Rapid Game Development In Python

Author: Richard Jones

How do you make a game?

Games consist largely of user input, game output and some sort of world simulation. The most interesting games are the ones that do something different or new in the simulation. Python is a really nice language for writing game simulations in. Fortunately, other people have done a really excellent job of providing Python libraries for user input and game output.

Python has...

Python
: Possibly the best language for writing game-world simulations in. It's clear to read and write, easy to learn, handles a lot of programming house-keeping and is reasonably fast.

PyGame
: Provides user input handling (mouse, keyboard, joystick) and game output via screen (shape drawing, image blitting, font rendering) and speakers (effects and music). Strictly 2d graphics.

PyOpenGL
: Gives Python the power of OpenGL for very high-performance 3d graphics.

(Soya3d, PGU, ...)
: Additional libraries over the top of the above that provide a solid basis for your world simulation, game map drawing code, model loading, event management, etc.

Managing the screen - the basics

Basic screen initialisation looks like:

```
>>> from pygame.locals import *
>>> screen = pygame.display.set_mode((1024, 768))
>>> screen = pygame.display.set_mode((1024, 768), FULLSCREEN)
```

You can call `set_mode` during your game to switch from windowed (the default) to full-screen. Other display mode flags (you just `|` them together):

DOUBLEBUF
: must be used for smooth animation

OPENGL
: lets you draw 3d scenes with PyOpenGL, but won't let you perform most pygame drawing functions.

There is an optional bit depth flag, but it's almost always better to not include it and go with whatever platform default is available.

If you're using DOUBLEBUF then you'll need to *flip* the screen after you've rendered it. This is as simple as:

```
>>> pygame.display.flip()
```

Drawing a car

We're going to draw a car on screen. We do this using one of the most important drawing primitives, the BLIT (Block Image Transfer). It copies an image from one place (eg. your source image) to another place (eg. the screen at X=50, Y=100):

```
>>> car = pygame.image.load('car.png')
>>> screen.blit(car, (50, 100))
>>> pygame.display.flip()
```

At this point, the car should appear on the screen with its top-left corner positioned at (50, 100). We always start counting X coordinates from the left, and Y coordinates from the top of the screen.

We can also rotate images:

```
>>> import math
>>> rotated = pygame.transform.rotate(car, 45 * math.pi / 180)
>>> screen.blit(car, (50, 100))
>>> pygame.display.flip()
```

Animating the car

Animating anything on screen involves drawing a scene, clearing it and drawing it again slightly differently:

```
>>> for i in range(100):
...  screen.fill((0, 0, 0))
...  screen.blit(car, (i, 0))
```

Animation also consists of doing that pretty quickly.

Note also that clearing and redrawing a screen is quite an unoptimal way of animating. It's usually better to update the parts of the screen that have changed instead. Sprites, mentioned later, help us do this.

Input handling

There's a number of ways to get user events in PyGame, the most common of which are:

```
>>> import pygame
>>> pygame.event.wait()
>>> pygame.event.poll()
>>> pygame.event.get()
```

`wait` will sit and block further game execution until an event comes along. This is not generally very useful for games, as you want to be animating things at the same time. `poll` will see whether there are any events waiting for processing. If there's no events, it returns NOEVENT and you can do other things. The final form, `get`, is like `poll` except that it returns *all* of the currently-outstanding events (you may also filter the events it returns to be only key-presses, or mouse moves, etc.)

I should also mention timing here, as it's important to the user experience. Without timing control, your game will run as fast as it possibly can on whatever platform it happens to be on. Timing control is easy to add:

```
>>> clock = pygame.time.Clock()
>>> FRAMES_PER_SECOND = 30
>>> deltat = clock.tick(FRAMES_PER_SECOND)
```

`tick` instructs the clock object to pause until 1/30th of a second has passed since the last call to `tick`. This effectively limits the number of calls to `tick` to 30 per second. The *actual* time between `tick` calls is returned (in milliseconds) – on slower computers you might not be achieving 30 ticks per second.

It's important to note that the 30 frames per second will also determine how often your game responds to user input, as that is checked at the same time that the screen is drawn. Checking for user input any slower than 30 frames per second will result in noticeable delays for the user.

30 times a second is a reasonable number to aim for - our eyes generally won't benefit from anything above

30. If your game is action-oriented, you may wish to aim for double that so that players feel their input is being processed in a reasonably responsive manner.

Bringing together some elements

The following code will animate our little car according to user controls. It consists broadly of four sections (initialisation, user input, animation and rendering):

```
# INTIALISATION
import pygame, math, sys
from pygame.locals import *
screen = pygame.display.set_mode((1024, 768))
car = pygame.image.load('car.png')
clock = pygame.time.Clock()
k_up = k_down = k_left = k_right = 0
speed = direction = 0
position = (100, 100)
TURN_SPEED = 5
ACCELERATION = 2
MAX_FORWARD_SPEED = 10
MAX_REVERSE_SPEED = -5
BLACK = (0,0,0)

while 1:
    # USER INPUT
    clock.tick(30)
    for event in pygame.event.get():
        if not hasattr(event, 'key'): continue
        down = event.type == KEYDOWN     # key down or up?
        if event.key == K_RIGHT: k_right = down * -5
        elif event.key == K_LEFT: k_left = down * 5
        elif event.key == K_UP: k_up = down * 2
        elif event.key == K_DOWN: k_down = down * -2
        elif event.key == K_ESCAPE: sys.exit(0)     # quit the game
    screen.fill(BLACK)

    # SIMULATION
    # .. new speed and direction based on acceleration and turn
    speed += (k_up + k_down)
    if speed > MAX_FORWARD_SPEED: speed = MAX_FORWARD_SPEED
    if speed < MAX_REVERSE_SPEED: speed = MAX_REVERSE_SPEED
    direction += (k_right + k_left)
    # .. new position based on current position, speed and direction
    x, y = position
    rad = direction * math.pi / 180
    x += -speed*math.sin(rad)
    y += -speed*math.cos(rad)
    position = (x, y)

    # RENDERING
    # .. rotate the car image for direction
    rotated = pygame.transform.rotate(car, direction)
    # .. position the car on screen
    rect = rotated.get_rect()
    rect.center = position
    # .. render the car to screen
    screen.blit(rotated, rect)
    pygame.display.flip()
```

More structure

So the above code is a little ad-hoc. Most games will want to organise things to be able to better-control

simulation and rendering. To do this, we can use *sprites*. A sprite holds an image (e.g. a car) and information about where that image should be drawn on screen (i.e. its position.) This information is stored on the sprite's `image` and `rect` attributes.

Sprites are always dealt with in groups - even if a group only has one Sprite. Sprite groups have a `draw` method which draws the group's sprites onto a supplied surface. They also have a `clear` method which can remove their sprites from the surface. The above car code rewritten using a sprite:

```
# INTIALISATION
import pygame, math, sys
from pygame.locals import *
screen = pygame.display.set_mode((1024, 768))
clock = pygame.time.Clock()

class CarSprite(pygame.sprite.Sprite):
    MAX_FORWARD_SPEED = 10
    MAX_REVERSE_SPEED = 10
    ACCELERATION = 2
    TURN_SPEED = 5

    def __init__(self, image, position):
        pygame.sprite.Sprite.__init__(self)
        self.src_image = pygame.image.load(image)
        self.position = position
        self.speed = self.direction = 0
        self.k_left = self.k_right = self.k_down = self.k_up = 0

    def update(self, deltat):
        # SIMULATION
        self.speed += (self.k_up + self.k_down)
        if self.speed > self.MAX_FORWARD_SPEED:
            self.speed = self.MAX_FORWARD_SPEED
        if self.speed < -self.MAX_REVERSE_SPEED:
            self.speed = -self.MAX_REVERSE_SPEED
        self.direction += (self.k_right + self.k_left)
        x, y = self.position
        rad = self.direction * math.pi / 180
        x += -self.speed*math.sin(rad)
        y += -self.speed*math.cos(rad)
        self.position = (x, y)
        self.image = pygame.transform.rotate(self.src_image, self.direction)
        self.rect = self.image.get_rect()
        self.rect.center = self.position

# CREATE A CAR AND RUN
rect = screen.get_rect()
car = CarSprite('car.png', rect.center)
car_group = pygame.sprite.RenderPlain(car)
while 1:
    # USER INPUT
    deltat = clock.tick(30)
    for event in pygame.event.get():
        if not hasattr(event, 'key'): continue
        down = event.type == KEYDOWN
        if event.key == K_RIGHT: car.k_right = down * -5
        elif event.key == K_LEFT: car.k_left = down * 5
        elif event.key == K_UP: car.k_up = down * 2
        elif event.key == K_DOWN: car.k_down = down * -2
        elif event.key == K_ESCAPE: sys.exit(0)

    # RENDERING
    screen.fill((0,0,0))
    car_group.update(deltat)
    car_group.draw(screen)
    pygame.display.flip()
```

Note that mostly the code has just been moved around a little. The benefit of sprites really comes in when you have a lot of images to draw on screen.

PyGame sprites have additional functionality that help us determine collisions. Checking for collisions is really pretty easy. Let's put some pads to drive over into the simulation:

```
class PadSprite(pygame.sprite.Sprite):
    normal = pygame.image.load('pad_normal.png')
    hit = pygame.image.load('pad_hit.png')
    def __init__(self, position):
        self.rect = pygame.Rect(self.normal.get_rect())
        self.rect.center = position
    def update(self, hit_list):
        if self in hit_list: self.image = self.hit
        else: self.image = self.normal
pads = [
    PadSprite((200, 200)),
    PadSprite((800, 200)),
    PadSprite((200, 600)),
    PadSprite((800, 600)),
]
pad_group = pygame.sprite.RenderPlain(*pads)
```

now at the animation point, just before we draw the car, we check to see whether the car sprite is colliding with any of the pads, and pass that information to pad.update() so each pad knows whether to draw itself “hit” or not:

```
collisions = pygame.sprite.spritecollide(car_group, pad_group)
pad_group.update(collisions)
pad_group.draw(screen)
```

So now we have a car, running around on the screen, controlled by the player and we can detect when the car hits other things on the screen.

Adding objectives

It'd be nice if we could determine whether the car has made a "lap" of the "circuit" we've constructed. We'll keep information indicating which order the pads must be visited:

```
class PadSprite(pygame.sprite.Sprite):
    normal = pygame.image.load('pad_normal.png')
    hit = pygame.image.load('pad_hit.png')
    def __init__(self, number, position):
        pygame.sprite.Sprite.__init__(self)
        self.number = number
        self.rect = pygame.Rect(self.normal.get_rect())
        self.rect.center = position
        self.image = self.normal
pads = [
    PadSprite(1, (200, 200)),
    PadSprite(2, (800, 200)),
    PadSprite(3, (200, 600)),
    PadSprite(4, (800, 600)),
]
current_pad_number = 0
```

Now we replace the pad collision from above with code that makes sure we hit them in the correct order:

```
pads = pygame.sprite.spritecollide(car, pad_group, False)
if pads:
    pad = pads[0]
    if pad.number == current_pad_number + 1:
        pad.image = pad.hit
        current_pad_number += 1
elif current_pad_number == 4:
```

```
    for pad in pad_group.sprites(): pad.image = pad.normal
    current_pad_number = 0
```

The last part of that text, resetting the `current_pad_number` is where we'd flag that the player has run a lap.

Adding a background

Currently we're clearing the screen on every frame before rendering (`screen.fill((0,0,0))`). This is quite slow (though you might not notice) and is easily improved upon. Firstly outside the animation loop we load up a background image and draw it to the screen:

```
background = pygame.image.load('track.png')
screen.blit(self.background, (0,0))
```

Now inside the loop, but before we update (move) the car, we ask the car's sprite to clear itself from the screen. We do this with the pads too:

```
pad_group.clear(screen, background)
car_group.clear(screen, background)
```

Now we're only ever updating the small areas of screen that we need to update. A further optimisation would be to recognise that the pads only get updated very infrequently, and not draw / clear them each frame unless their state actually changes. This optimisation is not necessary just yet, and a good rule of thumb is to not optimise unless you really need to -- it just unnecessarily complicates your code.

Phil's PyGame Utilities (PGU)

pgu is an extension for PyGame which includes several tools and libraries.

tools
: the tools are a tile editor and a level editor (tile, isometric, hexagonal).

gui
: full featured gui, html rendering, document layout, and text rendering.

game libs
: the libraries include a sprite and tile engine (tile, isometric, hexagonal), a state engine, a timer, and a high score system.

Soya3d

Soya 3D is a high level 3D engine for Python; it aims at being to 3D what Python is to programming: easy and powerful. It is designed with games in mind, focusing both on performance and ease-of-use. It relies on OpenGL, SDL and Cal3D.

Soya 3D is available under the GPL and currently runs on GNU/Linux, though ports to other OS are planned (Mac OS X, Windows,...) since it uses only portable libraries.

Features includes:

- Object model, including camera, light, world, volume,...
- Particle systems
- Fullscreen
- Tutorials and demos included
- Trees

- Raypicking
- Landscapes
- 3D character animation (with Cal3D)
- Export scripts for Blender, Obj/Mtl, Maya and 3DSmax
- Event management (keyboard, mouse,...)
- Cell-shading
- Shadows
- Environment mapping

PyWeek One

The first Python Game Programming Challenge (PyWeek) was held over the last week in August / first week of September, 2005. The PyWeek challenge goals were:

1. Must be challenging and fun,
2. Entries must be developed during the challenge, and must incorporate some theme decided at the start of the challenge,
3. Will hopefully increase the public body of python game tools, code and expertise,
4. Will let a lot of people actually finish a game, and
5. May inspire new projects (with ready made teams!)

What happened:

- 170 individuals signed up
- 100 individuals actually competed (that drop-off is normal and expected)
- 26 final entries were submitted!
- It was quite a challenge and a whole lot of fun

The additional goals of run-off benefits (points 3 and 5) have come to fruition as well, with a number of the projects continuing to be developed after the challenge, and a number of the frameworks being improved as a result of the challenge.

The feedback from the competitors was almost universally positive (the only gripes from people who unfortunately got dragged away from the challenge part-way through).

The next challenge will be run somewhere around February to April 2006.

References

Python http://www.python.org
PyGame http://www.pygame.org
PyOpenGL http://pyopengl.sf.net
Soya3d http://home.gna.org/oomadness
PGU http://www.imitationpickles.org/pgu
PyWeek http://www.mechanicalcat.net/tech/PyWeek
Game Programming In Python Book by Sean Riley available published by Charles River Media