FLARE-On 12 Challenge 1: Drill Baby Drill!

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: Drill Baby Drill! game screen

The game provides a help text instructing you to “Use right/left to move baby and up/down to raise or lower your drill. Find all lost bears. Don’t drill into a rock. Win game.” There is one lost bear per level and a total of five levels. Each level has a name and the order you will face them is randomized. The current level in the screenshot in Figure 1 is “California”. Moving right to left moves the baby along the surface, and the up and down arrows lower your drill. You cannot move laterally while your drill is lowered. Simply drill into the teddy bear to move to the next level.

The challenge here is that the game is fundamentally unfair, in that every column except the one with the bear will have a boulder in it somewhere, and if you hit the boulder you ruin your drill, and thus have to restart, as shown below in Figure 2.

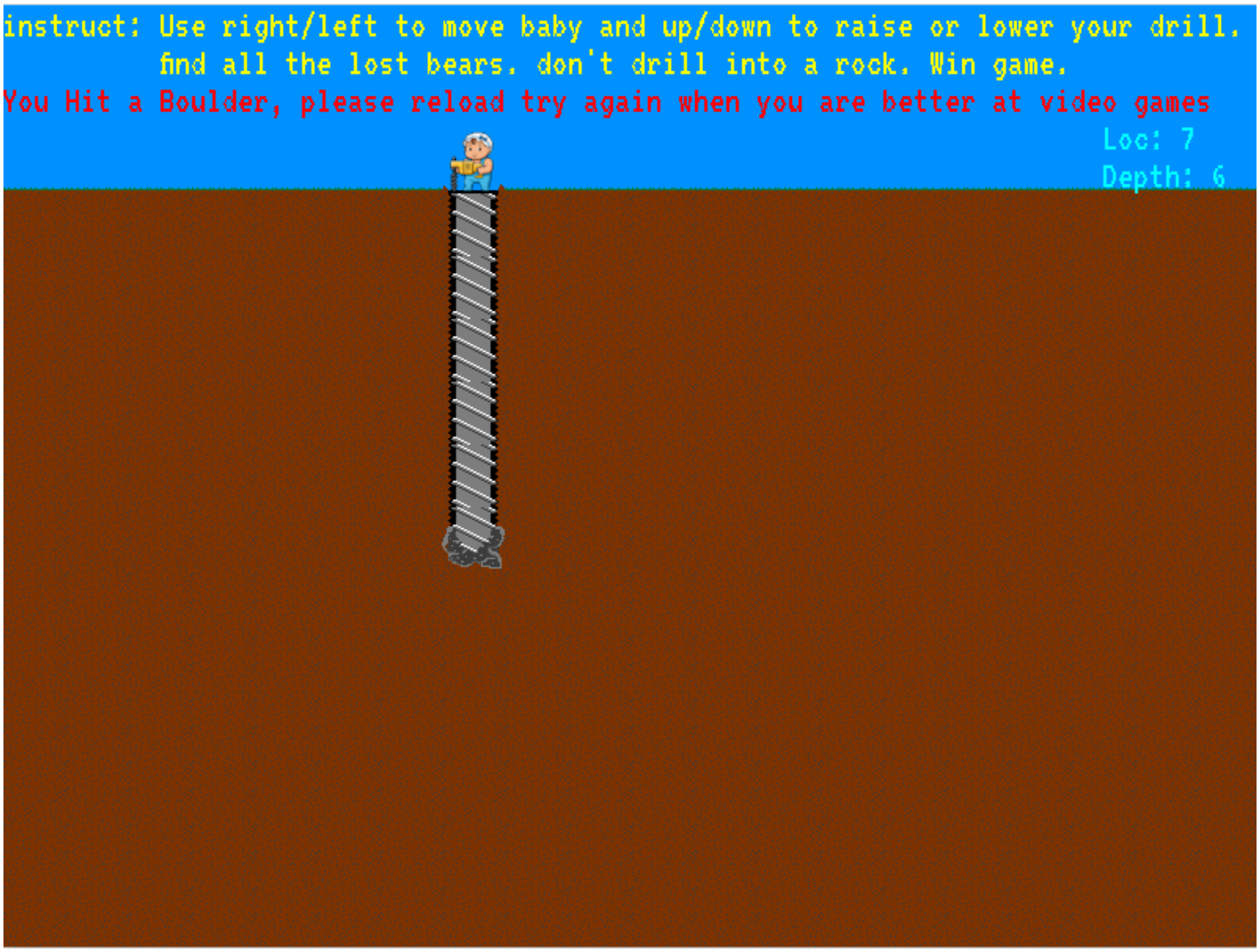

Figure 2: Hitting a Boulder

The only two ways to win the game are to get extremely lucky, or reverse engineer the provided source code (DrillBabyDrill.py) to figure out which column will contain the bear for each level (it’s not random). Please note

that the information displayed in Cyan on the upper right portion of the screen tells you your player's horizontal location as well as your current drill depth. The horizontal location will be important to solving the challenge.

DrillBabyDrill.py Source code

Inside DrillBabyDrill.py's main game loop you may notice two methods which seem to be critical to determining success or failure within the game logic. This is hitBoulder() and hitBear() within the player object class. An example of their usage is shown below in Figure 3 from line 275 in the source.

```python
            if player.hitBoulder():
                 boulder_mode = True

             if player.hitBear():
                 player.drill.retract()
                 bear_sum *= player.x
                 bear_mode = True
```

Figure 3: hitBoulder() and hitBear() usage

Let's now examine the implementation of these functions to identify how they are detecting a boulder or a bear.

```python
    def hitBoulder(self):
         global boulder_layout
         boulder_level = boulder_layout[self.x]
         return boulder_level == self.drill.drill_level

     def hitBear(self):
         return self.drill.drill_level == max_drill_level
```

Figure 4: Implementation of hitBoulder() and hitBear()

It appears from the implementations shown in Figure 4 that the boulder locations are contained in a global array called `boulder_layout`, and that a bear is detected simply if the drill hits the maximum drill level (presumably the bottom of the screen). The index into the boulder_layout array is `self.x`, which represents the players current horizontal position, which is also displayed to them on the screen. Lets now examine how this `boulder_layout` array is populated, as the column containing the bear must be a column containing no boulder, thus allowing the player to drill all the way to the bottom.

```python
        boulder_layout = []
         for i in range(0, tiles_width):
             if (i != len(LevelNames[current_level])):
                 boulder_layout.append(random.randint(2, max_drill_level))
             else:
                 boulder_layout.append(-1)
```

Figure 5: Populating the boulder_layout array

Here in Figure 5, line 212 in the code, we see a for loop which populates the `boulder_layout` array. Each element in this array represents the boulder location for a given horizontal column. The if statement controls whether or not a boulder will be placed at a random depth from 2 to `max_drill_level`, or placed at the unreachable depth of -1. A drilling column with a boulder at -1 would allow the player to drill all the way to the bottom of the screen, thus uncovering the bear.

The only factor that determines if a column receives a drillable boulder or not is the check:

```python
i != len(LevelNames[current_level])
```

Notice that here it is comparing `i`, which should be the column number in the iteration over possible drillable columns, with the string length of the name of the level. Line 53 in the source code defines an array of `LevelNames` for the game, as shown below in Figure 6:

```python
LevelNames = [
    'California',
    'Ohio',
    'Death Valley',
    'Mexico',
    'The Grand Canyon'
]
```

Figure 6: LevelNames Array

Note that each level name has a unique string length. Since the horizontal location is displayed to the player on the game screen, along with the name of the current level, it is possible to know exactly where to place your player on the screen to be able to drill to the bottom and find each bear. For example, if you are on level California, the string length of “California” is 10. So if you move to location 10 and drill down, you will find the bear as shown in Figure 7 below.

Figure 7: Finding a bear

At this point you are prompted to move to another level. Repeat this process four additional times and you will be presented with the victory screen and the flag as shown in Figure 10 at the end of this document. That is all the work you need to put into this challenge to solve it, but to illuminate the secrets of how the key is hidden keep reading.

You may remember seeing the code after the `hitBear()` function detects that a bear was drilled into, from line 278 in the source:

```python
            if player.hitBear():
                 player.drill.retract()
                 bear_sum *= player.x
                 bear_mode = True
```

Here the value `bear_sum` is updated based on the player's column when a bear is detected. The `bear_sum` value starts as 1 and is multiplied each time by the latest bear column number. The order in which the levels are solved will not matter as the same multiplication product will result since the level names always produce consistent lengths. This final `bear_sum` value is used to compute the flag value. The determination of when to produce the flag is based on the number of bears found in comparison to the length of the `LevelNames` array as shown in Figure 8 below.

```python
            if current_level == len(LevelNames) - 1 and not victory_mode:
                victory_mode = True
                flag_text = GenerateFlagText(bear_sum)
                print("Your Flag: " + flag_text)
```

Figure 8: End of game detection and calling the flag generator

The flag text generator takes the bear_sum value as input, which should be the product of the lengths of all the level names. The implementation of this function is shown here in Figure 9 below.

```python
def GenerateFlagText(sum):
    key = sum >> 8
    encoded =
"\xd0\xc7\xdf\xdb\xd4\xd0\xd4\xdc\xe3\xdb\xd1\xcd\x9f\xb5\xa7\xa7\xa0\xac\xa3\xb4\x88\xaf\xa6
\xaa\xbe\xa8\xe3\xa0\xbe\xff\xb1\xbc\xb9"
    plaintext = []
    for i in range(0, len(encoded)):
        plaintext.append(chr(ord(encoded[i]) ^ (key+i)))
    return ''.join(plaintext)
```

Figure 9: GenerateFlagText() function implementation

This function begins by bit shifting the `bear_sum` value 8 bits to the right. Given that the lengths of the level names are 4, 6, 10, 12, and 16, their product should be 46080. This value is B400 in hex, so shifting it to the right 8 bits results in the value B4. This forms the basis of the decryption key. What follows is a loop which XOR's each byte in the encoded data with the key B4 plus the index value. This means each byte gets a different decryption XOR key value in sequence (i.e. B4, B5, B6, ...). Decrypt each byte in the encoded string and you produce the final flag: **`drilling_for_teddies@flare-on.com`**

Figure 10: Victory Screen with flag