CS 1101 Computer Science I

Instructor Muller

stddraw API

(DRAFT of 1/15/2016)

This document describes the *application programmer interface* (API) for the **stddraw** library. An API describes the set of functions and other resources that are available in a given library. This library was ported from Java to Python by Pine Wu.

The functions in this library will be available for use in your programs if you include the line

```
from stddraw import *
```

in your **.py** file, (just below the header comments) and if you have placed the file **stddraw.py** in the appropriate **site-packages** folder on your computer. The instructions for tracking down the **site-packages** folder are specified in problem set 1.

In order to use the functions in the stddraw library, in your program, you'll first need to create a `Picture`. For example:

```
myPicture = Picture()
```

This binds the variable `myPicture` to a 2-dimensional *canvas* on which shapes can be drawn. The default size of the canvas is 512×512 pixels. The canvas is layed out as a 2-dimensional xy-plane, a *unit square* of size 1.0×1.0 with $(x, y) = (0.0, 0.0)$ referring the lower left corner, $(x, y) = (.5, .5)$ referring the middle of the canvas and $(x, y) = (1.0, 1.0)$ referring the upper right corner.

The system allows you to create **widgets** of various sorts, *buttons*, *labels*, *text fields*, *lines* and shapes of various sorts. These can all be drawn on the canvas. For example, if you wish to draw a square, you can use the `square` function:

```
myPicture = Picture()
myPicture.square(.5, .5, .1)
myPicture.start()
```

It is important to note that the last line, the statement:

```
myPicture.start()
```

at the end of your program, is required to activate the picture.

Types

In the remainder of this document we will use *types* to specify the inputs and outputs to the various functions in stddraw.py. We will use Python's built-in types `int`, `float` and `string`. We will use the symbols `handle`, `event` and `color` to refer to the types of *handles*, *events* and *colors* (resp). Handles and events will be described below. We'll use the symbol `void` to refer to the type of *no value*. Python has built-in types *tuple* and *function*. We'll be a bit more specific, using the notation `int` $*$ `int` as the type for a 2-tuple (or *pair*) of integers, etc and we'll use the right arrow $\rightarrow$ for function types, for example, with `float` $*$ `float` $\rightarrow$ `void` representing the type of a function that accepts a pair of floating point numbers and returns nothing.

Drawing Functions

The drawing functions generally return a `handle` for whatever widget has been drawn. Subsequent calls referring to the handle may change properties of the widget such as color or location. If you need a special color, use the `makeColor` function in (see Special Functions below). The default color for filled figures is 'Black'.

`line` : `float` $*$ `float` $*$ `float` $*$ `float` $*$ `color` $*$ `int` $\rightarrow$ `handle`

The call `line(x0, y0, x1, y1, color='black', penWidth=1)` draws a line of width `penWidth` from `(x0, y0)` to `(x1, y1)`, of the specified color. The default color is black, and the default `penWidth` is 1. The following invocations would all work

```
myPicture.line(x0, y0, x1, y1)
myPicture.line(x0, y0, x1, y1, 'Blue', 3)
myPicture.line(x0, y0, x1, y1, color='Blue')
```

`arc` : `float` $*$ `float` $*$ `float` $*$ `float` $*$ `int` $*$ `int` $\rightarrow$ `handle`

The call `arc(x0, y0, halfWidth, halfHeight, startAngle, degree)` draws an arc from `startAngle` to `startAngle`+`degree` out of an oval centered at `(x0, y0)` of `halfWidth` and `halfHeight`. If `startAngle` is 0, it means the direction is upward.

`filledArc` : `float` $*$ `float` $*$ `float` $*$ `float` $*$ `int` $*$ `int` $*$ `color` $\rightarrow$ `handle`

The call `filledArc(x0, y0, halfWidth, halfHeight, startAngle, degree, color)` draws the same figure as `arc` does, except the arc has a color.

`oval` : `float` $*$ `float` $*$ `float` $*$ `float` $\rightarrow$ `handle`

The call `oval(x0, y0, halfWidth, halfHeight)` draws an oval centered at `(x0, y0)`, of `halfWidth` and `halfHeight`.

`filledOval` : `float` ∗ `float` ∗ `float` ∗ `float` ∗ `color` → `handle`

The call `filledOval(x0, y0, halfWidth, halfHeight, color)` draws the same figure as `oval` does, except the oval has a color.

`circle` : `float` ∗ `float` ∗ `float` → `handle`

The call `circle(x0, y0, radius)` draws a circle centered at `(x0, y0)`, of radius `radius`.

`filledCircle` : `float` ∗ `float` ∗ `float` ∗ `color` → `handle`

The call `filledCircle(x0, y0, radius, color)` draws the same figure as `circle` does, except the circle has a color.

`rectangle` : `float` ∗ `float` ∗ `float` ∗ `float` → `handle`

The call `rectangle(x0, y0, halfWidth, halfHeight)` draws a rectangle centered at `(x0, y0)`, of `halfWidth` and `halfHeight`.

`filledRectangle` : `float` ∗ `float` ∗ `float` ∗ `float` ∗ `color` → `handle`

The call `filledRectangle(x0, y0, halfWidth, halfHeight, color)` draws the same figure as `rectangle` does, except it has a color.

`square` : `float` ∗ `float` ∗ `float` → `handle`

The call `square(x0, y0, radius)` draws a square centered at `(x0, y0)`, of `radius` that is half of its side's length.

`filledSquare` : `float` ∗ `float` ∗ `float` ∗ `color` → `handle`

The call `filledSquare(x0, y0, square, color)` draws the same figure as `square` does, except it has a color.

`polygon` : `float list` ∗ `float list` → `handle`

The call `polygon(xList, yList)` draws a polygon defined by the points (xList[0], yList[0]), (xList[1], yList[1])... (xList[n], yList[n]).

`filledPolygon` : `float list` ∗ `float list` ∗ `color` → `handle`

The call `filledPolygon(xList, yList)` draws the same figure as `polygon` does, except it has a color.

`text` : `float` $*$ `float` $*$ `string` $*$ `string` $\rightarrow$ `handle`

The call `text(x0, y0, message, anchor='sw')` draws `message` starting at `(x0, y0)`, anchored at southwest provide the anchor one of `'n'`, `'s'`, `'w'`, `'e'`, `'nw'`, `'ne'`, `'sw'` or `'se'`. `'ne'`, For anchors, `'n'` means the midpoint of north side overlaps with `(x0, y0)`, 'nw' means the northwest point overlaps with `(x0, y0)`.

`readGif` : `string` $\rightarrow$ `image`

The call `readGif(photoFileName)` returns a Tkinter PhotoImage object, which can be used in combination with drawing function `image` below.

`image` : `float` $*$ `float` $*$ `image` $*$ `string` $\rightarrow$ `handle`

The call `image(x0, y0, photo, anchor='sw')` renders `photo`, anchored at `(x0, y0)` with the provided anchor direction. For example:

```
picture = Picture()
photo = picture.readGif('myPhoto.gif')
handle = picture.image(x0, y0, photo, anchor='s')
```

Manipulating Functions

Given a handle of a widget, there are functions that allow for the alteration of properties of the widget. For example,

```
myPicture = Picture()
mySquare = myPicture.square(.5, .5, .1)        # mySquare has the

myPicture.move(mySquare, .1, .1)               # move the square
```

delete/move/configColor/(square.....), text has a special config method.

`move` : `handle` $*$ `float` $*$ `float` $\rightarrow$ `void`

The call `move(item, x0, y0)` moves `item` to its right by `x0`, and up by `y0`. The values of `x0` and `y0` can be negative.

`delete` : `handle` $\rightarrow$ `void`

The call `delete(handle)` deletes the widget with handle `handle`.

`configColor` : `handle` $*$ `color` $\rightarrow$ `void`

The call `configColor(item, color)` changes the color of the widget with handle `handle`.

`configText` : `handle` $*$ `string` $\rightarrow$ `void`

The call `configText(item, text)` changes the message of a text item.

`wait` : `handle` $*$ `int`$*$`(event` $\rightarrow$ `void)`$\rightarrow$ `void`

The call `wait(handle, milliseconds, event=action)` waits for `milliseconds` milliseconds. If an event `action` is given, it will be performed after that wait.

Events

Most of the widgets can respond to **events** such as mouse clicks. By "responding" to an event, we mean that a function can be executed when the event occurs.

`bind` : `string` $*$`(event` $\rightarrow$ `void)`$\rightarrow$ `void`

A call `bind(eventName, responder)` The value of `eventName` should be one of the strings:

- '$\langle$`Button-1`$\rangle$' means left click,
- '$\langle$`Button-3`$\rangle$' means right click,
- '$\langle$`Enter`$\rangle$' means Mouse on Canvas,
- '$\langle$`Leave`$\rangle$' means mouse leaves Canvas.

Provide `responder` in the following way:

```
def responder(event):
    do something
...
myPicture.bind('<Button-1>', responder)
```

Or as a one-liner:

```
myPicture.bind('<Button-1>', lambda event: do something)
```

Special Functions

`makeColor` : `int` $*$ `int` $*$ `int` $\rightarrow$ `color`

In the call `makeColor(red, green, blue)`, `red`, `green` and `blue` should be between 0 and 255.

`randomColor` : `void` $\rightarrow$ `color`

Returns a random color.

`setW` : `int` → `void`

The call `setW(w)` sets the canvas width to `w`.

`setH` : `int` → `void`

The call `setH(h)` sets the canvas width to h.

`getW` : `void` → `int`

The call `getW()` returns the width of the canvas.

`getH` : `void` → `int`

The call `getH()` returns the height of the canvas.

`clear` : `void` → `void`

The call `clear()` clears the canvas.

`start` : `void` → `void`

The call `start()` activates the picture.