Python Lecture 3 – week4

Contents

Game Design.

Now we have a rudimentary world collision system let's discuss what's next. Going back to our original design of Input, Output, Encapsulation, we have cover some input, out but no encapsulation. We can deal with say a title screen soon, as well as character sheet, but first of all let's get the character moving.

Once you finish this tutorial please spend the rest of your time this week working on your game design document.

Grab the source code we know works from here...run it to check and notice the player is smaller.

http://www.drewfx.com/TAFE/python/week4_source.zip

Movement Code

PyGame offers input for the mouse and keyboard. We have already seen it used for escaping the application here in Main.py...

```
    while gameRunning:

        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                gameRunning = False
            elif event.type == pygame.KEYDOWN:  #on key press
                if event.key == pygame.K_ESCAPE :
                    gameRunning = False
```

Let's extend this a little more and allow for the WASD keys to move the player around. We really need a player class to send movement message to, so let's make a new class.

Create a new python file called Player.py

Inside it create the following class specification...

```
''' Player class
'''

from GameScreenLayer import *

class Player():

    def __init__(self):
        ''' default constructor
        '''
        self.isMovingLeft = False
        self.isMovingRight = False
        self.isMovingUp = False
        self.isMovingDown = False
        self.xPos = 0
        self.yPos = 0
        self.speed = 2
        self.sprite = None
        self.spriteWidth = 0
        self.spriteHeight = 0
```

Notice firstly the `from GameScreenLayer import *`. We used that last time to create screen layers, and we need to now start using it's collision checking function.

We'll using the moving flags (isMovingLeft etc) to keep moving in a direction if the correct key is held down.

We'll need to get the size of the sprite as well. That's what will be store in spriteHeight and spriteWidth. So let's add that function.

```
#-----------------------------------------------------------------------
    def setSprite( self, sprite ):
        ''' used to test size of sprite
        '''
        self.sprite = sprite
        self.spriteWidth = sprite.get_width()
        self.spriteHeight = sprite.get_height()
```

Notice the dotted line above it. I've started adding these to split the code up a bit to make it easier to read. Pygame is providing the get_width and get_height functions. Sprite is really just a surface as stored in Main.py in the initialiseCollisionMap function.

As all good code writing goes, it's important to have get and set functions. For now we will need them for the position of the player so let's write these in.

```
#-----------------------------------------------------------------------
    def setPosition( self, position ):
        ''' sets the position of the sprite
        '''
        self.xPos = position[0]
        self.yPos = position[1]

#-----------------------------------------------------------------------
    def getPosition( self ):
        ''' return the position of the sprite
        '''
        return ( self.xPos, self.yPos )
```

Notice how I can pass parameters into and out of functions using the brackets. In the first function, setPosition, is a passed in parameter. It comes in as one 2 dimension tuple (x,y) and so to access each part I reference the [0]th and [1]st elements to get x and y out.

On sending the value back it needs to be sent back as a 2 dimensional tuple, so I surround the attributes xPos and yPos in round brackets.

The next one is quite complicated and deals with the collision code, so we will step through it a piece at a time. Let's handle the LEFT movement first.

```
#-----------------------------------------------------------------------
    def move(self, colMap, direction):
        ''' test for collision first
            move if able
        '''
        if direction == "Left":
            # test to the left
            if (colMap.isColliding( self.xPos - self.speed, self.yPos ) == False
                and
                colMap.isColliding( self.xPos - self.speed, self.yPos + self.spriteHeight ) == False):
                # nothing there so move it
                self.xPos -= self.speed
            else:
                # no longer able to move so cancel it
                self.isMovingLeft = False
```

This function takes in a pointer to colMap which was loaded last lesson. colMap has it's own functions, one of which we want is called isColliding, which takes a set of coordinates and converts them to map coordinates. If there is a collision cube (the red squares) there it return True. We want to stop the player from moving towards collision regions so we only let them move if there is not collision in the next step they take.

Notice also the two sets of isColliding settings. We need to check to the top and bottom of the sprite so none of the sprite can get in a collidable region.

Now it's time to add the Right code...

```
if direction == "Right":
    # test to the right
    if (colMap.isColliding( self.xPos + self.speed + self.spriteWidth, self.yPos ) == False
        and
        colMap.isColliding( self.xPos + self.speed + self.spriteWidth, self.yPos + self.spriteHeight ) == False):
        # nothing there so move it
        self.xPos += self.speed
    else:
        # no longer able to move so cancel it
        self.isMovingRight = False
```

This code check to the right of the player.

Now type in the Up code...

```
if direction == "Up":
    # test up
    if (colMap.isColliding( self.xPos, self.yPos - self.speed ) == False
        and
        colMap.isColliding( self.xPos + self.spriteWidth, self.yPos - self.speed) == False):
        # nothing there so move it
        self.yPos -= self.speed
    else:
        # no longer able to move so cancel it
        self.isMovingUp = False
```

And finally the Down code...

```
if direction == "Down":
    # test down
    if (colMap.isColliding( self.xPos, self.yPos + self.speed + self.spriteHeight ) == False
        and
        colMap.isColliding( self.xPos + self.spriteWidth, self.yPos + self.speed + self.spriteHeight ) == False):
        # nothing there so move it
        self.yPos += self.speed
    else:
        # no longer able to move so cancel it
        self.isMovingDown = False
```

Updating the Main file

We need to fire off these events from Main.py so save this file and open Main.py

At the top of Main.py we need to add a few lines to import the player class and assign one player class.

```
from Backpack import Backpack
from PaperDoll import PaperDoll
from ItemDatabase import ItemDatabase
from TileMap import TileFunctions
from GameScreenLayer import *
from Player import *
```

Directly after that we want to define all our globals for use in the rest of the program...

```
from Player import *

global collisionMap, playerPos, playerSprite, playerClass
global itemDatabase, playersPaperDoll, playersBackpack
global screen, screenList
```

Further down we will add the new player class. We also need to remember the collision map for use when we check where the player can move.

```
itemDatabase = ItemDatabase()
playersPaperDoll = PaperDoll()
playersBackpack = Backpack()
playerClass = Player()
collisionMap = None
```

Scroll down in to the initialiseCollisionMap function. At this line to allow for global variables to be used here.

```
def initialiseCollisionMap():
    # load up the collision map
    # create the collision map
    global collisionMap, playerPos, playerSprite, playerClass
```

Next find this spot to include a call to tell the player class where it should be placed in the world, and what the sprite is so we can use is size to test if the sprite has collided with objects.

```
    # get player pos because it has one
    # and set up for use in playerClass
    playerPos = __myMap.getPlayerStartPos( __map1 )
    playerClass.setPosition( playerPos )

    playerSprite = pygame.image.load("Player1.png")
    playerClass.setSprite( playerSprite )
```

Scroll further down to handlePlayer, add playerClass to the global list, and include a few lines above the existing code to allow movement of the player if we have set the movement flags such as isMovingLeft previously defined in the player class.

```
def handlePlayer():

    global playerPos, playerSprite, playerClass

    # test for moves
    if (playerClass.isMovingLeft):
        playerClass.move( collisionMap, "Left" )
    if (playerClass.isMovingRight):
        playerClass.move( collisionMap, "Right" )
    if (playerClass.isMovingUp):
        playerClass.move( collisionMap, "Up" )
    if (playerClass.isMovingDown):
        playerClass.move( collisionMap, "Down" )

    playerPos = playerClass.getPosition()
    screen.blit(playerSprite,playerPos)
```

Finally we can detect the key presses on DOWN and UP to set up the movement flags. We'll do the DOWN key presses first.

```
for event in pygame.event.get():
    if event.type == pygame.QUIT:
        gameRunning = False

    elif event.type == pygame.KEYDOWN:  #on key down

        if event.key == pygame.K_ESCAPE :
            gameRunning = False

        elif event.key == pygame.K_a :
            playerClass.isMovingLeft = True

        elif event.key == pygame.K_d :
            playerClass.isMovingRight = True

        elif event.key == pygame.K_w :
            playerClass.isMovingUp = True

        elif event.key == pygame.K_s :
            playerClass.isMovingDown = True
```

And finally the UP presses. We do this be we want the player to continue moving in a direction until the key is released.

```
        elif event.key == pygame.K_s :
            playerClass.isMovingDown = True

    elif event.type == pygame.KEYUP:  #on key up

        if event.key == pygame.K_a :
            playerClass.isMovingLeft = False

        elif event.key == pygame.K_d :
            playerClass.isMovingRight = False

        elif event.key == pygame.K_w :
            playerClass.isMovingUp = False

        elif event.key == pygame.K_s :
            playerClass.isMovingDown = False
```

Okay! Time to see the game running. Save your work and press the Play/Run button. You should be able to move around now and not run over the red squares using the WASD keys.

Now let's see it full screen. Press ESC and scroll up to the top of the Main.py file and change this line to true...

```
import pygame

pygame.init()


DEBUG_MODE = True
SCREENX = 640
SCREENY = 480
```

Save the file, hit Run and the screen will go full screen, and the collision map won't be visible. It's useful to have the choice to toggle between debug mode and release mode when we are building the game.

Adding your own art.

You'll notice some cases where you shouldn't be able to walk into a tree, or you can't get close enough. Now it's time for your mad art skilz!

Let's make our own collision maps, find a new spot for the player, and paint some new artwork.

It's also time to do some game design so write a one paragraph document on what you want your RPG to be, which will then dictate your art style.

Resources:

1) Python 2.5 (http://www.python.org/download/releases/2.5.5/)

2) PyGame for Python 2.5 (http://www.pygame.org/download.shtml)

3) Eclipse (http://www.eclipse.org/downloads/download.php?file=/eclipse/downloads/drops/R-3.5.2-201002111343/eclipse-SDK-3.5.2-win32.zip)

4) PyDev (http://sourceforge.net/projects/pydev/)
(just for my reference - don't download) Tutorials:
http://www.vogella.de/articles/Python/article.html#configuration

GLU-IT (http://www.downloadsofts.com/download/Graphic-Apps/Editors/Glue-Sprites-download-details.html)
AUDACITY (http://audacity.sourceforge.net/download/)

Tortoise SVN (http://downloads.sourceforge.net/tortoisesvn/TortoiseSVN-1.6.3.16613-win32-svn-1.6.3.msi?download)

Python 2.0 Quick Reference