Flare-On 11 Challenge 1: frog

By Nick Harbour (@nickharbour)

Overview

This challenge is a PyGame program that comes packaged with a README.txt file for launching instructions. If you are in a Windows environment, the associated .exe file may be executed to launch the game. Other environments may launch the .py file directly, once PyGame is installed.

When you launch the game you are presented with the following game screen, as shown below in Figure 1.

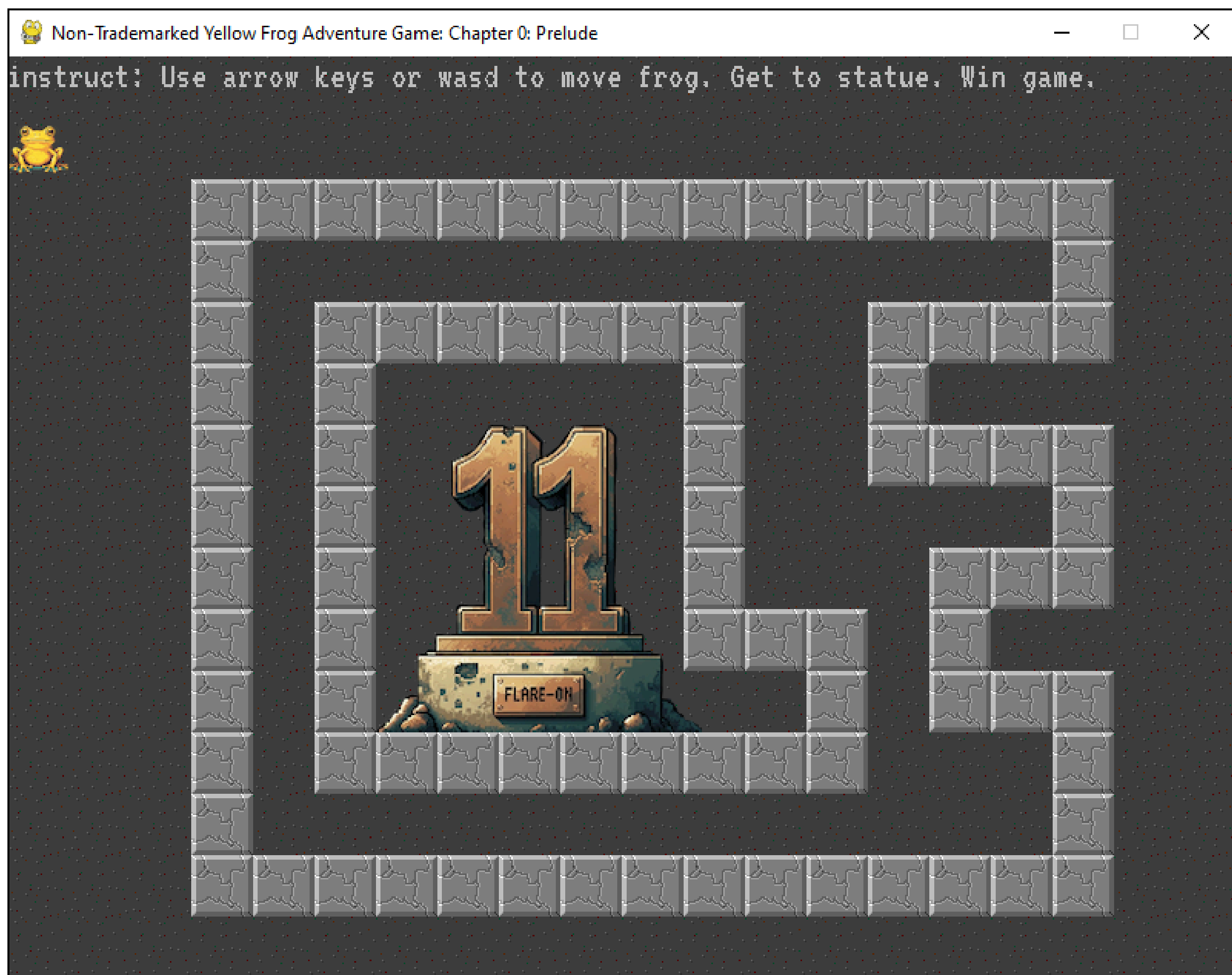

Figure 1: frog game screen

The game provides a single line of help text laconically instructing you to “instruct: Use arrow keys or wasds to move frog. Get to the statue. Win a game.” The yellow frog in the top left is your avatar for this game, and an astute observer will notice that it appears there is no way in through the double walled fortress to reach the statue of the “11”.

At this point, you can play the game and perhaps stumble upon a way in, or try reverse engineering the program by reading its source code: frog.py

frog.py Source code

Inside the frog.py source code you may notice that the game board is hard-coded by initializing an array of Block objects. These represent the individual wall blocks surrounding the statue. The definition of the Block class is short, and provided here in Figure 2 in its entirety

```python
class Block(pygame.sprite.Sprite):
    def __init__(self, x, y, passable):
        super().__init__()
        self.image = blockimage
        self.rect = self.image.get_rect()
        self.x = x
        self.y = y
        self.passable = passable
        self.rect.top = self.y * tile_size
        self.rect.left = self.x * tile_size

    def draw(self, surface):
        surface.blit(self.image, self.rect)
```

Figure 2: Block class definition

Most of the fields in Block are as expected, its location in X and Y coordinates, the image representing it, a rectangle member defining its boundaries. One conspicuous addition is a member variable called “passable”. If you search for other references to this member you will find it used in the function AttemptPlayerMove(), as shown by the excerpt in Figure 3 below.

```python
    # See if it is moving in to a NON-PASSABLE block.  hint hint.
    for block in blocks:
        if newx == block.x and newy == block.y and not block.passable:
            return False
```

Figure 3: Block Passable check

So the AttemptPlayerMove() function will only deny the frog to move to a location if the new coordinates are a block's coordinates and it does not have the passable flag. So Block objects with the passable flag true can be moved into. This is the classic false wall design that old school Final Fantasy players are all too familiar with. The way to get to the statue is to find the false walls and move into them. You could do this via brute force, or via the source code.

The function BuildBlocks() is called at game initialization in the main() function and is responsible for creating the game board. If you refer back to Figure 2, in the class definition of Block it is initialized with three values: X and Y coordinates and the Passable flag. We can determine where the passable blocks are by looking at blocks declared in this function for which the passable flag is set to True.

```python
        ...
        Block(15, 4, True),
        ...
        Block(13, 10, True),
        ...
```

Figure 4: Passable blocks declaration

If you plot these on the game grid it shows the following blocks in red which are defined as passable, in Figure 5 below.

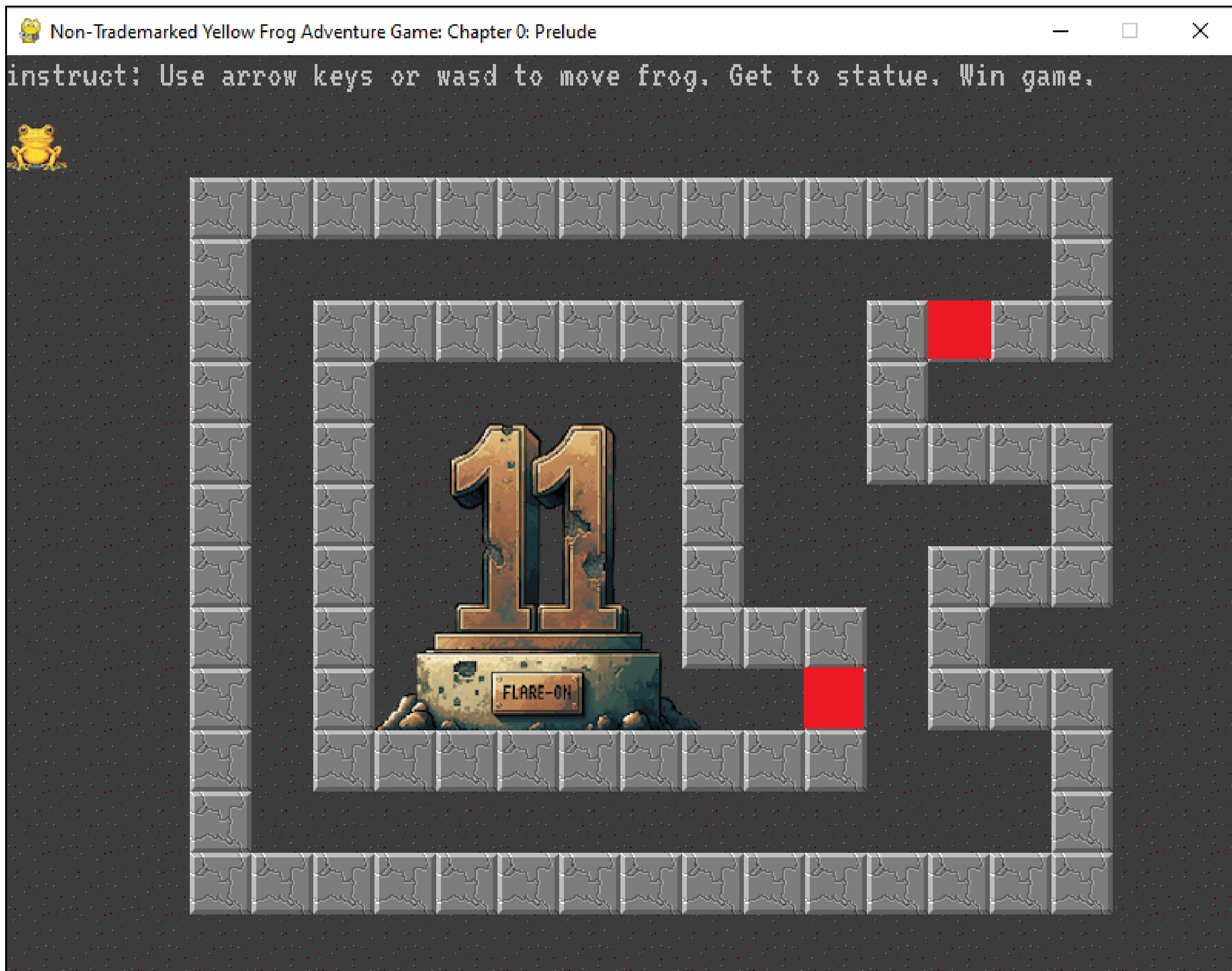

Figure 5: Passable Blocks on the Game Board

Navigating the frog through these passable blocks and onto the base of the statue will cause the victory condition to be achieved.

The victory condition is if the player's coordinates are the same as the victory_tile's location. According to the frog.py source, it is defined as shown in Figure 6 below.

```python
victory_tile = pygame.Vector2(10, 10)
```

Figure 6: Victory Tile Location Declaration

According to the code in the main() function as shown in Figure 7 below, if the player's coordinates match the victory_tile then it calls GenerateFlagText() with the player's location, then renders a "winimage" which is the background for the flag, and then writes the flag overtop of that.

```
Python
        if not victory_mode:
            # are they on the victory tile? if so do victory
            if player.x == victory_tile.x and player.y == victory_tile.y:
                victory_mode = True
                flag_text = GenerateFlagText(player.x, player.y)
                flag_text_surface = flagfont.render(flag_text, False, pygame.Color('black'))
                print("%s" % flag_text)
        else:
            screen.blit(winimage, (150, 50))
            screen.blit(flag_text_surface, (239, 320))
```

Figure 7: Displaying the Win Image and Flag

Since GenerateFlagText() takes the player's location as input, it may not generate the correct flag if you attempt to solve the challenge by forcing the flag to be rendered without the player being in the correct position. Figure 8 below shows the GenerateFlagText() function.

```
Python
def GenerateFlagText(x, y):
    key = x + y*20
    encoded =
"\xa5\xb7\xbe\xb1\xbd\xbf\xb7\x8d\xa6\xbd\x8d\xe3\xe3\x92\xb4\xbe\xb3\xa0\xb7\xff\xbd\xbc\xfc
\xb1\xbd\xbf"
    return ''.join([chr(ord(c) ^ key) for c in encoded])
```

The GenerateFlagText() function performs a simple single-byte XOR of a fixed string with a key byte that is produced by combining the player's location with the formula "X + Y * 20". Since we already know the victory tile location is supposed to be at coordinates 10, 10 according to the code shown in Figure 6, we can simply say that the single byte key is the value 210 decimal (0xD2 in hex).

You can solve this challenge by manually decoding the encoded flag string, or simply getting the frog to location (10,10) by passing through the passable blocks, causing the game to display the victory screen containing the flag as shown in Figure 9 below.

Figure 9: Victory Screen

Final Flag

```
Unset
welcome_to_11@flare-on.com
```