolor("blue")`.
- Get: `fillcolor()` returns current fill color tuple.

`color(*args)`

Sets or gets both pen and fill colors.

- Set: `color("red")` (both red) OR `color("red", "green")` (pen red, fill green).
- Get: Returns `(pen_color, fill_color)`.

`bgcolor(*args)`

Sets or gets the background color.

- Set: `bgcolor("black")`. Note: This immediately clears the screen.
- Get: Returns current background color.

6. Drawing Primitives & Text

`dot(size=None, color=None)` | **`stamp_dot(size=None, color=None)`**

Draws a filled circular dot at the current position.

- size: Diameter (defaults to `pensize + 4`).

`write(arg, move=False, align="left", font_size=1)`

Writes text using the internal bitmap font.

- arg: String or number to write.
- move: If `True`, turtle moves to the end of the text.
- align: `"left"`, `"center"`, `"right"`.
- font_size: float (1=Standard 8px high, 1.5=12px high, 2=16px high, etc.).

7. Filling Shapes

`begin_fill()`

Starts recording the path for filling.

`end_fill(fill_color=None)`

Fills the recorded path.

- fill_color: Optional override.
- redraw_outline: If `True` (default), redraws the border on top of the fill to ensure clean edges.

`cancel_fill()`

Aborts the fill recording without drawing.

8. Querying State (Getters)

`position()` | **`pos()`**

Returns `(x, y)` as a tuple.

`xcor()`

Returns the X coordinate (float).

`ycor()`

Returns the Y coordinate (float).

`heading()` | `getheading()`

Returns the current angle in degrees (float).

`distance(x, y=None)`

Returns the distance from the turtle to the point `(x, y)`.

`towards(x, y=None)`

Returns the angle (degrees) from the turtle towards the point `(x, y)`.

9. Animation & Screen Control

`tracer(n=None, delay=None)`

Controls drawing optimization.

- n: Update screen every `n`th frame.
  - `1`: Default (smooth animation).
  - `0`: Disable auto-update (fastest, requires manual `update()`).
- delay: Sets delay in milliseconds (same as `speed()`).
- Returns: Current tracer step `n` if called without args.

`update()`

Manually refreshes the screen.

- Required when `tracer(0)` is active to show the result.

`speed(s=None)`

Sets/Gets animation speed.

- Set: `0` or `'fastest'` (no delay), `1`-`10` (slowest to fast), or `'slowest'`, `'slow'`, `'normal'`, `'fast'`, `'fastest'`.
- Get: Returns current delay in ms (int) or string name.

`wait(ms)`

Pauses execution for `ms` milliseconds.

`clear()`

Clears the drawing but keeps the turtle's state and position.

`reset()`

Clears the drawing and resets the turtle to Home (0,0), black color, default speed.

10. Input & Viewport

`read_key()`

Pauses program execution until a key is pressed on the calculator.
Returns: Key code (int).

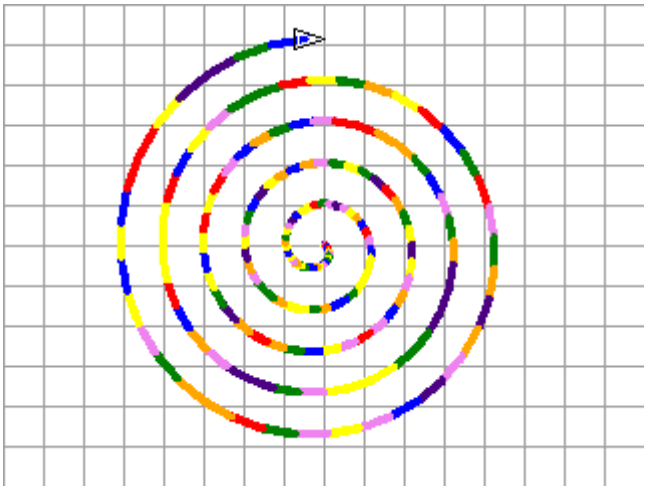
`show()`

Viewport Mode Only.
Starts an interactive viewer for the recorded commands.

- Navigation: Arrow keys (Pan), `+`/`−` (Zoom).
- Exit: `Enter`, `Esc`, `Backspace`.

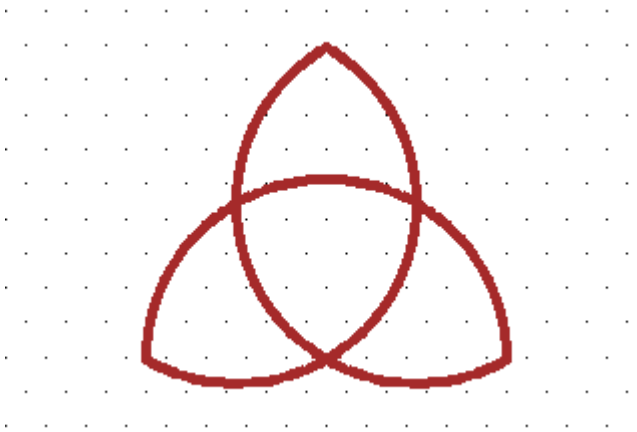
11. Examples

11.1. Random Color Spiral



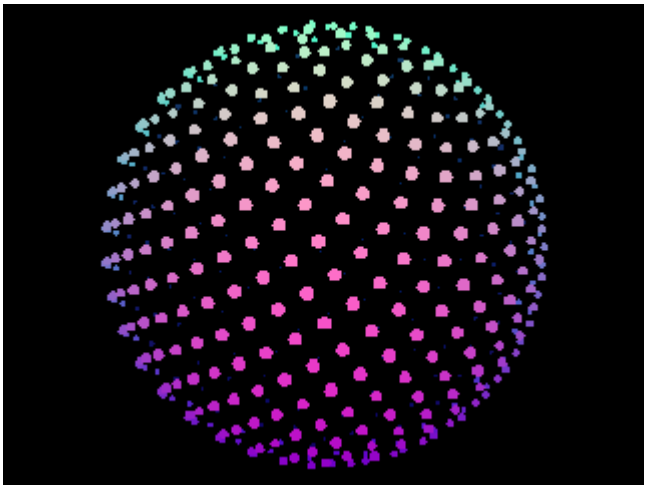
```
from tinyturtle import *
from urandom import choice

t = Turtle(grid=2)
colors = ['red','orange','yellow','green','blue','indigo','violet']
t.speed(2)
t.pensize(4)
for i in range(180):
    t.color(choice(colors))
    t.forward(0.1+i*0.1)
    t.right(10)
t.read_key()
```

11.2. Triple Arc Pattern



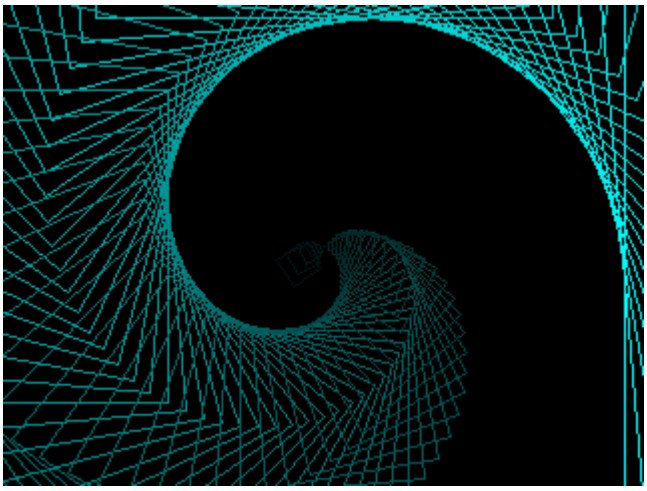
```
from tinyturtle import *

t = Turtle(visible=False, grid=(3,'black'))
t.speed(1)
t.pu()
t.goto(-90,-70)
t.pd()
t.width(5)
t.color('brown')
t.right(30)
for i in range(3):
    t.circle(90,180)
    t.left(60)
t.done()
```

11.3. Cyber Sphere



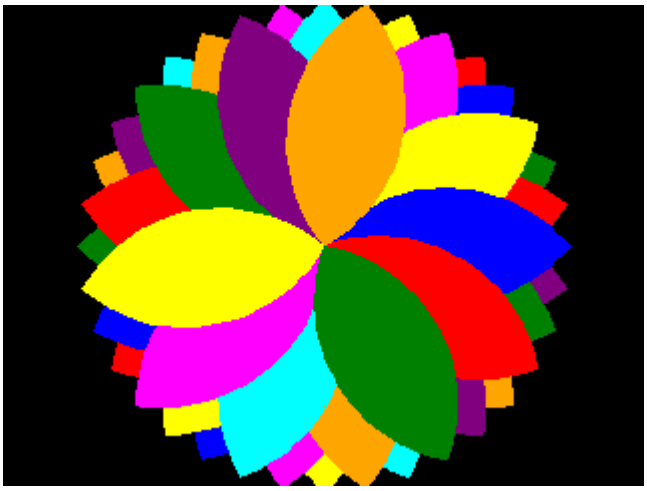
```
from tinyturtle import *

t = Turtle(visible=False)
t.bgcolor("black")

num_points = 500
radius = 110
golden_angle = math.pi * (3 - math.sqrt(5))
rot_x = 0.5
rot_y = 0.8
cx, sx = math.cos(rot_x), math.sin(rot_x)
cy, sy = math.cos(rot_y), math.sin(rot_y)
draw_list = []
for i in range(num_points):
    y = 1 - (i / (num_points - 1)) * 2
    r = math.sqrt(1 - y * y)
    theta = golden_angle * i
    x = math.cos(theta) * r * radius
    z = math.sin(theta) * r * radius
    y *= radius
    rx = x * cy - z * sy
    rz = x * sy + z * cy
    ry = y * cx - rz * sx
    rz_final = y * sx + rz * cx
    depth_factor = (rz_final + radius) / (2 * radius)
    r_val = int(255 * depth_factor)
    g_val = int(255 * ((ry / radius + 1) / 2))
    b_val = 200
    if depth_factor < 0.3:
        r_val //= 4
        g_val //= 4
        b_val //= 2
    dot_size = 1 + int(7 * depth_factor)
    draw_list.append((rz_final, rx, ry, r_val, g_val, b_val, dot_size))
draw_list.sort()

for point in draw_list:
    _, px, py, r, g, b, size = point
    t.pencolor(r, g, b)
    t.penup()
    t.goto(px, py)
    t.pendown()
    t.dot(size)
t.done()
```

11.4. 3D Tunnel



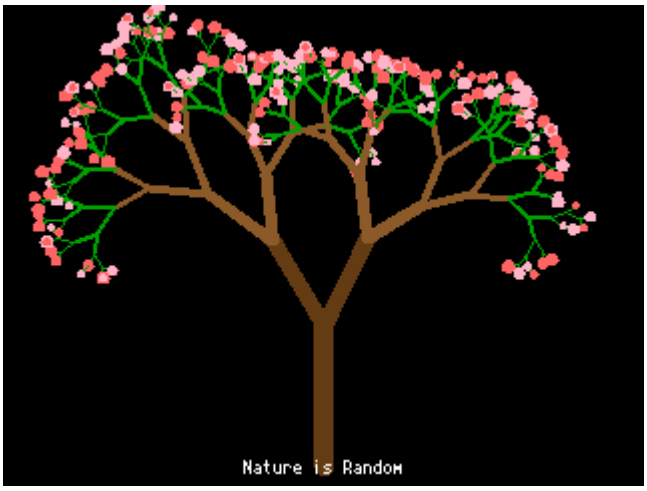
```
from tinyturtle import *

t = Turtle(visible=False)
t.bgcolor("black")

size = 300
angle = 0

for i in range(80):
    brightness = int(255 - (i * 3))
    if brightness < 0: brightness = 0
    t.pencolor(0, brightness, brightness)
    t.penup()
    t.home()
    t.setheading(angle)
    t.forward(size / 2)
    t.right(90)
    t.forward(size / 2)
    t.right(180)
    t.pendown()
    for _ in range(4):
        t.forward(size)
        t.right(90)
    size -= 4
    angle += 5
t.done()
```

11.5. Mandala



```
from tinyturtle import *

t = Turtle(visible = False)

colors = ["red", "orange", "yellow", "green", "cyan", "blue", "purple", "magenta"]

t.bgcolor("black")
t.speed(0)
t.width(1)
for i in range(36):
    t.color(colors[i % 8])
    t.begin_fill()
    t.circle(80, 100)
    t.left(80)
    t.circle(80, 100)
    t.end_fill()
    t.right(170)

t.pencolor("white")
t.done()
```

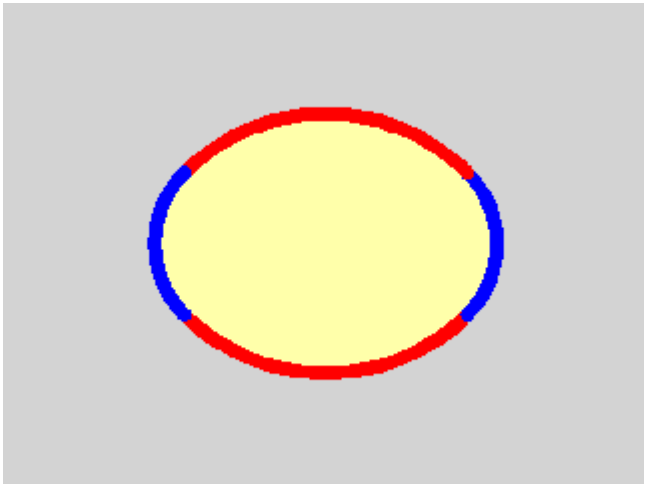
11.6. Random Tree




```
from tinyturtle import *
from urandom import randint

t = Turtle(visible=False)

t.bgcolor("black")
t.left(90)
t.up()
t.backward(110)
t.down()

def random_branch(length, thickness):
    if length < 5:
        if randint(0, 1) == 0:
            t.pencolor(255, 100, 100)
        else:
            t.pencolor(255, 180, 200)
        t.dot(randint(4, 7))
        return

    if thickness > 6:
        t.pencolor(100, 60, 20)
    elif thickness > 3:
        t.pencolor(140, 90, 40)
    else:
        t.pencolor(0, 160, 0)
    t.width(thickness)
    t.forward(length)

    angle_l = randint(15, 45)
    angle_r = randint(15, 45)
    scale_l = randint(65, 80) / 100.0
    scale_r = randint(65, 80) / 100.0
```

```
    t.left(angle_l)
    random_branch(length * scale_l, thickness * 0.75)
    t.right(angle_l + angle_r)
    random_branch(length * scale_r, thickness * 0.75)
    t.left(angle_r)
    t.up()
    t.backward(length)
    t.down()
random_branch(70, 10)
t.pencolor("white")
t.up()
t.goto(0, -115)
t.write("Nature is Random", align = "center")
t.done()
```

11.7. Arc Oval

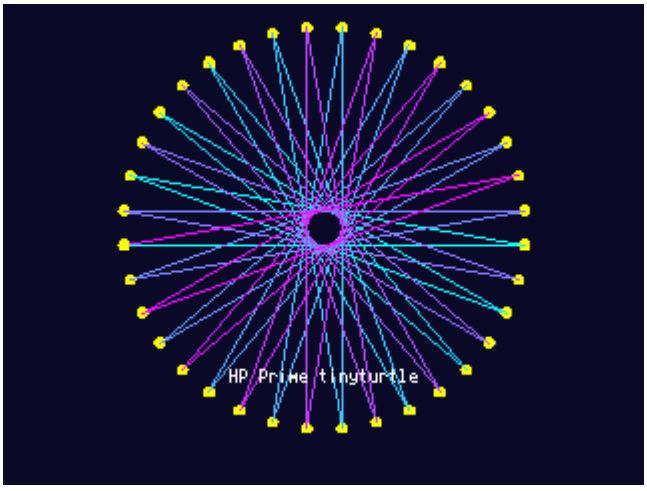


```
from tinyturtle import *

t = Turtle(visible = False)

t.bgcolor('lightgray')
t.up()
t.goto(-70, -35)
t.width(7)
t.down()
t.begin_fill()
t.seth(-45)
t.color('red')
t.circle(100, 90)
t.color('blue')
t.circle(50, 90)
t.color('red')
t.circle(100, 90)
```

```
t.color('blue')
t.circle(50, 90)
t.fillcolor(0xFFFFAA)
t.end_fill()
t.done()
```

11.8. Viewport Star

When viewport mode is active, use +/- keys and arrows to scale/pad the picture.




```
from tinyturtle import *

t = Turtle(visible = False, viewport = True)

t.reset()
t.bgcolor(10, 10, 40)
points = 36
angle = 170
size = 200
t.up(); t.goto(-100, 0); t.down()
for x in range(points):
    r = int(255 * (x / points))
    g = int(255 * ((points - x) / points))
    b = 255
    t.pencolor(r, g, b)
    t.forward(size)
    t.left(angle)
    t.dot(6, (255, 255, 0))
t.up()
t.goto(0,-70)
t.color('white')
t.write('HP Prime tinyturtle', False, 'center')
t.show()
```