Line By Line Chimp

Author: Pete Shinners
Contact: pete@shinners.org

Introduction

In the *pygame* examples there is a simple example named "chimp". This example simulates a punchable monkey moving around the screen with promises of riches and reward. The example itself is very simple, and a bit thin on error-checking code. This example program demonstrates many of pygame's abilities, like creating a window, loading images and sounds, rendering text, and basic event and mouse handling.

The program and images can be found inside the standard source distribution of pygame. You can run it by running python -m pygame.examples.chimp in your terminal.

This tutorial will go through the code block by block. Explaining how the code works. There will also be mention of how the code could be improved and what error checking could help out.

This is an excellent tutorial for people getting their first look at the *pygame* code. Once *pygame* is fully installed, you can find and run the chimp demo for yourself in the examples directory.

(no, this is not a banner ad, it's the screenshot)

Full Source

Import Modules

This is the code that imports all the needed modules into your program. It also checks for the availability of some of the optional pygame modules.

```
# Import Modules
import os
import pygame as pg

if not pg.font:
    print("Warning, fonts disabled")
if not pg.mixer:
    print("Warning, sound disabled")

main_dir = os.path.split(os.path.abspath(__file__))[0]
data_dir = os.path.join(main_dir, "data")
```

First, we import the standard "os" python module. This allow us to do things like create platform independent file paths.

In the next line, we import the pygame package. In our case, we import pygame as `pg`, so that all of the functionality of pygame is able to be referenced from the namespace `pg`.

Some pygame modules are optional, and if they aren't found, they evaluate to `False`. Because of that, we decide to print a nice warning message if the **`font`** or **`mixer`** modules in pygame are not available. (Although they will only be unavailable in very uncommon situations).

Lastly, we prepare two paths for the rest of the code to use. `main_dir` uses the os.path module and the __file__ variable provided by Python to locate the game's python file, and extract the folder from that path. It then prepares the variable `data_dir` to tell the loading functions exactly where to look.

Loading Resources

Here we have two functions we can use to load images and sounds. We will look at each function individually in this section.

```
def load_image(name, colorkey=None, scale=1):
    fullname = os.path.join(data_dir, name)
    image = pg.image.load(fullname)

    size = image.get_size()
    size = (size[0] * scale, size[1] * scale)
    image = pg.transform.scale(image, size)

    image = image.convert()
    if colorkey is not None:
        if colorkey == -1:
            colorkey = image.get_at((0, 0))
        image.set_colorkey(colorkey, pg.RLEACCEL)
    return image, image.get_rect()
```

This function takes the name of an image to load. It also optionally takes an argument it can use to set a colorkey for the image, and an argument to scale the image. A colorkey is used in graphics to represent a color of the image that is transparent.

The first thing this function does is create a full pathname to the file. In this example all the resources are in a "data" subdirectory. By using the os.path.join function, a pathname will be created that works for whatever platform the game is running on.

Next we load the image using the **pygame.image.load()** function. After the image is loaded, we make an important call to the convert() function. This makes a new copy of a Surface and converts its color format and depth to match the display. This means blitting the image to the screen will happen as quickly as possible.

We then scale the image, using the **pygame.transform.scale()** function. This function takes a Surface and the size it should be scaled to. To scale by a scalar, we can get the size and scale the x and y by the scalar.

Last, we set the colorkey for the image. If the user supplied an argument for the colorkey argument we use that value as the colorkey for the image. This would usually just be a color RGB value, like (255, 255, 255) for white. You can also pass a value of -1 as the colorkey. In this case the function will lookup the color at the topleft pixel of the image, and use that color for the colorkey.

```
def load_sound(name):
    class NoneSound:
        def play(self):
            pass

    if not pg.mixer or not pg.mixer.get_init():
        return NoneSound()

    fullname = os.path.join(data_dir, name)
    sound = pg.mixer.Sound(fullname)

    return sound
```

Next is the function to load a sound file. The first thing this function does is check to see if the **pygame.mixer** module was imported correctly. If not, it returns a small class instance that has a dummy play method. This will act enough like a normal Sound object for this game to run without any extra error checking.

This function is similar to the image loading function, but handles some different problems. First we create a full path to the sound image, and load the sound file. Then we simply return the loaded Sound object.

Game Object Classes

Here we create two classes to represent the objects in our game. Almost all the logic for the game goes into these two classes. We will look over them one at a time here.

```
class Fist(pg.sprite.Sprite):
    """moves a clenched fist on the screen, following the mous

    def __init__(self):
        pg.sprite.Sprite.__init__(self)  # call Sprite initial
        self.image, self.rect = load_image("fist.png", -1)
        self.fist_offset = (-235, -80)
        self.punching = False

    def update(self):
        """move the fist based on the mouse position"""
        pos = pg.mouse.get_pos()
        self.rect.topleft = pos
```

```
        self.rect.move_ip(self.fist_offset)
        if self.punching:
            self.rect.move_ip(15, 25)

    def punch(self, target):
        """returns true if the fist collides with the target"""
        if not self.punching:
            self.punching = True
            hitbox = self.rect.inflate(-5, -5)
            return hitbox.colliderect(target.rect)

    def unpunch(self):
        """called to pull the fist back"""
        self.punching = False
```

Here we create a class to represent the players fist. It is derived from the Sprite class included in the **pygame.sprite** module. The __init__ function is called when new instances of this class are created. The first thing we do is be sure to call the __init__ function for our base class. This allows the Sprite's __init__ function to prepare our object for use as a sprite. This game uses one of the sprite drawing Group classes. These classes can draw sprites that have an "image" and "rect" attribute. By simply changing these two attributes, the renderer will draw the current image at the current position.

All sprites have an update() method. This function is typically called once per frame. It is where you should put code that moves and updates the variables for the sprite. The update() method for the fist moves the fist to the location of the mouse pointer. It also offsets the fist position slightly if the fist is in the "punching" state.

The following two functions punch() and unpunch() change the punching state for the fist. The punch() method also returns a true value if the fist is colliding with the given target sprite.

```
class Chimp(pg.sprite.Sprite):
    """moves a monkey critter across the screen. it can spin t
    monkey when it is punched."""

    def __init__(self):
        pg.sprite.Sprite.__init__(self)  # call Sprite initial
        self.image, self.rect = load_image("chimp.png", -1, 4)
        screen = pg.display.get_surface()
        self.area = screen.get_rect()
        self.rect.topleft = 10, 90
        self.move = 18
        self.dizzy = False

    def update(self):
        """walk or spin, depending on the monkeys state"""
        if self.dizzy:
            self._spin()
        else:
            self._walk()

    def _walk(self):
        """move the monkey across the screen, and turn at the
        newpos = self.rect.move((self.move, 0))
        if not self.area.contains(newpos):
            if self.rect.left < self.area.left or self.rect.ri
```

```
                self.move = -self.move
                newpos = self.rect.move((self.move, 0))
                self.image = pg.transform.flip(self.image, Tru
        self.rect = newpos

    def _spin(self):
        """spin the monkey image"""
        center = self.rect.center
        self.dizzy = self.dizzy + 12
        if self.dizzy >= 360:
            self.dizzy = False
            self.image = self.original
        else:
            rotate = pg.transform.rotate
            self.image = rotate(self.original, self.dizzy)
        self.rect = self.image.get_rect(center=center)

    def punched(self):
        """this will cause the monkey to start spinning"""
        if not self.dizzy:
            self.dizzy = True
            self.original = self.image
```

The Chimp class is doing a little more work than the fist, but nothing more complex. This class will move the chimp back and forth across the screen. When the monkey is punched, he will spin around to exciting effect. This class is also derived from the base **Sprite** class, and is initialized the same as the fist. While initializing, the class also sets the attribute "area" to be the size of the display screen.

The update function for the chimp simply looks at the current "dizzy" state, which is true when the monkey is spinning from a punch. It calls either the _spin or _walk method. These functions are prefixed with an underscore. This is just a standard python idiom which suggests these methods should only be used by the Chimp class. We could go so far as to give them a double underscore, which would tell python to really try to make them private methods, but we don't need such protection. :)

The _walk method creates a new position for the monkey by moving the current rect by a given offset. If this new position crosses outside the display area of the screen, it reverses the movement offset. It also mirrors the image using the **pygame.transform.flip()** function. This is a crude effect that makes the monkey look like he's turning the direction he is moving.

The _spin method is called when the monkey is currently "dizzy". The dizzy attribute is used to store the current amount of rotation. When the monkey has rotated all the way around (360 degrees) it resets the monkey image back to the original, non-rotated version. Before calling the **pygame.transform.rotate()** function, you'll see the code makes a local reference to the function simply named "rotate". There is no need to do that for this example, it is just done here to keep the following line's length a little shorter. Note that when calling the rotate function, we are always rotating from the original monkey image. When rotating, there is a slight loss of quality. Repeatedly rotating the same image and the quality would get worse each time. Also, when rotating an image, the size of the image will actually change. This is because the corners of the image will be

rotated out, making the image bigger. We make sure the center of the new image matches the center of the old image, so it rotates without moving.

The last method is punched() which tells the sprite to enter its dizzy state. This will cause the image to start spinning. It also makes a copy of the current image named "original".

Initialize Everything

Before we can do much with pygame, we need to make sure its modules are initialized. In this case we will also open a simple graphics window. Now we are in the main() function of the program, which actually runs everything.

```
pg.init()
screen = pg.display.set_mode((1280, 480), pg.SCALED)
pg.display.set_caption("Monkey Fever")
pg.mouse.set_visible(False)
```

The first line to initialize *pygame* takes care of a bit of work for us. It checks through the imported *pygame* modules and attempts to initialize each one of them. It is possible to go back and check if modules failed to initialize, but we won't bother here. It is also possible to take a lot more control and initialize each specific module by hand. That type of control is generally not needed, but is available if you desire.

Next we set up the display graphics mode. Note that the **`pygame.display`** module is used to control all the display settings. In this case we are asking for a 1280 by 480 window, with the `SCALED` display flag. This automatically scales up the window for displays much larger than the window.

Last we set the window title and turn off the mouse cursor for our window. Very basic to do, and now we have a small black window ready to do our bidding. Usually the cursor defaults to visible, so there is no need to really set the state unless we want to hide it.

Create The Background

Our program is going to have text message in the background. It would be nice for us to create a single surface to represent the background and repeatedly use that. The first step is to create the surface.

```
background = pg.Surface(screen.get_size())
background = background.convert()
background.fill((170, 238, 187))
```

This creates a new surface for us that is the same size as the display window. Note the extra call to convert() after creating the Surface. The convert with no arguments will make sure our background is the same format as the display window, which will give us the fastest results.

We also fill the entire background with a certain green color. The fill() function usually takes an RGB triplet as arguments, but supports many input formats. See the **`pygame.Color`** for all the color formats.

Put Text On The Background, Centered

Now that we have a background surface, lets get the text rendered to it. We only do this if we see the **pygame.font** module has imported properly. If not, we just skip this section.

```
if pg.font:
    font = pg.font.Font(None, 64)
    text = font.render("Pummel The Chimp, And Win $$$", True,
    textpos = text.get_rect(centerx=background.get_width() / 2
    background.blit(text, textpos)
```

As you see, there are a couple steps to getting this done. First we must create the font object and render it into a new surface. We then find the center of that new surface and blit (paste) it onto the background.

The font is created with the font module's Font() constructor. Usually you will pass the name of a TrueType font file to this function, but we can also pass None, which will use a default font. The Font constructor also needs to know the size of font we want to create.

We then render that font into a new surface. The render function creates a new surface that is the appropriate size for our text. In this case we are also telling render to create antialiased text (for a nice smooth look) and to use a dark grey color.

Next we need to find the centered position of the text on our display. We create a "Rect" object from the text dimensions, which allows us to easily assign it to the screen center.

Finally we blit (blit is like a copy or paste) the text onto the background image.

Display The Background While Setup Finishes

We still have a black window on the screen. Lets show our background while we wait for the other resources to load.

```
screen.blit(background, (0, 0))
pg.display.flip()
```

This will blit our entire background onto the display window. The blit is self explanatory, but what about this flip routine?

In pygame, changes to the display surface are not immediately visible. Normally, a display must be updated in areas that have changed for them to be visible to the user. In this case the flip() function works nicely because it simply handles the entire window area.

Prepare Game Object

Here we create all the objects that the game is going to need.

```
whiff_sound = load_sound("whiff.wav")
punch_sound = load_sound("punch.wav")
chimp = Chimp()
fist = Fist()
allsprites = pg.sprite.RenderPlain((chimp, fist))
clock = pg.time.Clock()
```

First we load two sound effects using the load_sound function we defined above. Then we create an instance of each of our sprite classes. And lastly we create a sprite **Group** which will contain all our sprites.

We actually use a special sprite group named **RenderPlain**. This sprite group can draw all the sprites it contains to the screen. It is called RenderPlain because there are actually more advanced Render groups. But for our game, we just need simple drawing. We create the group named "allsprites" by passing a list with all the sprites that should belong in the group. We could later on add or remove sprites from this group, but in this game we won't need to.

The clock object we create will be used to help control our game's framerate. we will use it in the main loop of our game to make sure it doesn't run too fast.

Main Loop

Nothing much here, just an infinite loop.

```
going = True
while going:
    clock.tick(60)
```

All games run in some sort of loop. The usual order of things is to check on the state of the computer and user input, move and update the state of all the objects, and then draw them to the screen. You'll see that this example is no different.

We also make a call to our clock object, which will make sure our game doesn't run faster than 60 frames per second.

Handle All Input Events

This is an extremely simple case of working the event queue.

```
for event in pg.event.get():
    if event.type == pg.QUIT:
        going = False
    elif event.type == pg.KEYDOWN and event.key == pg.K_ESCAPE
        going = False
    elif event.type == pg.MOUSEBUTTONDOWN:
        if fist.punch(chimp):
            punch_sound.play()  # punch
            chimp.punched()
        else:
            whiff_sound.play()  # miss
    elif event.type == pg.MOUSEBUTTONUP:
        fist.unpunch()
```

First we get all the available Events from pygame and loop through each of them. The first two tests see if the user has quit our game, or pressed the escape key. In these cases we just set `going` to `False`, allowing us out of the infinite loop.

Next we just check to see if the mouse button was pressed or released. If the button was pressed, we ask the fist object if it has collided with the

chimp. We play the appropriate sound effect, and if the monkey was hit, we tell him to start spinning (by calling his punched() method).

Update the Sprites

```
allsprites.update()
```

Sprite groups have an update() method, which simply calls the update method for all the sprites it contains. Each of the objects will move around, depending on which state they are in. This is where the chimp will move one step side to side, or spin a little farther if he was recently punched.

Draw The Entire Scene

Now that all the objects are in the right place, time to draw them.

```
screen.blit(background, (0, 0))
allsprites.draw(screen)
pg.display.flip()
```

The first blit call will draw the background onto the entire screen. This erases everything we saw from the previous frame (slightly inefficient, but good enough for this game). Next we call the draw() method of the sprite container. Since this sprite container is really an instance of the "RenderPlain" sprite group, it knows how to draw our sprites. Lastly, we flip() the contents of pygame's software double buffer to the screen. This makes everything we've drawn visible all at once.

Game Over

User has quit, time to clean up.

```
pg.quit()
```

Cleaning up the running game in *pygame* is extremely simple. Since all variables are automatically destructed, we don't really have to do anything, but calling pg.quit() explicitly cleans up pygame's internals.
