U: Using The LiveWires Modules

Rhodri James Revision 1.16, October 27, 2001

Credits

For the LaTeX source of this sheet, and for more information on LiveWires and on this course, see the LiveWires web site at `http://www.livewires.org.uk/python/`

Introduction

The LiveWires modules provides a lot of help for programmers writing games, but it's not very easy to understand. This sheet is an attempt to demystify the libraries, and to show you how to use them in your own games. It requires a fair bit of experience with Python to understand, but don't worry — if you've used one of our games worksheets then you shouldn't have much of a problem.

This sheet isn't a complete listing of absolutely every detail of every method (function) in the library. That's what Sheet W is for; here, we are just trying to give you the information that will let you use Sheet W effectively.

What Sort Of Game?

Generally speaking, the LiveWires libraries provide help for two different sorts of game. Either you are whizzing around the screen in an unconstrained manner, like when you play *Asteroids* or another shoot-'em-up like that, or you're moving around on a grid of squares, like when you play Chess. If you are thinking of something different, then you'll have to start with the `games` library and do it yourself; if you want to move around on a hexagonal grid, for instance, you're on your own!

The upshot of this is that once you know what sort of game you are writing, you know which section of this document to start in. For general games (as we'll call the shoot-'em-ups and the like), you need the `games` module only. For board games, you'll find a lot of additional help in the `boards` library, though you need `games` as well.

What follows is some general advice in writing games, and how to use the libraries to help you.

General Games

The first thing that *any* game needs is a window in which to act. In our terms, this is the `games.Screen` class. You can only have one window (one `Screen` or more likely a subclass of it), so think about it carefully.

The `Screen` class itself doesn't require much setting up. The only parameters you can supply to its `init_screen` method are the width and height of the window, and for most purposes it's easier to leave those with the default values. Something that you are more likely to want to fiddle with is the window's background. You can either set the colour of the background using the `set_background_colour` method, or you can put a picture in it instead as the Asteroids game does. That needs a short sequence of instructions in your class initialisation method:

```
image = load_image(filename)
self.set_background(image)
```

Your screen subclass is a good place to put all the random variables that you might otherwise make into globals. If you look at any of the games worksheets, you'll see that all of them stick variables like `game_over`, `current_player` and the like into the subclass of `Screen` that they create. You don't have to do this, but it helps protect you against unexpected name clashes and having to declare some variables to be `global` to get around scoping problems.

> *Python's scope rules are beyond the scope (sorry) of this worksheet. The problems that you might encounter are mentioned briefly in Sheet 5. If you really want to know, ask a leader.*

Finally, your screen subclass is what drives the main program loop. All programs that use the `games` library have a main program (as distinct from the class, function and constant definitions) that looks something like this:

```
myscreen = MyScreen(... whatever...)
myscreen.mainloop()
```

That's all. The `games` library (and Pygame underneath it) do all the hard work of running the screen, moving the moving objects and so on.

Things on the Screen

Everything that you can display in the window will be one of the subclasses of `Object` that the library provides: `Sprite`, `Polygon`, `Circle` or `Text`, or one of their subclasses. These tend to fall into one of two categories: things that stay where they're put, and things that move around.

Things That Stay Put

These are things like the central planet in Space War, the walls and bricks of a game of Breakout, status messages and scores. They are very straightforward, simple objects of their type, and can often be used "as is." If you make them into a subclass at all, it's usually because you want to do something when another object collides with them — see the `Planet` object in Space War for an example of that.

Probably the most common objects of this type are `Text` objects used to keep track of the player's score. It is usual to put these in your subclass of `Screen`, where you can easily get at them to update them. There are two trick that you need to know for scores. First, the position that you give to a `Text` object is where the *centre* of the message will go. This usually isn't what you were expecting, and can be a bit of a pain.

Second, `Text` objects will only show text strings, they don't know how to handle numbers. To put a number into a `Text` object you need to convert it to a string somehow, either using Python's `str` function or the "%" operator. `%` looks strange and is a lot harder to understand, but it gives you more control over exactly what goes on the screen.

For example, suppose you set up your game like this:

```
self.score_msg = games.Text(self, x, y, "Score: 0000")
self.score = 0
```

Then suppose you've changed `self.score` to be 13, and you want to change what's shown on the screen to match. You can either say:

```
self.score_msg.set_text("Score: " + str(self.score))
```

which will produce the message `Score: 13`, or you can say:

```
self.score_msg.set_text("Score: %04d" % self.score)
```

which produces `Score: 0013` instead.

Ask a leader if you want to know more about the `%` operator and how to use it.

Another type of object you may wish to use like this is a status message that goes away after a short while. The library provides the `Message` class to do exactly this; you can simply create the message, not bother to keep the resulting "handle" and let it disappear after a short while under it's own steam.

Things That Move

Things That Move tend to be much the same as Things That Stay Put, except that they have `games.Mover` as a subclass as well as whichever `Object` class you want. `Mover` does everything you need to move your object across the screen at whatever speed you specify, and also allows you to spin the object at whatever angular speed you want as well. This is handy for simulating the behaviour of real space ships, for example, but makes for a fiendishly difficult game!

One thing you must do if you use the `Mover` class is to write a `moved()` routine for it. `moved` is called after every movement of the object (even if that movement happens to keep the object in exactly the same place), allowing you to check to see whether your object has collided with anything.

Collision detection is most easily handled by the method used in Space War and Asteroids: define yourself a class called `Hittable` which has a single method called `check_for_hits`, which can be called from your `moved` routines. `check_for_hits` must then do the following:

- Get a list of objects that overlap this one, using the function `self.overlapping_objects()` provided by the library.
- For each object in the list:
  - Check if the object is `Hittable`. The Python function `isinstance(object, class)` returns true if the `object` belongs to `class` or a subclass of `class`.
  - If it is `Hittable`:
    * Run `self.hit(object)` to tell `self` it's been hit.
    * Run `object.hit(self)` to tell the object it's been hit.

You then simply add `Hittable` to every class that makes a difference when it is hit by something else, whether it's a class of moving objects or not. Each `Hittable` subclass must define a `hit` method to do whatever needs to be done when it's hit. For objects like planets that don't do anything themselves but cause other objects (like, say, spaceships) to blow up when they hit them, this is usually just:

```
def hit(self, what_hit_me):
  pass
```

Keyboard Handling

General games tend to need to know if a given key is pressed at a particular moment, for example for rotating a spaceship while the key is pressed. This is best done using the `Screen` class method `is_pressed`, which returns true if the key that it's given is currently pressed down. For instance

```
self.screen.is_pressed(games.K_a)
```

is a way for an object on the screen to ask if the A key is currently held down. You can refer to the keys from A to Z as `games.K_a`, `games.K_b` and so on up to `games.K_z`. Similarly 0 to 9 are `games.K_0` to `games.K_9`, the Return key is `games.K_RETURN`, the space bar is `games.K_SPACE`, and the arrow keys are `games.K_LEFT`, `games.K_RIGHT`, `games.K_UP` and `games.K_DOWN`. If you want to know what any of the other keys are, ask a leader.

Usually you will want to checked for pressed keys in the `moved` method of whatever object needs to be affected. If you have a two player game such as Space War, bear in mind that the two different player's objects (ships, say) will need to respond to two different sets of keys. You will need to find some way of telling which ship which keys it should respond to.

If one of your key presses has a global effect (perhaps some kind of smart bomb), it may be easiest to pick that up in your `Screen` subclass. While `Screen` has no `moved` method, it does have a `tick` method that is called after all the objects have been moved. The default `tick` method does nothing, so you can simply write your own version in your class.

An alternative, better solution if you simply want a one-shot action the moment a key is pressed is to use the `handle_keypress` method described in the section on board games.

Mouse Handling

There are two methods of the `Screen` class that are likely to be appropriate to help general games in using mouse control. Along the same lines as `is_pressed`, the function `self.mouse_buttons` returns the current state of the mouse buttons. This is returned as a tuple of three numbers, the state of the left, middle and right buttons respectively; a value of 1 indicates that the button is pressed, while 0 indicates that it isn't.

Similarly, `self.mouse_position` returns the current position of the mouse on the screen as an (x, y) tuple. None of the current games worksheets use the mouse in this way (like a joystick), so there are no easy examples to copy!

Different Types of Objects

This is a very brief overview of the different kinds of `Object` available from the library, and what you can do with them.

Sprite

A `Sprite` is a bitmap image, such as an asteroid. It can be placed on the screen, moved around, rotated and such all as normal. The only thing you can't do that you can do with other `Objects` is change its colour, obviously.

An `Animation` is a subclass of `Sprite` that consists of a sequence of bitmaps that replace one another on the screen, like the pictures in a flip book or the animated GIFs on many Web pages.

Do not try to make a moving `Animation`*,* only tears can come from such a thing. Both `Animation` and `Mover` are subclasses of `Timer`, and you will end up with either a moving image that doesn't change or a changing image that doesn't move. If you really must have a moving animated image, you will have to write your own class using the library's `Animation` and `Mover` classes as a guide.

Text

A `Text` object is exactly what it sounds like; text on the screen. It can be placed, rotated, moved and have its colour changed. You cannot change the font size after creating the `Text` object, but you can change the text itself. The only thing you need to be wary of is that the position on the screen that you specify for a `Text` object is the *centre* of the text string, not the left-hand side as you might be expecting.

A `Message` is a `Text` object that destroys itself, disappearing from the screen after a preset amount of time. It can be made to call another function when it disappears, making it a handy control for countdowns to explosions or the like.

Do not try to make a moving `Message`*,* no good can come of it. Both `Message` and `Mover` are subclasses of `Timer`, and you will end up with either a moving message that doesn't die off or a disappearing message that won't move. If you really must do such a thing, you'll have to write your own class using the library's `Message` and `Mover` classes as a guide.

Circle

A `Circle` is, as you'd expect, a simple circle on the screen. It can be filled in or not, and can have a different outline colour from its interior "fill colour." It can be placed, moved, rotated (though that has no visible effect, obviously), have its outline and fill colours changed and be made bigger or smaller at any point.

Polygon

A `Polygon` is every shape on the screen that isn't a `Circle`. You must supply a *path*, a list of points relative to the centre of the object that get joined together by straight lines to make up the object. Like a `Circle`, it can be filled in or not, and can have a different outline colour from its fill colour. It can be placed anywhere on the screen, moved, rotated, have its colours changed or have its path changed to something different as desired.

Other Useful Classes

Timer

This is a useful class for any purpose where you want something to change over time, either repeatedly over a period of time or once off like an alarm. You simply need to supply a `tick` method in whatever subclass you make that will be called when the timer "goes off."

As noted above, don't try to mix a `Timer` with an `Animation`, a `Message` or a `Mover`. All three of them already subclass `Timer`, and it won't work properly. You can easily get the same effect in a `Mover` subclass by doing your work in the `moved` method you have to write anyway, since that is called once every frame.

Mover

As noted above, this is the class you want to add in to get objects moving across the screen. All you have to do is write a `moved` method in whatever subclass you make. This method will be called after the object has been moved every frame, even if the object was moved no distance at all!

Don't try to mix it with a `Timer`, an `Animation` or a `Message`, it won't work properly. You can often achieve the effects that you want by adding some extra code to your `moved` method.

Board Games

Board games make use of the same classes as general games, so you should read the descriptions of those above. The various subclasses of `Object` are the most common, so you should particularly read up on `Circle`s and `Polygon`s. `Mover` subclasses are extremely rare, but not without their ingenious uses if you are feeling creative!

However, board games tend to have a number of common features, so the `boards` library is provided to make life a little easier. In particular, the `SingleBoard` and `GameCell` classes give you a skeleton on which you can build most types of board games quite easily.

SingleBoard

Like general games, board games also require a window to take place in. In order to provide a number of useful facilities, the `boards` library supplies a subclass of `Screen` called `SingleBoard` that is the basis of most board games. You will almost always want to make your own subclass with your own specific features, just as with the `Screen` class.

`SingleBoard` provides everything that `Screen` does. It also expects to contain an $n \times m$ grid of `GameCell`s, another class that the library provides, and will create this grid when it is initialised. If you want the grid to consist of something other than `GameCell`s (and you almost always want a subclass of `GameCell` rather than the class itself), you will need to override the `new_gamecell` method to return something else. For example:

```
def new_gamecell(self, i, j):
  return MyGameCell(self, i, j)
```

You can make holes in the board, places where cells don't get created, by overriding the method `on_board`. The default version returns 0 if the row or column specified is too large or small for the board size, and your version must do this also. However you can also return 0 in other places to prevent cells being created there. For example, to create a Monopoly-style track around the edge of the board you would write the following:

```
def on_board(self, i, j):
  return i == 0 or j == 0 or i == self._n_cols-1 or j == self._n_rows-1
```

Individual cells can be accessed as `self.grid[i][j]` whenever needed. If you need to perform an action on all cells, the method `map_grid` can help avoid writing tedious loops and checking whether cells really do exist or not. This method is given another function, and calls it on each cell in turn. So the line

```
self.map_grid(do_something)
```

translates to

```
do_something(self.grid[0][0])
do_something(self.grid[0][1])
do_something(self.grid[0][2])
```

and so on.

GameCell

A `GameCell` is a `Polygon` shaped to be a square of the appropriate size that makes up part of the board. It can also provide two other useful things; a list of which other `GameCells` are in which directions from this cell, and a list of all the `GameCells` that are neighbours of the cell regardless of direction. The two lists are useful in different circumstances, so you have to explicitly request them to be set up when you initialise your `GameBoard` by calling `self.create_directions` or `self.create_neighbours` respectively.

The `direction` array is ordered; `self.direction[boards.LEFT]` is the cell to the left of `self`, or None if there is no such cell. This is what you are most likely to want, since it is useful for sorting out where pieces can move on the board. If you alter the arrays yourself, you can create "wormholes" on the board where moving left off one square (say) unexpectedly puts you elsewhere.

The `neighbours` list is unordered; it contains all the cells that touch the current cell, but it tells you nothing about where exactly they are. It is most useful when you need to know how many neighbours a cell has, such as in Minesweeper. Again, you can fiddle with these lists to make unexpected cells neighbour one another, but this is less likely to be useful.

The library supplies a bunch of constants called `boards.LEFT`, `boards.UP`, `boards.UP_LEFT` and so on for use with `direction`, and a selection of utility functions for rotating from one direction to another. For a full list of these, see Sheet W.

For efficiency reasons, a `GameCell` is a *static* `Polygon`, which means that it isn't redrawn unless the library absolutely has to. Since the library is somewhat too strict in its definition of "absolutely has to" you will usually need to make your subclass of `GameCell` also a subclass of `Container` to get it to redraw properly. This also helps to keep any `Objects` that are put onto a cell visible despite the behind-the-scenes work that is done to maintain the cursor, of which more below.

Container

This is a utility subclass designed to do most of the work involved in keeping together `Objects` which are intended to be together. One `Object` is also subclassed as a `Container`, and has a number of elements which will be the other `Objects` which are intended stay with it. The `Container` is initialised with an ordered list of those elements, after which they will keep their relative positions to the `Container` when it is moved or raised. They will also all be removed from the screen when the `Container`'s `destroy` method is called.

Because `Container` overrides several of the normal methods of the `Object` class, it must appear first in the brackets when you make a subclass.

For example, suppose that a counter is to be represented on the screen as a white circle with a smaller red circle in the middle. The class declaration for this is:

```
class Counter(boards.Container, games.Circle):
  def __init__(self, board, x, y):
    self.init_container(['blob'])
    self.init_circle(board, x, y, 10, colour.white)
    self.blob = Circle(board, x, y, 5, colour.red)
```

After that, supposing that `counter` is an instance of our `Counter`, any call of `counter.move_to()` will move both the white circle and the red circle, leaving the red circle on top. Similarly `counter.raise_object()` will raise both circles to the front of the screen, still leaving the red circle on top.

This class is particularly useful in combination with the `GameCell` class above to get around a problem caused by the cursor. Using the cursor (see below) means changing the colour of the outline of a particular `GameCell`. In order to be sure that all sides of the cell are seen, the library raises the cell that the cursor is on to the top of the stack. If there's a counter in the cell, at this point it will disappear underneath the cell, which is hardly the idea. If however the cell is a `Container` which is declare to contain the counter, the counter will be brought to the top as well, and will stay visible, all without any additional effort on the part of the programmer.

It might be possible to use the `Container` class in a general game to keep a set of objects like a complex spaceship together, but this is untested. Note also that as yet, `Container` doesn't do anything about rotations, though that wouldn't be hard to change.

Key Handling

Board games tend to need single keypresses to take single actions, rather than the continuous control of most shoot-'em-ups. For this reason, you will usually want to write a `handle_keypress` method in your `SingleBoard` subclass to control what happens when keys are pressed.

`handle_keypress` is called whenever a key is pressed that the library doesn't understand as an instruction to move the cursor (q.v.). This is true even if you haven't switched on the cursor! The function is passed the key code of the offending key as a parameter, using exactly the same codes as for `is_pressed` (see above). It is then up to the programmer to decide exactly what to do with the keypress, if anything.

By default, the library understands the arrow keys to move the cursor left, right, up and down. This behaviour can be changed by altering the dictionary `self.key_movements`, which translates a key code into a tuple of two values, the change in column number and the change in row number. The default definition is this:

```
self.key_movements = {
  boards.K_UP:    (0, -1),
  boards.K_DOWN:  (0, 1),
  boards.K_LEFT:  (-1, 0),
  boards.K_RIGHT: (1, 0)
}
```

You can replace this with your own definition in your `__init__` routine, or simply add more keys if (for instance) you want to use the numeric keypad to add diagonal directions.

If you do not wish to use the cursor at all, it's a better idea to call your keypress handling method `keypress` instead; you get exactly the same parameters, but `key_movements` is completely ignored and no automatic movement of the cursor will be done.

Cursor Movement

A useful facility provided by the `SingleBoard` class is the cursor. This is a cell highlighted on the screen by changing the colour of its edges (by default to red). The highlighting can be moved around the screen using the call `move_cursor`; the arrow keys by default will do this, as described above. This gives a notion of the "currently selected cell on the board" which is needed for much keyboard handling; the `SingleBoard` element `cursor` holds the `GameCell` currently highlighted for this purpose. Thus a common sight in keypress handling methods is code like this:

```
if key == FIRST_KEY:
  self.cursor.do_first_thing()
elif key == SECOND_KEY:
  self.cursor.do_second_thing()
```

If you wish to use the cursor, the highlighting is turned on by calling `enable_cursor()` (usually in an `__init__` routine) and turned off by `disable_cursor()`. It's considered polite to turn the cursor off when the game is over.

If you need to know that the cursor has moved, the library calls the `SingleBoard` method `cursor_moved` after changing the position of the cursor, or after enabling or disabling the cursor. This allows you to handle for yourself exactly how the cursor appears on the screen, or whatever other processing you need. Since this feature hasn't been needed to date, it's possible that it doesn't entirely fit the bill — if you need to do something and it doesn't appear to help, tell a leader and see if something can be done.

Mouse Handling

Along the same lines as the keypress handling, the library will call two methods of the `Screen` or `SingleBoard` class when mouse clicks happen. You should override the default action of doing nothing by writing your own version of these functions if you want to take note of the mouse.

When a mouse button is pressed, the library calls `mouse_down`, giving it the position of the pointer when the button was pressed as an (x, y) tuple and a number representing the button which has been pressed. The left mouse button is represented by 0, the middle button (if present) by 1, and the right button by 2. Similarly, when a mouse button is released the library calls `mouse_up`, giving it the same parameters as for `mouse_down`.

For the purposes of most board games you will want to convert the (x, y) screen coordinates into a row and column number, in order to work out which `GameCell` has been clicked on. The library provides a conversion function `coords_to_cell` in the `SingleBoard` class to do just that, so a mouse handler commonly looks something like this:

```
def mouse_up(self, (x, y), button):
  (i, j) = self.coords_to_cell(x, y)
  if self.on_board(i, j):
    self.grid[i][j].do_something()
```

It's usually considered kind to take action on `mouse_up` events rather than `mouse_down`. This allows a player to change his or her mind at the last minute by moving the pointer somewhere safe before letting go.

Final Notes

This sheet should have given you an overview of how the LiveWires games libraries hang together, so that you can go on to develop your own games using them. It doesn't supply a detailed description of each method supplied by each class, just comments on how to use them to create games. You will also need to read the full descriptions in Sheet W to find out the syntax of the functions to find out how to use them properly, but this sheet will help you work out which functions you want to use.

The LiveWires libraries are in a continuous state of development. If you come across a feature that you think should be added to the library to make your life easier when writing a particular game, please do let us know. We may not be able to provide it straight away, but it's always better to know what people want before starting work on something completely different.

Good luck in writing your own games!