
qq Documentation

Release 1.0

Radomir Dopieralski

January 13, 2017

1	Tiles	3
1.1	Map definition	4
1.2	Drawing the map	5
2	Main loop	9
2.1	Initialization	9
2.2	Main game loop	9
2.3	Empty map	10
2.4	Tile cache	10
3	Sprites	13
3.1	Immobile objects	13
3.2	Animation	15
4	Player character	17
4.1	Main loop cleanup	17
5	Indices and tables	19

This is a sort of a tutorial about writing a tile-based animated game in [Python](#) using [PyGame](#). I'm writing it, because it took me some time to figure out a way that works well for me – it wasn't immediately obvious – and I'd like to save others some effort. Please don't treat it as one and the only, or even just best way to do it. Of course, I'm always interested about hearing about better ways. I'm also not going to tell you how to write the whole game: I will concentrate on the parts directly related to [PyGame](#). You are on your own with the rest.

You can get the final source code and all the graphics at <https://bitbucket.org/thesheep/qq>

Contents:

Tiles

Just to make sure we are on the same page: if you ever played any RPG games for Game Boy, NES, SNES or some of the titles on Game Boy Advance or Nintendo DS, you have surely noticed that the display consists of a scrolled map, composed of square (or rectangular) elements, with animated characters, monsters and other objects added to it. This is what we are going to make. Usually games also have a more or less sophisticated system of menus for inventory and combat, and of course plot and dialogs a lot of finer details – that’s outside the scope of this article.

As was said, the map is composed of rectangular elements, fit together on a grid. Sometimes there are so many of them and they are so well drawn that you can hardly tell where one of them begins and other ends. We will be calling them “tiles”, because they are arranged like tiles on a kitchen floor. Incidentally, they will sometimes be tiles of actual floor tiles, but also of grass, walls, ground, trees, etc. – usually most non-moving things in those games are created using tiles. The graphics for those tiles is usually taken from tilesets. Tilesets are, well sets of tiles, usually in a form of images containing rows of tiles aligned one next to the other, one of each tile type. Our program will slice them into single tiles, and “stamp” those on the screen to create the map. An example of a small tileset may look like this:



To use that tileset, we will need some code that will load it into memory and slice into single tiles. We will use something like this:

```
import pygame
import pygame.locals

def load_tile_table(filename, width, height):
    image = pygame.image.load(filename).convert()
    image_width, image_height = image.get_size()
    tile_table = []
    for tile_x in range(0, image_width/width):
        line = []
        tile_table.append(line)
        for tile_y in range(0, image_height/height):
            rect = (tile_x*width, tile_y*height, width, height)
            line.append(image.subsurface(rect))
    return tile_table

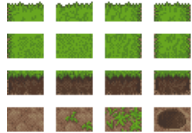
if __name__=='__main__':
    pygame.init()
    screen = pygame.display.set_mode((128, 98))
    screen.fill((255, 255, 255))
```

```

table = load_tile_table("ground.png", 24, 16)
for x, row in enumerate(table):
    for y, tile in enumerate(row):
        screen.blit(tile, (x*32, y*24))
pygame.display.flip()
while pygame.event.wait().type != pygame.locals.QUIT:
    pass

```

The function `load_tile_table()` will load tiles from a tileset file, and return them sliced up as a list of lists. At first I did that by creating separate surfaces, but later I found out that using `subsurface` is better – it doesn't create copies in memory. Once the tiles are loaded, they are displayed, a little spaced so that you can see where they end:



1.1 Map definition

We usually don't want to hardcode the maps in our game, so it's best to put them in a file. You might distract yourself by writing a map editor, or use an existing editor and merely distract yourself trying to figure out how to parse the files it produces. I took an easier route and just used a plain text file, parsed with the `ConfigParser`. I define the map as lines of characters, every character representing one map square. What exactly that square contains is defined later in the file, in sort of a legend. So, an example map definition may look like this (save it in a file "level.map"):

```

[level]
tileset = ground.png
map = .....
     .#####.##.
     .#...#...#.
     .#...####.
     .##.##....
     .....

[.]
name = floor
tile = 0, 3

[#]
name = wall
wall = true
block = true

```

This way, for every map square I have a dictionary of key-value pairs describing that square. If I need to add something, I just add a new character to the legend, with the description. Now, the parsing of that goes something like this:

```

import ConfigParser

class Level(object):
    def load_file(self, filename="level.map"):
        self.map = []
        self.key = {}
        parser = ConfigParser.ConfigParser()
        parser.read(filename)
        self.tileset = parser.get("level", "tileset")

```



```

self.map = parser.get("level", "map").split("\n")
for section in parser.sections():
    if len(section) == 1:
        desc = dict(parser.items(section))
        self.key[section] = desc
self.width = len(self.map[0])
self.height = len(self.map)

def get_tile(self, x, y):
    try:
        char = self.map[y][x]
    except IndexError:
        return {}
    try:
        return self.key[char]
    except KeyError:
        return {}

```

With this code, you can just use `get_tile()` to get a dict of values for any particular map square. We will add a couple of convenience methods for accessing often used values:

```

class Level(object): # Continued...
    def get_tile(self, x, y):
        """Tell what's at the specified position of the map."""

        try:
            char = self.map[y][x]
        except IndexError:
            return {}
        try:
            return self.key[char]
        except KeyError:
            return {}

    def get_bool(self, x, y, name):
        """Tell if the specified flag is set for position on the map."""

        value = self.get_tile(x, y).get(name)
        return value in (True, 1, 'true', 'yes', 'True', 'Yes', '1', 'on', 'On')

    def is_wall(self, x, y):
        """Is there a wall?"""

        return self.get_bool(x, y, 'wall')

    def is_blocking(self, x, y):
        """Is this place blocking movement?"""

        if not 0 <= x < self.width or not 0 <= y < self.height:
            return True
        return self.get_bool(x, y, 'block')

```

1.2 Drawing the map

Now we can just iterate over all map squares and draw appropriate tiles in the right places. This works fine for floor tiles and maybe some simple walls, but you might have noticed that we have a lot of different tiles for corners, straight

walls, etc. You could of course use different characters for them in your map definition, but making them match manually is pretty boring and can be easily automated. So we have a flag “wall” for our map squares, and when that flag is set we choose the right tile by looking at the neighboring squares. The code is not very pretty, but it does what we want. Oh, I forgot, we don’t draw our map directly on the screen, we draw it on a separate “background” surface, and we only do it once per map. Then we can draw that surface on the screen, draw some more elements on top and still be able to erase those elements, by drawing parts from the background over them. We might also make the background larger than the screen and implement scrolling, although I won’t cover that here.

This is enough if we have a “flat” map, for example only different kinds of terrain. But there is a problem if we want walls. Walls should obscure the view and so we need to draw parts of them on top of anything we add to the map. We will do it by keeping a dict of all the elements that need to be drawn on top, called overlays. Our finished map-drawing code may look like this:

```
class Level(object): # Continued...

    def is_wall(self, x, y):
        return self.get_bool(x, y, 'wall')

    def render(self):
        wall = self.is_wall
        tiles = MAP_CACHE[self.tileset]
        image = pygame.Surface((self.width*MAP_TILE_WIDTH, self.height*MAP_TILE_HEIGHT))
        overlays = {}
        for map_y, line in enumerate(self.map):
            for map_x, c in enumerate(line):
                if wall(map_x, map_y):
                    # Draw different tiles depending on neighbourhood
                    if not wall(map_x, map_y+1):
                        if wall(map_x+1, map_y) and wall(map_x-1, map_y):
                            tile = 1, 2
                        elif wall(map_x+1, map_y):
                            tile = 0, 2
                        elif wall(map_x-1, map_y):
                            tile = 2, 2
                        else:
                            tile = 3, 2
                    else:
                        if wall(map_x+1, map_y+1) and wall(map_x-1, map_y+1):
                            tile = 1, 1
                        elif wall(map_x+1, map_y+1):
                            tile = 0, 1
                        elif wall(map_x-1, map_y+1):
                            tile = 2, 1
                        else:
                            tile = 3, 1
                    # Add overlays if the wall may be obscuring something
                    if not wall(map_x, map_y-1):
                        if wall(map_x+1, map_y) and wall(map_x-1, map_y):
                            over = 1, 0
                        elif wall(map_x+1, map_y):
                            over = 0, 0
                        elif wall(map_x-1, map_y):
                            over = 2, 0
                        else:
                            over = 3, 0
                    overlays[(map_x, map_y)] = tiles[over[0]][over[1]]
                else:
                    try:
```

```
        tile = self.key[c]['tile'].split(',')
        tile = int(tile[0]), int(tile[1])
    except (ValueError, KeyError):
        # Default to ground tile
        tile = 0, 3
    tile_image = tiles[tile[0]][tile[1]]
    image.blit(tile_image,
               (map_x*MAP_TILE_WIDTH, map_y*MAP_TILE_HEIGHT))
    return image, overlays
```

Then we can draw the background image on the screen, followed by all the movable objects and sprites, and then draw the overlays on top of them. We will keep them as a sprite group for convenience.

Main loop

2.1 Initialization

In order to make use of that map of ours, we need to actually call the renderer code somewhere. It's time to create the heart of our game – the main game loop.

First we need to initialize some globals that our game will be using, load the map tiles and initialize PyGame's screen and clock. Then we draw out map on a separate surface for the background, and blit that to the screen, refreshing it afterwards with flip – that refreshes the whole screen, we will later see how to only refresh the parts that changed:

```
if __name__ == "__main__":
    screen = pygame.display.set_mode((424, 320))

    MAP_TILE_WIDTH = 24
    MAP_TILE_HEIGHT = 16
    MAP_CACHE = {
        'ground.png': load_tile_table('ground.png', MAP_TILE_WIDTH,
                                      MAP_TILE_HEIGHT),
    }

    level = Level()
    level.load_file('level.map')

    clock = pygame.time.Clock()

    background, overlay_dict = level.render()
    overlays = pygame.sprite.RenderUpdates()
    for (x, y), image in overlay_dict.iteritems():
        overlay = pygame.sprite.Sprite(overlays)
        overlay.image = image
        overlay.rect = image.get_rect().move(x * 24, y * 16 - 16)
    screen.blit(background, (0, 0))
    overlays.draw(screen)
    pygame.display.flip()
```

2.2 Main game loop

For now the game loop will just redraw the overlays 15 times per second, and react to any events: record keypresses and exit when you close the window:

```
game_over = False
while not game_over:

    # XXX draw all the objects here

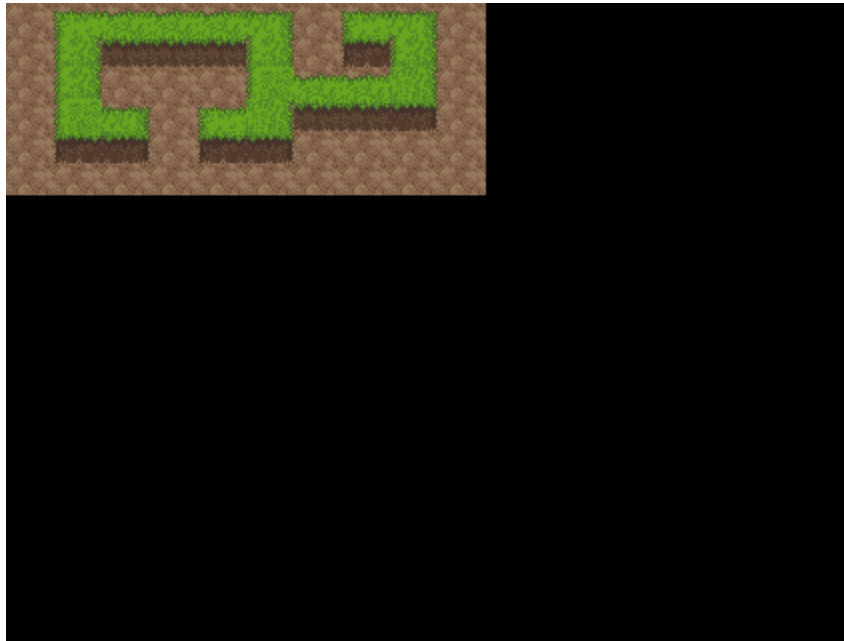
    overlays.draw(screen)
    pygame.display.flip()
    clock.tick(15)
    for event in pygame.event.get():
        if event.type == pygame.locals.QUIT:
            game_over = True
        elif event.type == pygame.locals.KEYDOWN:
            pressed_key = event.key
```

We are doing 15 iterations of the loop per second, using pygame’s clock. In each iteration we will draw our objects on the background, and then draw the overlays over them.

Later we will move the main loop into a separate “Game” object, and put some of the state we need to keep track of on it. For now it’s enough.

2.3 Empty map

By now you should have a program that you can run and that displays roughly something like this:



You can edit the “map.txt” file and see how it changes the map displayed. You can also change the tileset image, but remember to add it to the loading code.

2.4 Tile cache

Actually, adding the loading to the initialization code every time you add a new tileset is kind of lame. We can do better than that: we will make a cache object that will act as a dict, but will automatically load any tilesets we request:

```

class TileCache:
    """Load the tilesets lazily into global cache"""

    def __init__(self, width=32, height=None):
        self.width = width
        self.height = height or width
        self.cache = {}

    def __getitem__(self, filename):
        """Return a table of tiles, load it from disk if needed."""

        key = (filename, self.width, self.height)
        try:
            return self.cache[key]
        except KeyError:
            tile_table = self._load_tile_table(filename, self.width,
                                                self.height)

            self.cache[key] = tile_table
            return tile_table

    def _load_tile_table(self, filename, width, height):
        """Load an image and split it into tiles."""

        image = pygame.image.load(filename).convert()
        image_width, image_height = image.get_size()
        tile_table = []
        for tile_x in range(0, image_width/width):
            line = []
            tile_table.append(line)
            for tile_y in range(0, image_height/height):
                rect = (tile_x*width, tile_y*height, width, height)
                line.append(image.subsurface(rect))
        return tile_table

```

Now we can just do:

```
MAP_CACHE = TileCache(MAP_TILE_WIDTH, MAP_TILE_HEIGHT)
```


3.1 Immobile objects

Our map is kinda neat by now, but it's still a little bit empty. Before we populate it with the player character and monsters, lets just put some immobile crates in it. It should be easier, as they are not animated.

We will need to add it to our map:

```
[level]
tileset = ground.png
map = .....
      .#####.##.
      .#...#...#.
      .#.b####.
      .##.##..b.
      .....b...

[.]
name = floor
tile = 0, 3

[#]
name = wall
wall = true
block = true

[b]
name = crate
tile = 0, 3
block = true
sprite = crate.png
```

Now we need to modify our level loading code to use the “sprite” attribute:

```
self.items = {}
for y, line in enumerate(self.map):
    for x, c in enumerate(line):
        if not self.is_wall(x, y) and 'sprite' in self.key[c]:
            self.items[(x, y)] = self.key[c]
```

Then we will load the sprite graphics into a separate cachem and keep a group of sprites for our level in our main loop. Our sprites will all be 32 by 32 pixels:

```
level = Level()
level.load_file('level.map')

SPRITE_CACHE = TileCache(32, 32)
sprites = pygame.sprite.RenderUpdates()
for pos, tile in level.items.iteritems():
    sprite = Sprite(pos, SPRITE_CACHE[tile["sprite"]])
    sprites.add(sprite)
```

We still need to define our Sprite class. For now it will just use the very first frame as the image:

```
class Sprite(pygame.sprite.Sprite):
    def __init__(self, pos=(0, 0), frames=None):
        super(Sprite, self).__init__()
        self.image = frames[0][0]
        self.rect = self.image.get_rect()
        self.pos = pos

    def _get_pos(self):
        """Check the current position of the sprite on the map."""

        return (self.rect.midbottom[0]-12)/24, (self.rect.midbottom[1]-16)/16

    def _set_pos(self, pos):
        """Set the position and depth of the sprite on the map."""

        self.rect.midbottom = pos[0]*24+12, pos[1]*16+16
        self.depth = self.rect.midbottom[1]

    pos = property(_get_pos, _set_pos)

    def move(self, dx, dy):
        """Change the position of the sprite on screen."""

        self.rect.move_ip(dx, dy)
        self.depth = self.rect.midbottom[1]
```

Finally, we can draw all those sprites in our main loop, just before we draw the overlays:

```
sprites.clear(screen, background)
dirty = sprites.draw(screen)
overlays.draw(screen)
pygame.display.update(dirty)
clock.tick(15)
```

Note that instead using `pygame.display.flip()` we switched to `pygame.display.update()`, which takes a list of rectangles to update as a parameter. This way we are only updating the parts of the screen that contain sprites – the only thing that can move.

Your program should now display something like this:



3.2 Animation

That map is getting updated at 15 frames per minute, but we can't really see that because there is nothing moving. Lets fix that and add some animated monsters.

We actually already have almost all that we need – just need to extend the Sprite class a little bit:

```
class Sprite(pygame.sprite.Sprite):
    def __init__(self, pos=(0, 0), frames=None):
        super(Sprite, self).__init__()
        self.frames = frames
        self.animation = self.stand_animation()
        self.image = frames[0][0]
        self.rect = self.image.get_rect()
        self.pos = pos

    def stand_animation(self):
        while True:
            for frame in self.frames[0]:
                self.image = frame
                yield None
                yield None

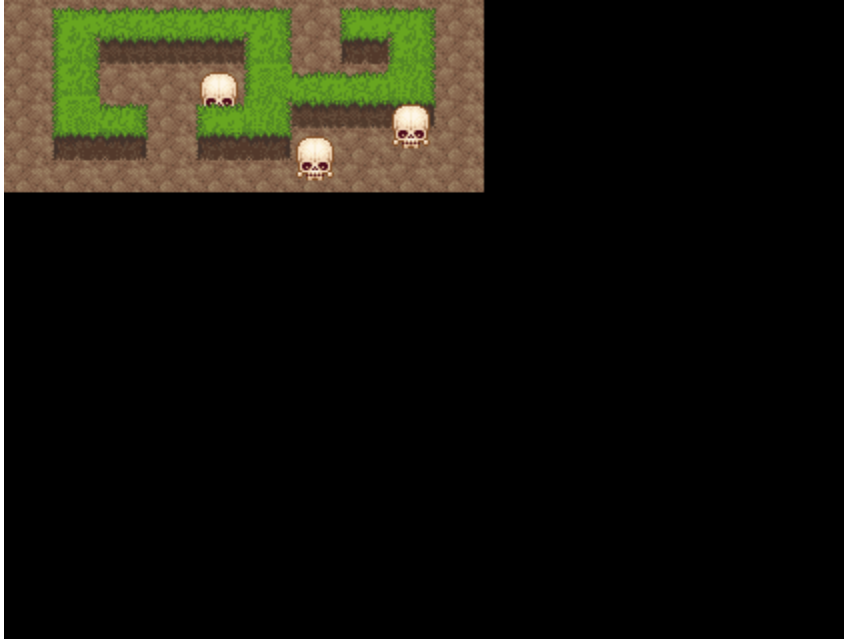
    def update(self, *args):
        self.animation.next()
```

We are using iterators for defining animations – this way we don't need to keep track of the current frame and such – we just iterate over the animation. The default animation just changes the frame to the next one every two ticks – we need to change our crate sprites to “skeleton.png” to see the difference.

Lastly, we need to update the sprites between clearing and drawing them:

```
sprites.clear(screen, background)
sprites.update()
dirty = sprites.draw(screen)
```

If everything went right, you should now see some animated skeletons:



Player character

It's finally time to add a little bit of interactivity to this game. We will introduce a player character that walks around the map. But before we do that, we need to clean up our main loop a little bit, get rid of most of the globals and create a Game object.

4.1 Main loop cleanup

Indices and tables

- `genindex`
- `modindex`
- `search`