<https://www.pygame.org/docs/tut/tom/MakeGames.html>

A common method of organising the code for a game is to divide it into the following six sections:

Load **modules** which are required in the game. Standard stuff, except that you should remember to import the Pygame local names as well as the Pygame module itself

**Resource handling classes**; define some classes to handle your most basic resources, which will be loading images and sounds, as well as connecting and disconnecting to and from networks, loading save game files, and any other resources you might have.

Game **object classes**; define the classes for your game object. In the pong example, these will be one for the player's bat (which you can initialise multiple times, one for each player in the game), and one for the ball (which can again have multiple instances). If you're going to have a nice in-game menu, it's also a good idea to make a menu class.

Any other game **functions**; define other necessary functions, such as scoreboards, menu handling, etc. Any code that you could put into the main game logic, but that would make understanding said logic harder, should be put into its own function. So as plotting a scoreboard isn't game logic, it should be moved into a function.

**Initialize the game**, including the Pygame objects themselves, the background, the game objects (initialising instances of the classes) and any other little bits of code you might want to add in.

The **main loop**, into which you put any input handling (i.e. watching for users hitting keys/mouse buttons), the code for updating the game objects, and finally for updating the screen.

Every game you make will have some or all of those sections, possibly with more of your own. For the purposes of this tutorial, I will write about how TomPong is laid out, and the ideas I write about can be transferred to almopst any kind of game you might make. I will also assume that you want to keep all of the code in a single file, but if you're making a reasonably large game, it's often a good idea to source certain sections into module files. Putting the game object classes into a file called "objects.py", for example, can help you keep game logic separate from game objects. If you have a lot of resource handling code, it can also be handy to put that into "resources.py". You can then "from objects,resources import \*" to import all of the classes and functions.

**2. Pygame fundamentals**

2.1. The basic Pygame game

For the sake of revision, and to ensure that you are familiar with the basic structure of a Pygame program, I'll briefly run through a basic Pygame program, which will display no more than a window with some text in it, that should, by the end, look something like this (though of course the window decoration will probably be different on your system):

The full code for this example looks like this:

import pygame

from pygame.locals import \*

def main():

# Initialise screen

pygame.init()

screen = pygame.display.set\_mode((150, 50))

pygame.display.set\_caption('Basic Pygame program')

# Fill background

background = pygame.Surface(screen.get\_size())

background = background.convert()

background.fill((250, 250, 250))

# Display some text

font = pygame.font.Font(None, 36)

text = font.render("Hello There", 1, (10, 10, 10))

textpos = text.get\_rect()

textpos.centerx = background.get\_rect().centerx

background.blit(text, textpos)

# Blit everything to the screen

screen.blit(background, (0, 0))

pygame.display.flip()

# Event loop

while 1:

for event in pygame.event.get():

if event.type == QUIT:

return

screen.blit(background, (0, 0))

pygame.display.flip()

if \_\_name\_\_ == '\_\_main\_\_': main()

**2.2. Basic Pygame objects**

As you can see, the code consists of **three main objects: the screen, the background, and the text**. Each of these objects is created by first calling an instance of an in-built Pygame object, and then modifying it to fit our needs. The screen is a slightly special case, because we still modify the display through Pygame calls, rather than calling methods belonging to the screen object. But for all other Pygame objects, we first create the object as a copy of a Pygame object, giving it some attributes, and build our game objects from them.

With the background, we first create a Pygame Surface object, and make it the size of the screen. We then perform the convert() operation to convert the Surface to a single pixel format. This is more obviously necessary when we have several images and surfaces, all of different pixel formats, which makes rendering them quite slow. By converting all the surfaces, we can drastically speed up rendering times. Finally, we fill the background surface with white (255, 255, 255). These values are RGB (Red Green Blue), and can be worked out from any good paint program.

With the text, we require more than one object. First, we create a font object, which defines which font to use, and the size of the font. Then we create a text object, by using the render method that belongs to our font object, supplying three arguments: the text to be rendered, whether or not it should be anti-aliased (1=yes, 0=no), and the color of the text (again in RGB format). Next we create a third text object, which gets the rectangle for the text. The easiest way to understand this is to imagine drawing a rectangle that will surround all of the text; you can then use this rectangle to get/set the position of the text on the screen. So in this example we get the rectangle, set its centerx attribute to be the centerx attribute of the background (so the text's center will be the same as the background's center, i.e. the text will be centered on the screen on the x axis). We could also set the y coordinate, but it's not any different so I left the text at the top of the screen. As the screen is small anyway, it didn't seem necessary.

**2.3. Blitting**

Now we have created our game objects, we need to actually render them. If we didn't and we ran the program, we'd just see a blank window, and the objects would remain invisible. The term used for rendering objects is blitting, which is where you copy the pixels belonging to said object onto the destination object. So to render the background object, you blit it onto the screen. In this example, to make things simple, we blit the text onto the background (so the background will now have a copy of the text on it), and then blit the background onto the screen.

Blitting is one of the slowest operations in any game, so you need to be careful not to blit too much onto the screen in every frame. If you have a background image, and a ball flying around the screen, then you could blit the background and then the ball in every frame, which would cover up the ball's previous position and render the new ball, but this would be pretty slow. A better solution is to blit the background onto the area that the ball previously occupied, which can be found by the ball's previous rectangle, and then blitting the ball, so that you are only blitting two small areas.

**2.4. The event loop**

Once you've set the game up, you need to put it into a loop so that it will continuously run until the user signals that he/she wants to exit. So you start an open while loop, and then for each iteration of the loop, which will be each frame of the game, update the game. The first thing is to check for any Pygame events, which will be the user hitting the keyboard, clicking a mouse button, moving a joystick, resizing the window, or trying to close it. In this case, we simply want to watch out for for user trying to quit the game by closing the window, in which case the game should return, which will end the while loop. Then we simply need to re-blit the background, and flip (update) the display to have everything drawn. OK, as nothing moves or happens in this example, we don't strictly speaking need to re-blit the background in every iteration, but I put it in because when things are moving around on the screen, you will need to do all your blitting here.

**2.5. Ta-da!**

And that's it - your most basic Pygame game! All games will take a form similar to this, but with lots more code for the actual game functions themselves, which are more to do your with programming, and less guided in structure by the workings of Pygame. This is what this tutorial is really about, and will now go onto.

**3. Kicking things off**

The first sections of code are relatively simple, and, once written, can usually be reused in every game you consequently make. They will do all of the boring, generic tasks like loading modules, loading images, opening networking connections, playing music, and so on. They will also include some simple but effective error handling, and any customisation you wish to provide on top of functions provided by modules like sys and pygame.

**3.1. The first lines, and loading modules**

First off, you need to start off your game and load up your modules. It's always a good idea to set a few things straight at the top of the main source file, such as the name of the file, what it contains, the license it is under, and any other helpful info you might want to give those will will be looking at it. Then you can load modules, with some error checking so that Python doesn't print out a nasty traceback, which non-programmers won't understand. The code is fairly simple, so I won't bother explaining any of it:

#!/usr/bin/env python

#

# Tom's Pong

# A simple pong game with realistic physics and AI

# http://www.tomchance.uklinux.net/projects/pong.shtml

#

# Released under the GNU General Public License

VERSION = "0.4"

try:

import sys

import random

import math

import os

import getopt

import pygame

from socket import \*

from pygame.locals import \*

except ImportError, err:

print "couldn't load module. %s" % (err)

sys.exit(2)

**3.2. Resource handling functions**

In the Line By Line Chimp example, the first code to be written was for loading images and sounds. As these were totally independent of any game logic or game objects, they were written as separate functions, and were written first so that later code could make use of them. I generally put all my code of this nature first, in their own, classless functions; these will, generally speaking, be resource handling functions. You can of course create classes for these, so that you can group them together, and maybe have an object with which you can control all of your resources. As with any good programming environment, it's up to you to develop your own best practice and style.

It's always a good idea to write your own resource handling functions, because although Pygame has methods for opening images and sounds, and other modules will have their methods of opening other resources, those methods can take up more than one line, they can require consistent modification by yourself, and they often don't provide satisfactory error handling. Writing resource handling functions gives you sophisticated, reusable code, and gives you more control over your resources. Take this example of an image loading function:

def load\_png(name):

""" Load image and return image object"""

fullname = os.path.join('data', name)

try:

image = pygame.image.load(fullname)

if image.get\_alpha() is None:

image = image.convert()

else:

image = image.convert\_alpha()

except pygame.error, message:

print 'Cannot load image:', fullname

raise SystemExit, message

return image, image.get\_rect()

Here we make a more sophisticated image loading function than the one provided by Pygame (image.load). Note that the first line of the function is a documentation string describing what the function does, and what object(s) it returns. The function assumes that all of your images are in a directory called data, and so it takes the filename and creates the full pathname, for example data/ball.png, using the os module to ensure cross-platform compatibility. Then it tries to load the image, and convert any alpha regions so you can achieve transparency, and it returns a more human-readable error if there's a problem. Finally it returns the image object, and its rect.

You can make similar functions for loading any other resources, such as loading sounds. You can also make resource handling classes, to give you more flexibility with more complex resources. For example, you could make a music class, with an \_\_init\_\_ function that loads the sound (perhaps borrowing from a load\_sound() function), a function to pause the music, and a function to restart. Another handy resource handling class is for network connections. Functions to open sockets, pass data with suitable security and error checking, close sockets, finger addresses, and other network tasks, can make writing a game with network capabilities relatively painless.

Remember the chief task of these functions/classes is to ensure that by the time you get around to writing game object classes, and the main loop, there's almost nothing left to do. Class inheritance can make these basic classes especially handy. Don't go overboard though; functions which will only be used by one class should be written as part of that class, not as a global function.

**4. Game object classes**

Once you've loaded your modules, and written your resource handling functions, you'll want to get on to writing some game objects. The way this is done is fairly simple, though it can seem complex at first. You write a class for each type of object in the game, and then create an instance of those classes for the objects. You can then use those classes' methods to manipulate the objects, giving objects some motion and interactive capabilities. So your game, in pseudo-code, will look like this:

#!/usr/bin/python

[load modules here]

[resource handling functions here]

class Ball:

[ball functions (methods) here]

[e.g. a function to calculate new position]

[and a function to check if it hits the side]

def main:

[initiate game environment here]

[create new object as instance of ball class]

ball = Ball()

while 1:

[check for user input]

[call ball's update function]

ball.update()

This is, of course, a very simple example, and you'd need to put in all the code, instead of those little bracketed comments. But you should get the basic idea. You crate a class, into which you put all the functions for a ball, including \_\_init\_\_, which would create all the ball's attributes, and update, which would move the ball to its new position, before blitting it onto the screen in this position.

You can then create more classes for all of your other game objects, and then create instances of them so that you can handle them easily in the main function and the main program loop. Contrast this with initiating the ball in the main function, and then having lots of classless functions to manipulate a set ball object, and you'll hopefully see why using classes is an advantage: It allows you to put all of the code for each object in one place; it makes using objects easier; it makes adding new objects, and manipulating them, more flexible. Rather than adding more code for each new ball object, you could simply create new instances of the Ball class for each new ball object. Magic!

**4.1. A simple ball class**

Here is a simple class with the functions necessary for creating a ball object that will, if the update function is called in the main loop, move across the screen:

class Ball(pygame.sprite.Sprite):

"""A ball that will move across the screen

Returns: ball object

Functions: update, calcnewpos

Attributes: area, vector"""

def \_\_init\_\_(self, vector):

pygame.sprite.Sprite.\_\_init\_\_(self)

self.image, self.rect = load\_png('ball.png')

screen = pygame.display.get\_surface()

self.area = screen.get\_rect()

self.vector = vector

def update(self):

newpos = self.calcnewpos(self.rect,self.vector)

self.rect = newpos

def calcnewpos(self,rect,vector):

(angle,z) = vector

(dx,dy) = (z\*math.cos(angle),z\*math.sin(angle))

return rect.move(dx,dy)

Here we have the Ball class, with an \_\_init\_\_ function that sets the ball up, an update function that changes the ball's rectangle to be in the new position, and a calcnewpos function to calculate the ball's new position based on its current position, and the vector by which it is moving. I'll explain the physics in a moment. The one other thing to note is the documentation string, which is a little bit longer this time, and explains the basics of the class. These strings are handy not only to yourself and other programmers looking at the code, but also for tools to parse your code and document it. They won't make much of a difference in small programs, but with large ones they're invaluable, so it's a good habit to get into.

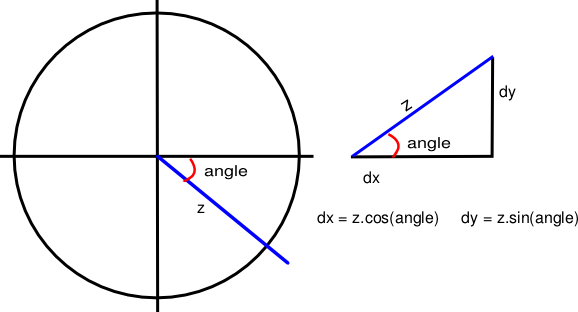
**4.1.1. Diversion 1: Sprites**

The other reason for creating a class for each object is sprites. Each image you render in your game will be a sprite object, and so to begin with, the class for each object should inherit the Sprite class. This is a really nice feature of Python - class inheritance. Now the Ball class has all of the functions that come with the Sprite class, and any object instances of the Ball class will be registered by Pygame as sprites. Whereas with text and the background, which don't move, it's OK to blit the object onto the background, Pygame handles sprite objects in a different manner, which you'll see when we look at the whole program's code.

Basically, you create both a ball object, and a sprite object for that ball, and you then call the ball's update function on the sprite object, thus updating the sprite. Sprites also give you sophisticated ways of determining if two objects have collided. Normally you might just check in the main loop to see if their rectangles overlap, but that would involve a lot of code, which would be a waste because the Sprite class provides two functions (spritecollide and groupcollide) to do this for you.

**4.1.2. Diversion 2: Vector physics**

Other than the structure of the Ball class, the notable thing about this code is the vector physics, used to calculate the ball's movement. With any game involving angular movement, you won't get very far unless you're comfortable with trigonometry, so I'll just introduce the basics you need to know to make sense of the calcnewpos function.

To begin with, you'll notice that the ball has an attribute vector, which is made up of angle and z. The angle is measured in radians, and will give you the direction in which the ball is moving. Z is the speed at which the ball moves. So by using this vector, we can determine the direction and speed of the ball, and therefore how much it will move on the x and y axes:

The diagram above illustrates the basic maths behind vectors. In the left hand diagram, you can see the ball's projected movement represented by the blue line. The length of that line (z) represents its speed, and the angle is the direction in which it will move. The angle for the ball's movement will always be taken from the x axis on the right, and it is measured clockwise from that line, as shown in the diagram.

From the angle and speed of the ball, we can then work out how much it has moved along the x and y axes. We need to do this because Pygame doesn't support vectors itself, and we can only move the ball by moving its rectangle along the two axes. So we need to resolve the angle and speed into its movement on the x axis (dx) and on the y axis (dy). This is a simple matter of trigonometry, and can be done with the formulae shown in the diagram.

https://www.pygame.org/docs/tut/tom/formulae.pngIf you've studied elementary trigonometry before, none of this should be news to you. But just in case you're forgetful, here are some useful formulae to remember, that will help you visualise the angles (I find it easier to visualise angles in degrees than in radians!)

**5. User-controllable objects**

So far you can create a Pygame window, and render a ball that will fly across the screen. The next step is to make some bats which the user can control. This is potentially far more simple than the ball, because it requires no physics (unless your user-controlled object will move in ways more complex than up and down, e.g. a platform character like Mario, in which case you'll need more physics). User-controllable objects are pretty easy to create, thanks to Pygame's event queue system, as you'll see.

**5.1. A simple bat class**

The principle behind the bat class is similar to that of the ball class. You need an \_\_init\_\_ function to initialise the ball (so you can create object instances for each bat), an update function to perform per-frame changes on the bat before it is blitted the bat to the screen, and the functions that will define what this class will actually do. Here's some sample code:

class Bat(pygame.sprite.Sprite):

"""Movable tennis 'bat' with which one hits the ball

Returns: bat object

Functions: reinit, update, moveup, movedown

Attributes: which, speed"""

def \_\_init\_\_(self, side):

pygame.sprite.Sprite.\_\_init\_\_(self)

self.image, self.rect = load\_png('bat.png')

screen = pygame.display.get\_surface()

self.area = screen.get\_rect()

self.side = side

self.speed = 10

self.state = "still"

self.reinit()

def reinit(self):

self.state = "still"

self.movepos = [0,0]

if self.side == "left":

self.rect.midleft = self.area.midleft

elif self.side == "right":

self.rect.midright = self.area.midright

def update(self):

newpos = self.rect.move(self.movepos)

if self.area.contains(newpos):

self.rect = newpos

pygame.event.pump()

def moveup(self):

self.movepos[1] = self.movepos[1] - (self.speed)

self.state = "moveup"

def movedown(self):

self.movepos[1] = self.movepos[1] + (self.speed)

self.state = "movedown"

As you can see, this class is very similar to the ball class in its structure. But there are differences in what each function does. First of all, there is a reinit function, which is used when a round ends, and the bat needs to be set back in its starting place, with any attributes set back to their necessary values. Next, the way in which the bat is moved is a little more complex than with the ball, because here its movement is simple (up/down), but it relies on the user telling it to move, unlike the ball which just keeps moving in every frame. To make sense of how the ball moves, it is helpful to look at a quick diagram to show the sequence of events:

https://www.pygame.org/docs/tut/tom/event-flowchart.png

What happens here is that the person controlling the bat pushes down on the key that moves the bat up. For each iteration of the main game loop (for every frame), if the key is still held down, then the state attribute of that bat object will be set to "moving", and the moveup function will be called, causing the ball's y position to be reduced by the value of the speed attribute (in this example, 10). In other words, so long as the key is held down, the bat will move up the screen by 10 pixels per frame. The state attribute isn't used here yet, but it's useful to know if you're dealing with spin, or would like some useful debugging output.

As soon as the player lets go of that key, the second set of boxes is invoked, and the state attribute of the bat object will be set back to "still", and the movepos attribute will be set back to [0,0], meaning that when the update function is called, it won't move the bat any more. So when the player lets go of the key, the bat stops moving. Simple!

**5.1.1. Diversion 3: Pygame events**

So how do we know when the player is pushing keys down, and then releasing them? With the Pygame event queue system, dummy! It's a really easy system to use and understand, so this shouldn't take long :) You've already seen the event queue in action in the basic Pygame program, where it was used to check if the user was quitting the application. The code for moving the bat is about as simple as that:

for event in pygame.event.get():

if event.type == QUIT:

return

elif event.type == KEYDOWN:

if event.key == K\_UP:

player.moveup()

if event.key == K\_DOWN:

player.movedown()

elif event.type == KEYUP:

if event.key == K\_UP or event.key == K\_DOWN:

player.movepos = [0,0]

player.state = "still"

Here assume that you've already created an instance of a bat, and called the object player. You can see the familiar layout of the for structure, which iterates through each event found in the Pygame event queue, which is retrieved with the event.get() function. As the user hits keys, pushes mouse buttons and moves the joystick about, those actions are pumped into the Pygame event queue, and left there until dealt with. So in each iteration of the main game loop, you go through these events, checking if they're ones you want to deal with, and then dealing with them appropriately. The event.pump() function that was in the Bat.update function is then called in every iteration to pump out old events, and keep the queue current.

First we check if the user is quitting the program, and quit it if they are. Then we check if any keys are being pushed down, and if they are, we check if they're the designated keys for moving the bat up and down. If they are, then we call the appropriate moving function, and set the player state appropriately (though the states moveup and movedown and changed in the moveup() and movedown() functions, which makes for neater code, and doesn't break encapsulation, which means that you assign attributes to the object itself, without referring to the name of the instance of that object). Notice here we have three states: still, moveup, and movedown. Again, these come in handy if you want to debug or calculate spin. We also check if any keys have been "let go" (i.e. are no longer being held down), and again if they're the right keys, we stop the bat from moving.

**6. Putting it all together**

So far you've learnt all the basics necessary to build a simple game. You should understand how to create Pygame objects, how Pygame displays objects, how it handles events, and how you can use physics to introduce some motion into your game. Now I'll just show how you can take all those chunks of code and put them together into a working game. What we need first is to let the ball hit the sides of the screen, and for the bat to be able to hit the ball, otherwise there's not going to be much game play involved. We do this using Pygame's collision methods.

**6.1. Let the ball hit sides**

The basics principle behind making it bounce of the sides is easy to grasp. You grab the coordinates of the four corners of the ball, and check to see if they correspond with the x or y coordinate of the edge of the screen. So if the top right and top left corners both have a y coordinate of zero, you know that the ball is currently on the top edge of the screen. We do all this in the update function, after we've worked out the new position of the ball.

if not self.area.contains(newpos):

tl = not self.area.collidepoint(newpos.topleft)

tr = not self.area.collidepoint(newpos.topright)

bl = not self.area.collidepoint(newpos.bottomleft)

br = not self.area.collidepoint(newpos.bottomright)

if tr and tl or (br and bl):

angle = -angle

if tl and bl:

self.offcourt(player=2)

if tr and br:

self.offcourt(player=1)

self.vector = (angle,z)

Here we check to see if the area contains the new position of the ball (it always should, so we needn't have an else clause, though in other circumstances you might want to consider it. We then check if the coordinates for the four corners are colliding with the area's edges, and create objects for each result. If they are, the objects will have a value of 1, or TRUE. If they don't, then the value will be None, or FALSE. We then see if it has hit the top or bottom, and if it has we change the ball's direction. Handily, using radians we can do this by simply reversing its positive/negative value. We also check to see if the ball has gone off the sides, and if it has we call the offcourt function. This, in my game, resets the ball, adds 1 point to the score of the player specified when calling the function, and displays the new score.

Finally, we recompile the vector based on the new angle. And that is it. The ball will now merrily bounce off the walls and go offcourt with good grace.

**6.2. Let the ball hit bats**

Making the ball hit the bats is very similar to making it hit the sides of the screen. We still use the collide method, but this time we check to see if the rectangles for the ball and either bat collide. In this code I've also put in some extra code to avoid various glitches. You'll find that you'll have to put all sorts of extra code in to avoid glitches and bugs, so it's good to get used to seeing it.

else:

# Deflate the rectangles so you can't catch a ball behind the bat

player1.rect.inflate(-3, -3)

player2.rect.inflate(-3, -3)

# Do ball and bat collide?

# Note I put in an odd rule that sets self.hit to 1 when they collide, and unsets it in the next

# iteration. this is to stop odd ball behaviour where it finds a collision \*inside\* the

# bat, the ball reverses, and is still inside the bat, so bounces around inside.

# This way, the ball can always escape and bounce away cleanly

if self.rect.colliderect(player1.rect) == 1 and not self.hit:

angle = math.pi - angle

self.hit = not self.hit

elif self.rect.colliderect(player2.rect) == 1 and not self.hit:

angle = math.pi - angle

self.hit = not self.hit

elif self.hit:

self.hit = not self.hit

self.vector = (angle,z)

We start this section with an else statement, because this carries on from the previous chunk of code to check if the ball hits the sides. It makes sense that if it doesn't hit the sides, it might hit a bat, so we carry on the conditional statement. The first glitch to fix is to shrink the players' rectangles by 3 pixels in both dimensions, to stop the bat catching a ball that goes behind them (if you imagine you just move the bat so that as the ball travels behind it, the rectangles overlap, and so normally the ball would then have been "hit" - this prevents that).

Next we check if the rectangles collide, with one more glitch fix. Notice that I've commented on these odd bits of code - it's always good to explain bits of code that are abnormal, both for others who look at your code, and so you understand it when you come back to it. The without the fix, the ball might hit a corner of the bat, change direction, and one frame later still find itself inside the bat. Then it would again think it has been hit, and change its direction. This can happen several times, making the ball's motion completely unrealistic. So we have a variable, self.hit, which we set to TRUE when it has been hit, and FALSE one frame later. When we check if the rectangles have collided, we also check if self.hit is TRUE/FALSE, to stop internal bouncing.

The important code here is pretty easy to understand. All rectangles have a colliderect function, into which you feed the rectangle of another object, which returns 1 (TRUE) if the rectangles do overlap, and 0 (FALSE) if not. If they do, we can change the direction by subtracting the current angle from pi (again, a handy trick you can do with radians, which will adjust the angle by 90 degrees and send it off in the right direction; you might find at this point that a thorough understanding of radians is in order!). Just to finish the glitch checking, we switch self.hit back to FALSE if it's the frame after they were hit.

We also then recompile the vector. You would of course want to remove the same line in the previous chunk of code, so that you only do this once after the if-else conditional statement. And that's it! The combined code will now allow the ball to hit sides and bats.

**6.3. The Finished product**

The final product, with all the bits of code thrown together, as well as some other bits of code to glue it all together, will look like this:

#!/usr/bin/python

#

# Tom's Pong

# A simple pong game with realistic physics and AI

# http://www.tomchance.uklinux.net/projects/pong.shtml

#

# Released under the GNU General Public License

VERSION = "0.4"

try:

import sys

import random

import math

import os

import getopt

import pygame

from socket import \*

from pygame.locals import \*

except ImportError, err:

print "couldn't load module. %s" % (err)

sys.exit(2)

def load\_png(name):

""" Load image and return image object"""

fullname = os.path.join('data', name)

try:

image = pygame.image.load(fullname)

if image.get\_alpha is None:

image = image.convert()

else:

image = image.convert\_alpha()

except pygame.error, message:

print 'Cannot load image:', fullname

raise SystemExit, message

return image, image.get\_rect()

class Ball(pygame.sprite.Sprite):

"""A ball that will move across the screen

Returns: ball object

Functions: update, calcnewpos

Attributes: area, vector"""

def \_\_init\_\_(self, (xy), vector):

pygame.sprite.Sprite.\_\_init\_\_(self)

self.image, self.rect = load\_png('ball.png')

screen = pygame.display.get\_surface()

self.area = screen.get\_rect()

self.vector = vector

self.hit = 0

def update(self):

newpos = self.calcnewpos(self.rect,self.vector)

self.rect = newpos

(angle,z) = self.vector

if not self.area.contains(newpos):

tl = not self.area.collidepoint(newpos.topleft)

tr = not self.area.collidepoint(newpos.topright)

bl = not self.area.collidepoint(newpos.bottomleft)

br = not self.area.collidepoint(newpos.bottomright)

if tr and tl or (br and bl):

angle = -angle

if tl and bl:

#self.offcourt()

angle = math.pi - angle

if tr and br:

angle = math.pi - angle

#self.offcourt()

else:

# Deflate the rectangles so you can't catch a ball behind the bat

player1.rect.inflate(-3, -3)

player2.rect.inflate(-3, -3)

# Do ball and bat collide?

# Note I put in an odd rule that sets self.hit to 1 when they collide, and unsets it in the next

# iteration. this is to stop odd ball behaviour where it finds a collision \*inside\* the

# bat, the ball reverses, and is still inside the bat, so bounces around inside.

# This way, the ball can always escape and bounce away cleanly

if self.rect.colliderect(player1.rect) == 1 and not self.hit:

angle = math.pi - angle

self.hit = not self.hit

elif self.rect.colliderect(player2.rect) == 1 and not self.hit:

angle = math.pi - angle

self.hit = not self.hit

elif self.hit:

self.hit = not self.hit

self.vector = (angle,z)

def calcnewpos(self,rect,vector):

(angle,z) = vector

(dx,dy) = (z\*math.cos(angle),z\*math.sin(angle))

return rect.move(dx,dy)

class Bat(pygame.sprite.Sprite):

"""Movable tennis 'bat' with which one hits the ball

Returns: bat object

Functions: reinit, update, moveup, movedown

Attributes: which, speed"""

def \_\_init\_\_(self, side):

pygame.sprite.Sprite.\_\_init\_\_(self)

self.image, self.rect = load\_png('bat.png')

screen = pygame.display.get\_surface()

self.area = screen.get\_rect()

self.side = side

self.speed = 10

self.state = "still"

self.reinit()

def reinit(self):

self.state = "still"

self.movepos = [0,0]

if self.side == "left":

self.rect.midleft = self.area.midleft

elif self.side == "right":

self.rect.midright = self.area.midright

def update(self):

newpos = self.rect.move(self.movepos)

if self.area.contains(newpos):

self.rect = newpos

pygame.event.pump()

def moveup(self):

self.movepos[1] = self.movepos[1] - (self.speed)

self.state = "moveup"

def movedown(self):

self.movepos[1] = self.movepos[1] + (self.speed)

self.state = "movedown"

def main():

# Initialise screen

pygame.init()

screen = pygame.display.set\_mode((640, 480))

pygame.display.set\_caption('Basic Pong')

# Fill background

background = pygame.Surface(screen.get\_size())

background = background.convert()

background.fill((0, 0, 0))

# Initialise players

global player1

global player2

player1 = Bat("left")

player2 = Bat("right")

# Initialise ball

speed = 13

rand = ((0.1 \* (random.randint(5,8))))

ball = Ball((0,0),(0.47,speed))

# Initialise sprites

playersprites = pygame.sprite.RenderPlain((player1, player2))

ballsprite = pygame.sprite.RenderPlain(ball)

# Blit everything to the screen

screen.blit(background, (0, 0))

pygame.display.flip()

# Initialise clock

clock = pygame.time.Clock()

# Event loop

while 1:

# Make sure game doesn't run at more than 60 frames per second

clock.tick(60)

for event in pygame.event.get():

if event.type == QUIT:

return

elif event.type == KEYDOWN:

if event.key == K\_a:

player1.moveup()

if event.key == K\_z:

player1.movedown()

if event.key == K\_UP:

player2.moveup()

if event.key == K\_DOWN:

player2.movedown()

elif event.type == KEYUP:

if event.key == K\_a or event.key == K\_z:

player1.movepos = [0,0]

player1.state = "still"

if event.key == K\_UP or event.key == K\_DOWN:

player2.movepos = [0,0]

player2.state = "still"

screen.blit(background, ball.rect, ball.rect)

screen.blit(background, player1.rect, player1.rect)

screen.blit(background, player2.rect, player2.rect)

ballsprite.update()

playersprites.update()

ballsprite.draw(screen)

playersprites.draw(screen)

pygame.display.flip()

if \_\_name\_\_ == '\_\_main\_\_': main()

As well as showing you the final product, I'll point you back to TomPong, upon which all of this is based. Download it, have a look at the source code, and you'll see a full implementation of pong using all of the code you've seen in this tutorial, as well as lots of other code I've added in various versions, such as some extra physics for spinning, and various other bug and glitch fixes.

Oh, find TomPong at <http://www.tomchance.uklinux.net/projects/pong.shtml>.