Cocos Programming Guide for v0.2.x

For a list of changes since last version, check Changelog

Introduction

"Cocos" is a framework for building games, demos, and other graphical/interactive applications. It is built over pyglet. It provides some conventions and classes to help you structure a "scene based application".

A cocos application consists of several scenes, and a workflow connecting the different scenes. It provides you a "director" (a singleton) which handles that workflow between scenes. Each scene is composed of an arbitrary number of layers; layers take care about drawing to the screen (using the pyglet and OpenGL APIs), handle events and in general contain all of the game/application logic.

Cocos simplifies the game development in these areas:

- Defining a workflow for your game
- Composing scenes and scene components
- Creating special effects & transitions in and between scenes
- Managing sprites
- Basic menus
- and more

Requirements

- Python 2.4 or later
- pyglet 1.1 or later. Cocos 0.2.x does not work with pyglet 1.0

Install

```
cp -r cocos $PROJECT_HOME
```

or

```
export PYTHONPATH=$PYTHONPATH:/path/to/cocos
```

or

```
import sys
sys.path.insert(0, PATH_TO_COCOS))
```

Contact us

Website: http://code.google.com/p/los-cocos/

If you find any bug, please report it at: http://code.google.com/p/los-cocos/issues/list

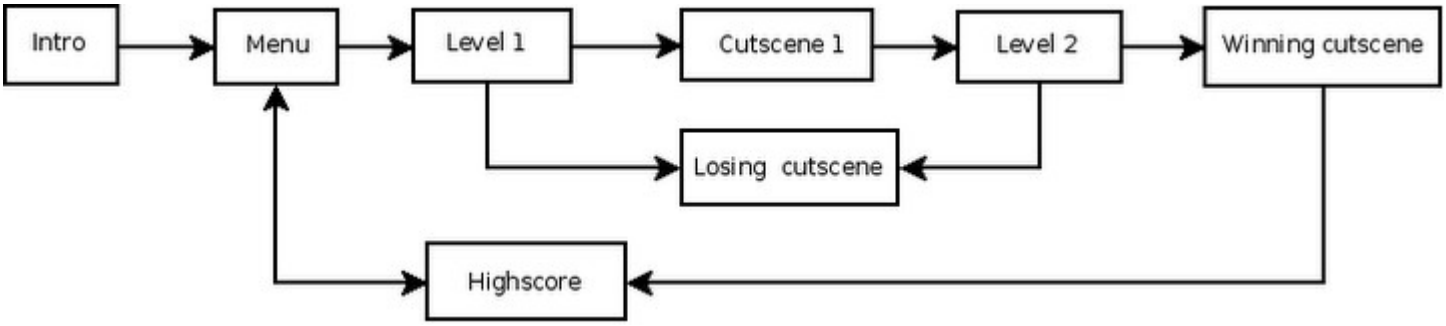
Basic concepts

There are some basic concepts introduced in this library that you will need to know when developing a cocos application:

Scenes & Transitions

An scene is a more or less an independent piece of the app workflow. Some people may call them "screens" or "stages". Your app can have many scenes, but only one of them is active at a given time.

For example, you could have a game with the following scenes: Intro, Menu, Level 1, Cutscene 1, Level 2, Winning cutscene, losing cutscene, High scores screen. You can define every of one of these scenes more or less as separate apps; there is a bit of glue between them containing the logic for connecting scenes (the intro goes to the menu when interrupted or finishing, Level 1 can lead you to the cutscene 1 if finished or to the losing cutscene if you lose, etc.).



A cocos Scene is composed of one or more layers, all of them piled up. Layers give the scene an appearance and behavior; the normal use case is to just make instances of Scene() with the layers that you want.

There is also a family of Scene classes called "Transitions", which allow you to make transitions between two scenes (fade out/in, slide from a side, etc).

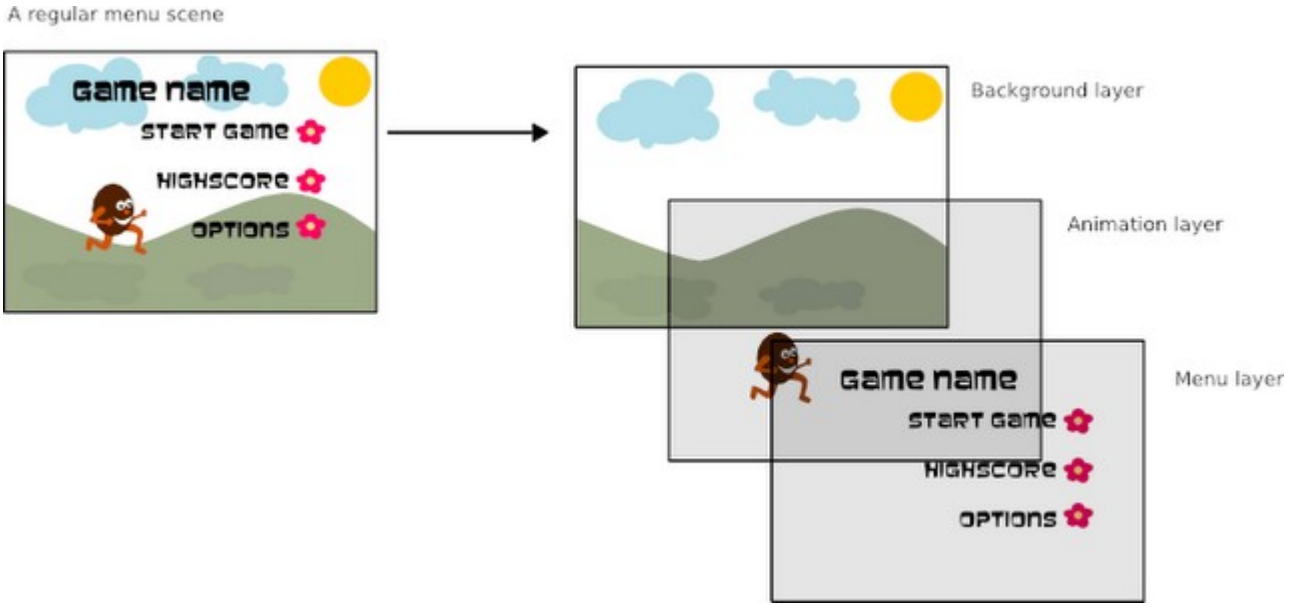
Director

The director is the component which takes care about going back and forth between scenes.

The director is a shared (singleton) object. It knows which scene is currently active, and it handles a stack of scenes to allow things like "scene calls" (pausing a scene and putting it on hold while other enters, and then returning to the original). Is the one who will actually change the scene, after a layer has asked for push, replacement or end of the current scene.

Layers & Effects

A layer has as size the whole drawable area (window or screen), and knows how to draw itself. It can be semi transparent (having holes and/or partial transparency in some/all places), allowing to see other layers behind it. Layers are the ones defining appearance and behavior, so most of your programming time will be spent coding Layer subclasses that do what you need.



The layer is where you define event handlers. Events are propagated to layers (from front to back) until some layer catches the event and accepts it.

Even if any serious app will require you to define some layer classes, cocos provides a library of useful predefined layer (a simple menu, solid color, a multiplexor between other layers, and a layer to add animated sprites/particle effects)

Besides that, you can apply an "effect" to a layer; effects transform the appearance of the whole layer. There are some classes to allow you to more or less easily define new effects. Cocos bring a couple of ready-to-use effects (colorization i.e. changing color or transparency of a layer, and repositioning i.e. moving or resizing a layers). Effects can be dynamically enabled/disabled/reconfigured; this allow to redefine the given effects to create effects as "flickering" or "shaking", and apply these effects to your layers.

Actions & Sprites

A Cocos' sprite is like any other computer sprite. It is a 2D image that can be moved, rotated, scaled, etc.

These sprites supports actions. An Action is "something" that the sprite can do, and an sprite can do multiple actions at the same time. Also, 2 or more different sprites can run the same action at the same time. There are different kind of actions. There are 3 actions categories:

- Actions that lets you transform the sprite's properties (position, rotation, transparency...)
- Actions that combines other actions
- Actions that do something else

There is another way to categorize the actions:

- Interval Actions
- Non-Interval Actions

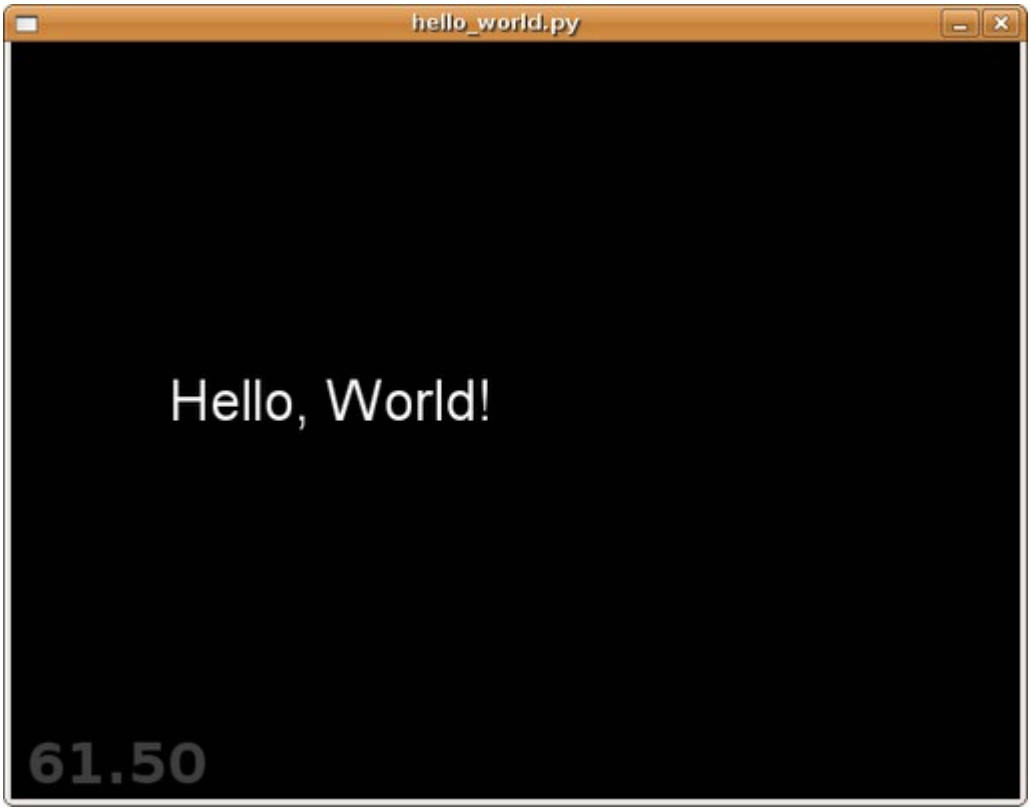
The *Interval Actions* are the ones that take place in a certain period of time and they have certain properties like:

- They can run forward in time
- They can run backwards in time
- They can be accelerated
- The flow of time can be transformed using a custom function

Details

A First Program

A very very simple program could look like this:



To do so, follow the following steps:

1. Define a layer which shows something (For example, a "Hello world" string).
2. To define a new layer class you inherit `cocos.layer.Layer`. In the `draw' method you should define the code that paints the layer on screen (using pyglet or opengl functions):

```
class HelloWorld(cocos.layer.Layer):
    def __init__(self):
        super(HelloWorld,self).__init__()
        # see pyglet documentation for help on this lines
        ft = font.load('Arial', 36)
        self.text = font.Text(ft, 'Hello, World!', x=100, y=240)

    def draw(self):
        # this funcition is called every frame
        self.text.draw()
```

3. Start the director. This initializes the window/display, and sets up cocos

```
# director init takes the same arguments as pyglet.window
director.init()
```

4. Create an instance of the layer defined above...

```
# We create a new layer, an instance of HelloWorld
hello_layer = HelloWorld ()
```

5. ...and a scene, with just that layer inside:

```
# A scene that contains the layer hello_layer
main_scene = cocos.scene.Scene (hello_layer)
```

6. Finally, tell the director to run with that scene

```
# And now, start the application, starting with main_scene
director.run (main_scene)
```

Of course, the last three steps sometimes can be shortened to just `director.run( cocos.scene.Scene( HelloWorld() ) )`; our hello world app is a bit verbose to make clear which are the steps involved.

This example is complete at `samples/hello_world.py`, and you can run it and play with it:

hello_world.py

```
import cocos
from cocos.director import director
from pyglet import font
class HelloWorld(cocos.layer.Layer):
    def __init__(self):
        super(HelloWorld,self).__init__()
        # see pyglet documentation for help on this lines
        ft = font.load('Arial', 36)
        self.text = font.Text(ft, 'Hello, World!', x=100, y=240)

    def draw(self)
        # this function is called on every frame
        self.text.draw()
if __name__ == "__main__":
    # director init takes the same arguments as pyglet.window
    director.init()
    # We create a new layer, an instance of HelloWorld
    hello_layer = HelloWorld ()
    # A scene that contains the layer hello_layer
    main_scene = cocos.scene.Scene (hello_layer)
    # And now, start the application, starting with main_scene
    director.run (main_scene)
    # or you could have written, without so many comments:
    #      director.run( cocos.scene.Scene( HelloWorld() ) )
```

Try for example changing the window parameters (arguments passed to `director.init`) to change the window size or make a full screen app. Or try modifying the step method to display something else.

Director, Scene, Layer

This section shows how to make more complex apps, with more layers and scenes.

Multiple Layers

In this demo, we will make a scene with several layers. We will create several instances of a single layer class for brevity, although of course you could use instances of different layer classes.

We start the example defining a layer class which shows squares. Its constructors has arguments to change the square position, color and size:

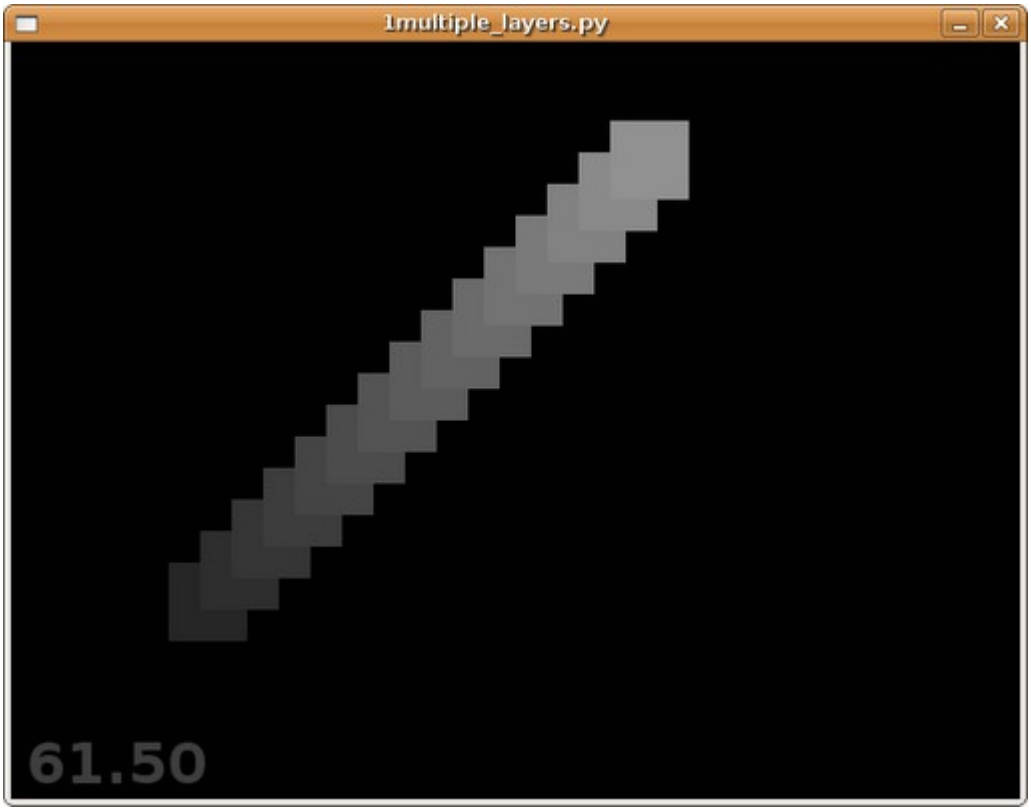
```
class Square(cocos.layer.Layer):
    """Square (color, c, y, size=50) : A layer drawing a square at (x,y) of
    given color and size"""
    def __init__(self, color, x, y, size=50):
        super(Square,self).__init__()
        self.x = x
        self.y = y
        self.size = size
        self.color = color

    def draw(self):
        gl.glColor4f(*self.color)
        x, y = self.x, self.y
        w = x+self.size; h=y+self.size
        gl.glBegin(gl.GL_QUADS)
        gl.glVertex2f( x, y )
        gl.glVertex2f( x, h )
        gl.glVertex2f( w, h )
        gl.glVertex2f( w, y )
        gl.glEnd()
        gl.glColor4f(1,1,1,1)
```

This is a good time to note a few things about the coordinate system used by cocos. Cocos has a "virtual" screen size, that is initially identical to the "physical" screen size (by screen I mean the window app or the fullscreen). The physical screen size might change if you resize the window (which is only allowed if you pass `resizable=True` to `director.init()`) or if you switch to fullscreen mode (by calling `director.window.set_fullscreen(True)`). However, even if the physical screen size changes, the virtual screen size stays the same as it was on the moment of initializing the director. All the drawing commands use "virtual" coordinates, so when defining `step` methods you normally don't need to care if your app was resized or changed to fullscreen.

Making a scene with more than one layer is very easy: Just use multiple parameters when creating the `Scene` instance: `Scene (a_layer, another_layer, a_third layer)`. In our demo, we will create a large number of layers and put them into a list:

```
    layers = [ Square((0.03*i,0.03*i,0.03*i,1) , i*20, i*20) for i in range(5,20) ]
```

This list contains layers showing 15 overlapping squares, going from the bottom-left of your screen to the top-right and going from dark gray to light gray.



Now, let's create a scene with all those layers inside:

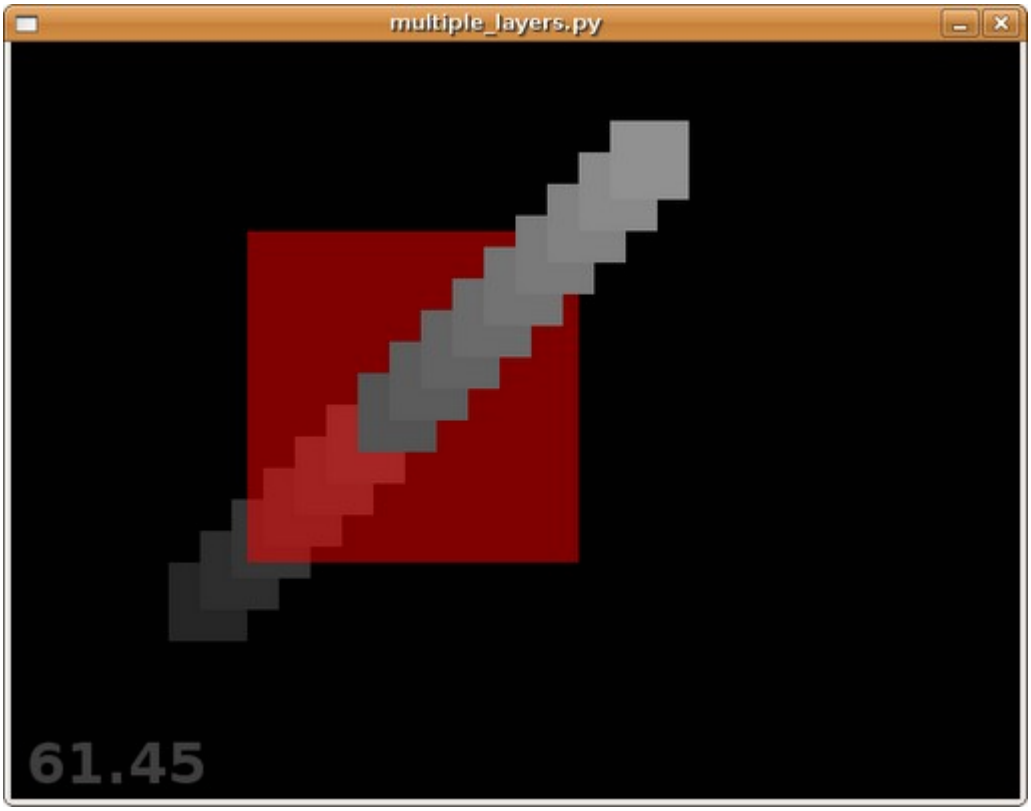
```
sc = cocos.scene.Scene(*layers)
```

Note that the layers are passed from bottom to top. That means that the last layer of the list in our example (the light gray top-right square) will be in front of every other layer. Every layer has a "z-value", a numeric value which is higher for "upper" layers. When you create a Scene, each layer is added with a "z-value" equal to its position in the argument list (0 to 14 in the example above).

Sometimes you will need to add or remove layers from a scene after its creation. You can do that with the `add` and `remove` methods of the scene. For example, you can call:

```
sc.add( 5.5, Square((1,0,0,0.5), 150,150, 210 ), "big_one" )
```

This adds a large, red, semitransparent square more or less at the middle of the screen. The first argument, "5.5" is the z-value of the new layer. 5.5 means that it will be over the first 6 small squares (the ones with z-value 0 to 5) and below the other 9 (the ones with zvalues 6 to 14).



The last argument "big_one" is optional; it is a name you can give to the layer, to reference it later if you want to remove it. For example, if you later want to remove the big red square you can do it like this:

```
    sc.remove ("big_one")
```

This example is complete at `samples/multiple_layers.py`, and you can run it and play with it:

```
import cocos
from cocos.director import director
from pyglet import gl
# Defining a new layer type...
class Square(cocos.layer.Layer):
    """Square (color, c, y, size=50) : A layer drawing a square at (x,y) of
    given color and size"""
    def __init__(self, color, x, y, size=50):
        super(Square,self).__init__()
        self.x = x
        self.y = y
        self.size = size
        self.color = color

    def draw(self):
        gl.glColor4f(*self.color)
        x, y = self.x, self.y
        w = x+self.size; h=y+self.size
        gl.glBegin(gl.GL_QUADS)
        gl.glVertex2f( x, y )
        gl.glVertex2f( x, h )
```

```
        gl.glVertex2f( w, h )
        gl.glVertex2f( w, y )
        gl.glEnd()
        gl.glColor4f(1,1,1,1)

if __name__ == "__main__":
    director.init()
    # Create a large number of layers
    layers = [ Square((0.03*i,0.03*i,0.03*i,1) , i*20, i*20) for i in range(5,20) ]
    # Create a scene with all those layers
    sc = cocos.scene.Scene(*layers)
    # You can also add layers to a scene later:
    sc.add( 5.5, Square((1,0,0,0.5), 150,150, 210 ), "big_one" )
    director.run( sc )
```

Things you can try to do with this code:

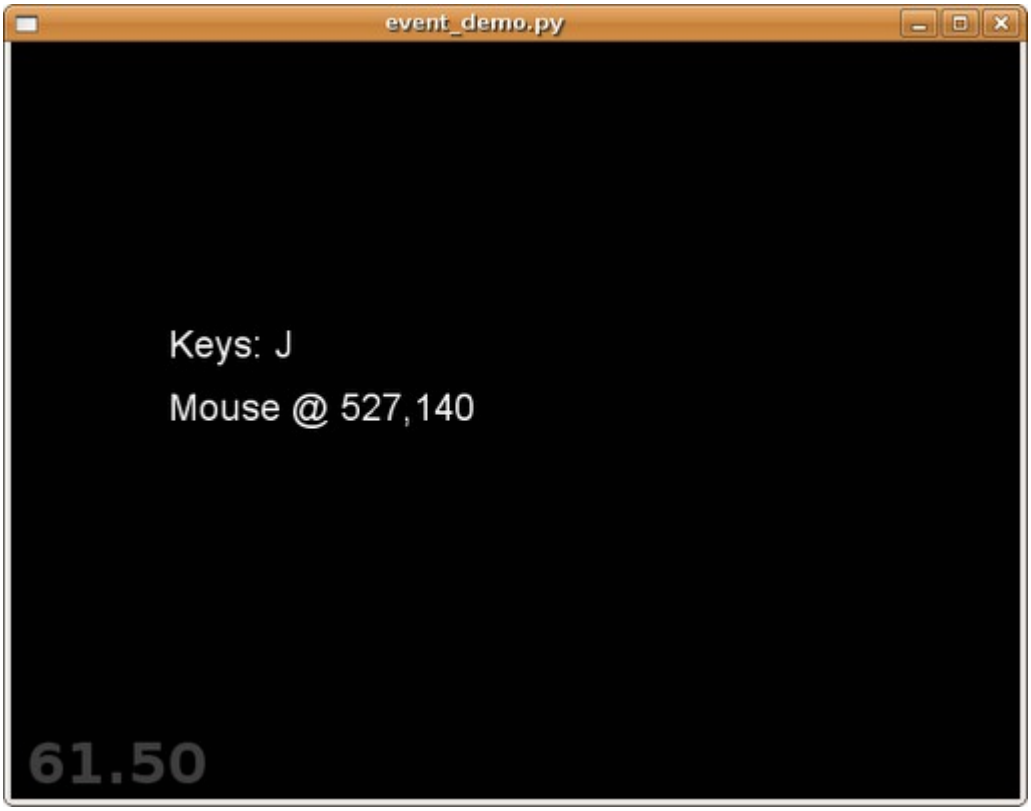
- add an instance of `ColorLayer` (check documentation) to put a solid color background.
- rearrange the layer order,
- create the scene initially empty (`sc = cocos.scene.Scene()`), add all layers manually (with `sc.add` in a loop), and remove a few by name.

Events

All our previous examples are non-interactive. They display something, but do not respond to user input (except for quitting when you press ESC or close the window).

All layers are event handlers from the pyglet event framework. When a layer is being shown, its event handlers will be active. You can read more about pyglet events at http://www.pyglet.org/doc/programming_guide/the_pyglet_event_framework.html

In this section we will build step by step the demo provided in `samples/event_demo.py`; this is a very simple cocos app which shows which keys are pressed, and reacts to mouse motion and clicks. Run the app before reading on te get a clearer idea of what we are trying to build.



This demo has a scene with two layers; one shows which keys are currently pressed (none, one, or maybe several at the same time), the other one shows text with the mouse position, and clicking moves the text.

We start defining the `KeyDisplay` layer class. As always, we put some initialization on `__init__` and the code for displaying it in `step`:

```
class KeyDisplay(cocos.layer.Layer):
    def __init__(self):
        super( KeyDisplay, self ).__init__()
        self.text = pyglet.text.Label("", x=100, y=280, batch=self.batch)
        # To keep track of which keys are pressed:
        self.keys_pressed = set()
        self.update_text()

    def update_text(self):
        key_names = [pyglet.window.key.symbol_string (k) for k in self.keys_pressed]
        text = 'Keys: '+','.join (key_names)
        # Update self.text
        self.text.text = text
```

This class defines a `key_pressed` set, which should be the set of keys pressed at any time. However, this code as it is still does nothing. We need to tell this layer to update this set when a key is pressed or released. In other words, we need to add event handlers to the layer. Adding event handlers to a layer is just a matter of adding methods to it called `on_<event name>`. The two events that interest us now are `on_key_press` and `on_key_release`:

```
    def on_key_press (self, key, modifiers):
        """This function is called when a key is pressed.

        'key' is a constant indicating which key was pressed.
        'modifiers' is a bitwise or of several constants indicating which
           modifiers are active at the time of the press (ctrl, shift, capslock,
etc.)

        """
        self.keys_pressed.add (key)
        self.update_text()
    def on_key_release (self, key, modifiers):
        """This function is called when a key is released.

        'key' is a constant indicating which key was pressed.
        'modifiers' is a bitwise or of several constants indicating which
           modifiers are active at the time of the press (ctrl, shift, capslock,
etc.)
        Constants are the ones from pyglet.window.key
        """
        self.keys_pressed.remove (key)
        self.update_text()
```

With that code, the layer is now fully working. You can press and release keys or key combinations, and you will se how the display is updated telling you which keys are pressed at any time.

Handling mouse input is similar. You have three events of interest: `on_mouse_press`, `on_mouse_release` and `on_mouse_motion`. With that, we can define our layer:

```
class MouseDisplay(cocos.layer.Layer):
    def __init__(self):
        super( MouseDisplay, self ).__init__()
        self.x = 100
        self.y = 240
        self.text = pyglet.text.Label('No mouse events yet', font_size=18,
x=self.x, y=self.y, batch=self.batch)

    def update_text (self, x, y):
        text = 'Mouse @ %d,%d' % (x, y)
        self.text.text = text
        self.text.x = self.x
        self.text.y = self.y
```

And then add event handlers to update the text when the mouse is moved, and change the text position when any button is clicked.

```
    def on_mouse_motion (self, x, y, dx, dy):
        """This function is called when the mouse is moved over the app.

        (x, y) are the physical coordinates of the mouse
        (dx, dy) is the distance vector covered by the mouse pointer since the
          last call.
        """
        self.update_text (x, y)
```

```
    def on_mouse_press (self, x, y, buttons, modifiers):
        """This function is called when any mouse button is pressed
        (x, y) are the physical coordinates of the mouse
        'buttons' is a bitwise or of pyglet.window.mouse constants LEFT, MIDDLE,
RIGHT
        'modifiers' is a bitwise or of pyglet.window.key modifier constants
           (values like 'SHIFT', 'OPTION', 'ALT')
        """
        self.x, self.y = director.get_virtual_coordinates (x, y)
        self.update_text (x,y)
```

The only thing a bit unusual here is the call to `director.get_virtual_coordinates (x, y)`. As explained in the example before, cocos has two coordinates systems, a physical one and a virtual one. The mouse event handlers receive their arguments from pyglet in physical coordinates. If you want to map that to virtual coordinates, you need to use the `director.get_virtual_coordinates` method, which does the correct mapping. If you put instead `self.x, self.y = x,y` in the `on_mouse_press` handler above, you will see that the app seems to work, but if you resize the window, the clicks will move the text to the wrong place.

The demo does not have much more code, just creating a scene with these two layers and running it:

```
director.init(resizable=True)
# Run a scene with our event displayers:
director.run( cocos.scene.Scene( KeyDisplay(), MouseDisplay() ) )
```

You can now play to the demo and change it. Some things you can try are:

- Change the `on_mouse_press` handler and remove the mapping to virtual coordinates; note how it behaves strangely after resizing the window
- Note that the mouse coordinates on screen are physical coordinates, so their range changes when resizing the window; modify the demo to show virtual coordinates.
- Change the code to be able to move the mouse coordinates label when you drag the mouse (hint: a mouse drag is a sequence of button_press, several motions, and a button_release event)
- Change the code so the keyboard display also shows the modifiers set at each time

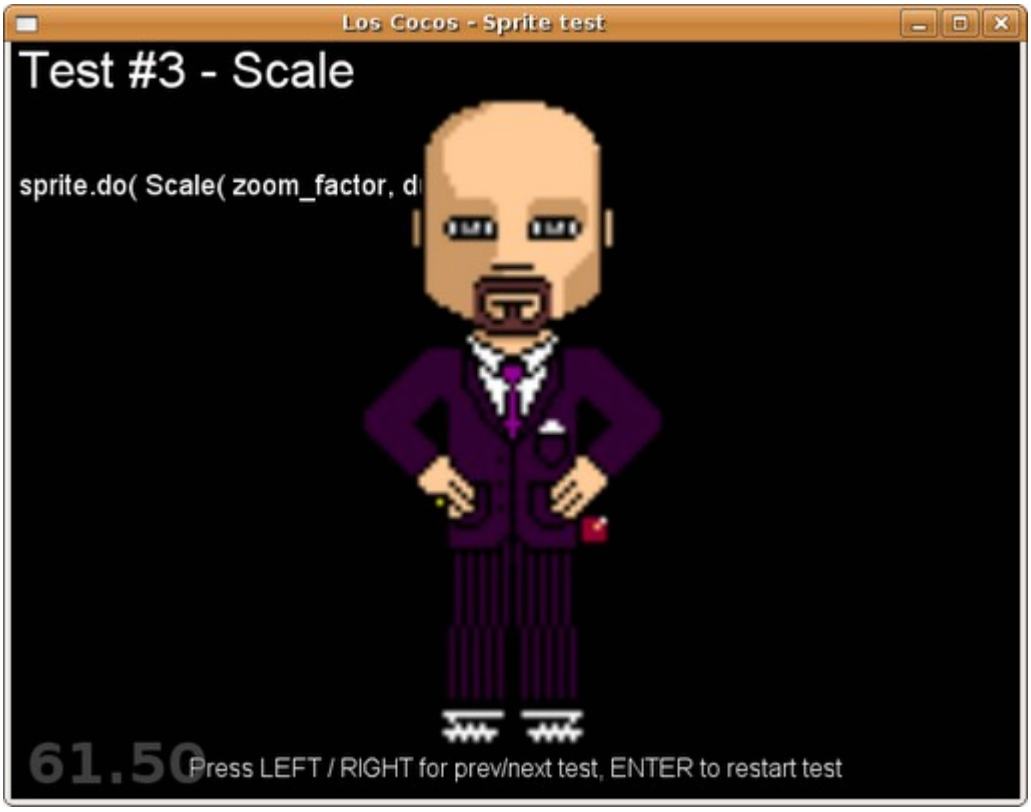
If you want more details about these and other events, and the arguments they get, check the pyglet documentation.

Scenes Stack

Note that you can add a single layer object to multiple scenes, allowing sharing (this can be useful if you want to reuse a layer, specially if you need it to keep state between scenes).

Sprites

Actions



You can find these examples here: *los-cocos/test/test_sprite.py* Run the file in order to see them working.

Transformation actions

The `duration` parameter is specified in seconds. The `x,y` parameters are specified in pixels. The (0,0) point is the bottom-left point.

- `Move( (delta_x,delta_y), duration)`
  - Moves the sprite `delta_x` and `delta_y` pixels in `duration` time.
- `Goto( (x,y), duration )`
  - Moves the sprite to the `x,y` coordinates.
  - The difference between `Move()` and `Goto()` is that `Move()` is relative to the current coordinates and `Goto()` is absolute.
- `Rotate( degrees, duration )`
  - Rotates the sprites degrees in `duration` time. Positives degrees rotates the sprite clockwise.
  - The rotation is relative to the current angle.

- `Scale( zoom_factor, duration )`
  - Scales the sprites `zoom_factor` in `duration` time.
  - The scale is relative to the current zoom_factor.
- `Jump( height, x, number_of_jumps, duration )`
  - Moves the sprites `x` pixels in `duration` time. `x` is relative to the current x position.
  - During that movement the sprite will do a `number_of_jumps` of `height`.
- `Bezier( bezier_configuration, duration )`
  - Moves the sprite using a bezier path in `duration` time. The movement is relative to the current position.
- `Place( (x,y) )`
  - Places the sprite in the `x,y` coordinates.
- `FadeIn( duration )`
  - Fades the sprite in in `duration` time.
- `FadeOut( duration )`
  - Fades the sprite out in `duration` time.
- `Blink( times_to_blink, duration )`
  - Blinks the sprite `times_to_blink` in `duration` time.
- `Show()`
  - Shows the sprite.
- `Hide()`
  - Hides the sprite. To show it again, use the `Show` action.

Composite actions

- `Repeat( action, times, mode=PingPongMode )`
  - Repeats an action.
  - If `times` is -1, then it will repeat the action forever. Default is -1
  - If mode is `PingPongMode` (default mode) it will repeat the action forwards and backwards.
  - If mode is `RestartMode`, then it will always repeat the actions forwards.
- `Sequence( list_of_actions, dir=ForwardDir )` or `action1 + action2 + action3` ...
  - Runs a list of actions is sequential mode. First it runs the first action. When this action finished, it execute the next one until the last one is executed.
- `Spawn( list_of_actions )` or `action1 | action2 | action3` ...
  - Execute the list_of_actions at the same time

Misc Actions

- `CallFunc( function )`
  - Calls a function. Just that.

    - It is useful to when you want to trigger something.
- `CallFuncS( function )`
    - It is the same as `CallFunc` with the difference that `function` will receive the sprite as the 1st argument
- `Delay( seconds )`
    - It will delay the execution some *seconds*
- `RandomDelay( lo_seconds, hi_seconds )`
    - It will delay the execution random seconds between `lo_seconds` and `hi_seconds`

Interval Actions

An interval action is an action that takes place within a certain period of time. It has an start time and a finish time. The finish time is the parameter *duration* plus the start time. These *Interval Actions* have some interesting properties, like:

- They can run Forward (default)
- They can run Backwards
- They can be accelerated. You can transform the time using a custom function.

For example, if you run an action in a *Forward* direction and the you run it again in a *Backward* direction, then you are simulation a Ping-Pong movement.

These actions has 3 special parameters:

- *dir* (direction): It can be ForwardDir or BackwardDir . Default is: ForwardDir
- *mode* (repeat mode): It can be PingPongMode or RestartMode . Default is : PingPongMode .
- *time_func* (a function): A function that alters the speed of time

Examples of *Interval Actions*:

```
move = Move( (200,0), 5 ) # Moves 200 pixels to the right in 5 seconds.
                          # Direction: ForwardDir (default)
                          # RestartMode: PingPongMode (default)
                          # time_func: No alter function (default)
rmove = Repeat( move )    # Will repeat the action *move* forever
                          # The repetitions are in PingPongMode
                          # times: -1 (default)

move2 = Move( (200,0), 5, time_func=accelerate )
                          # Moves 200 pixels to the right in 5 seconds
                          # time_func=accelerate. This means that the
                          # speed is not linear. It will start to action
                          # very slowly, and it will increment the speed
                          # in each step. The total running time will be
                          # 5 seconds.
move3 = Move( (200,0), 5, dir=BackwardDir )
                          # Moves 200 pixels to the **left** in 5 seconds
                          # But when you use this direction (BackwardDir)
                          # the starting coords and the finishing coords
                          # are inverted
```

Implementing Actions

Let's see in detail how the *Goto* action is implemented:

```
class Goto( IntervalAction ):
```

It is a sublcass of `IntervalAction`, since this is an action that takes place within a certain period of time

```
    def init(self, dst_coords, duration=5):
        self.end_position = Point2( *dst_coords )
        self.duration = duration
```

In the `init()` method, you shall define the action's arguments. `Goto(coords,duration)` receives as arguments a tuple (the destination coordinates), and a duration time specified in seconds. The coordinates are transformed into a `Point2` structure to facilitate the arithmetics.

```
    def start( self ):
        self.start_position = self.target.position
        self.delta = self.end_position-self.start_position
```

The `start()` method is called before the action starts. The difference between `init()` and `start()` is that at the moment that `start()` is called `self.target` is initialized with the target sprite. So, in the `start()` method you shall initialize everything that depends on the target sprite.

```
    def step(self,dt):
        self.target.position = (self.start_position +
                    self.delta * (
                        max(0,min(1,float(self.get_runtime() )/self.duration))
                    ))
```

The calculation is done in the `step(dt)` method. `dt` is the delta time. `self.get_runtime()` returns how many seconds have elapsed since the start of action. When `self.get_runtime() >= self.duration` the action finished.

Transitions

Implementing New Transitions

Effects

Implementing New Effects

Packaged Layers

Basic Layers

`ColorLayer( *color )` creates a layer filled with color (RGBA)

`MultiplexLayer( *layers )` A Composite layer that only enables one layer at the time

Menus

Changes

from Cocos v0.1.x to Cocos v0.2.0

- `Director`: uses the new pyglet 1.1 loop
- `Director`: doesn't have the step() method. Uses on_draw() instead.
- `Director`: enable_alpha_blending() is not called automatically
- `Scene`: does not have the step() method. Uses on_draw() instead.
- `Layer`: step() is called only when it is enabled with Layer.enable_step()
- `Layer`: Removed AnimationLayer. No longer necessary.
- `Layer`: uses Batch.draw() to draw objects. Propagates the message draw() to the objects it contains.
- `ActionSprite`: is a subclass of pyglet.sprite.Sprite
- `ActionSprite`: `Move()` and `Goto()` uses (x,y) instead of (x,y,0)
- `ActionSprite`: removed `Animate` since it is part of pyglet.sprite.Sprite)
- `ActionSprite`: uses pyglet.clock.schedule() to generate a ticker.
- `ActionSprite`: renamed `RepeatMode` to `RestartMode`
- `ActionSprite`: `Rotate` rotates clockwise with positive degrees