HSF8 - A CHIP-8 Emulator

Backstory

A long time ago (sometime last year, though it may have been a bit before that) I wanted to look into how emulators work and make one myself. After doing a bit of digging (by a bit of digging, I mean spending about 5 minutes on DuckDuckGo) I came across CHIP-8. It's often recommended as the easiest to emulate, and as an introduction / 'tutorial' to making emulators. However I didn't know much about how computers worked at the time, and hence when I tried to jump straight in, I took a hard fall and failed hard, without even getting anything running. Fast forward to now, I've learned a lot more about how computers work and programming in general, and my CS teacher posted about an incoming final due in two weeks. Feeling confident-ish with my much gained knowledge since then, I decided to try again and do a second attempt for my final. Instead of jumping straight in, I decided to look at the freecodecamp article on how to make a CHIP-8 emulator (https://www.freecodecamp.org/news/creating-your-very-own-chip-8-emulator/), as well as ttom795's open source Chippure emulator (https://github.com/ttom795/Chippure) (great resources, btw!) before, which definitely helped. I also decided to target and test the IBM Logo ROM first, since I saw that as the easiest to get correct. I then moved on to testing misc stuff, with my primary goal to get Space Invaders to work, which I used test_opcode to help me test what was working. After a while, I finished HSF8 in its current state. So, let's go into how it works! We're going to first be building what I call "JEFIBM" - or 'Just Enough For IBM', which will be able to load the IBM Logo.

Libraries to Import that we're using

```
from tkinter import *
import math
import time
import random
import sys
```

Drawing Pixels

```
#gen screen
cols = 64
rows = 32
scale = 4
screen = Tk()
screen.title('JEFIBM by 0xilis')
screen.geometry(str(cols*scale)+'x'+str(rows*scale))
screen.config(bg='#000000')
```

For drawing pixels, we're using the tkinter library. This code here generates the screen. CHIP-8 games are 64x32, however that's probably too small for us, so we're going to scale it up by 4. This means our actual display will be 256x128. Let's have the background be black, since it's probably what you're expecting for CHIP-8. Now, let's set up drawing pixels.

```
def drawPixel(x, y): #col is x row is y
  canvas.create_rectangle(
    x*scale, y*scale, (x*scale)+scale, (y*scale)+scale,
    fill="white",
    tag="squa"
    )
  canvas.pack()
  screen.update()

def cls():
  canvas.delete('all')
```

We're adapting for scale here. x*scale, would make a pixel at say position 5 for 64x32, be 20 for 258x128, and (x*scale)+scale makes the pixel be 4. This *should* make it easy to adjust the scale if we need later (but we won't). We use screen.update() to force update the display so it doesn't just finish rendering once the program terminates. We're also adding the cls() function that clears the screen, which will be useful later on. Try drawPixel(5,5) to test if rendering works. Well - actually, there's one caveat. This renderer that I made - doesn't actually work properly, and I had to rewrite it later. However it should be fine for IBM, so I'll talk about it later.

Loading ROMs

Ight, let's load ROMs. CHIP-8 is most often implemented on machines with 4K RAM. However, the CHIP-8 interpreter itself occupies the first 512 bytes of the memory space on these machines. Hence, most programs begin at location 512 (0x200).

```
memory = [0x0 for i in range(4096)]
filename = input("Input rom name: ") + ".ch8"
dafile = open(filename, 'rb')
program = dafile.read()
dafile.close()
pc = 0x200
v = [0x0 for i in range(16)] #16 8 bit registers
selfi  = 0 #Stores memory addresses. Set this to 0 since we aren't storing anything at
initialization.
for i in range(len(program)):
  memory[0x200+i] = program[i]
while True:
    daopcode = memory[pc] << 8 | memory[pc+1]
    print(pc) #useful for debugging to see where your program is
    excopcode(daopcode)
```

Okay, a bit complicated, but I'll try to explain the best I can. We're storing the memory all in the memory variable, initially all blank. Ask the user for a ROM, and add .ch8 at the end so we don't have to type .ch8 each time. We read the program into the program variable, then we fill the memory with it, but make it push the ROM to start instead at location 512 (0x200). Let's break down daopcode = memory[pc] << 8 | memory[pc+1]. memory[pc] << 8 gets the first piece of memory and shifts it 8 bits. to find the opcode, and then excopcode(daopcode) to execute the opcode. Now that that's explained, time for the fun part: the actual opcodes.

Opcodes (The Fun Part)

```
def excopcode(opcode):
  global pc
  global v
  global selfi
  pc += 2 #increment program counter so we, the CPU, know where the next
instruction is

  x = (opcode & 0x0F00) >> 8 #x - A 4-bit value, the lower 4 bits of the high byte of the
instruction
  y = (opcode & 0x00F0) >> 4 #y - A 4-bit value, the upper 4 bits of the low byte of the
instruction
  code = opcode & 0xF000 #to check code without nnn
  kk = opcode & 0xFF #kk or byte - An 8-bit value, the lowest 8 bits of the instruction

  print("executing " + str(hex(opcode)))
```

Yayy. So, comments help, but I'm going to explain a bit more. code = opcode & 0xF000 will remove nnn from the opcode, so opcodes like 0x121a will become 0x1000. kk is the lowest byte - ex if your opcode is 0x7004, kk is 0x04. Rest are explained in comments. Let's add our first opcodes! I'd say a good reesource is http://devernay.free.fr/hacks/chip8/C8TECH10.HTM

```
print("executing " + str(hex(opcode)))
if code == 0x0000:
  if opcode == 0x00E0: #cls
    cls()
  elif code == 0x1000: #JP addr
    pc = opcode & 0xFFF
```

These are very important opcodes that nearly every CHIP-8 ROM uses. There are two opcodes with the code 0x0000, and good news to us is that they're static and don't have anything to pass into them, so we can just do opcode == on them. 0x00E0 clears the screen, so we're going to use our cls() function. 0x00EE should probably be implemented later, but since we're only doing IBM for now and the program never uses this opcode we'll just push it off to the side and get to it later. 0x1nnn is the jump opcode. 0xFFF is nnn, ex. 0x1426 & 0xFFF would give us 0x426 for 0xFFF. We're going to store that in our program counter.

```
elif code == 0x6000: #LD Vx, byte
  v[x] = kk

elif code == 0x7000: #ADD Vx, byte
  v[x] += kk

elif code == 0xA000:
  selfi = opcode & 0xFFF
```

The instruction 0x6xkk sets v[x] to the value of kk. The instuction 0x7xkk adds kk to v[x]. 0xAnnn gets nnn and sets selfi to the value of it. v is 16 8-Bit registers, and we're saving to the x register. Pretty self explanitory imo.

```
elif code == 0xD000: #DRW Vx, Vy, nibble
    width = 8
    height = opcode & 0xF
    v[0xF] = 0
    for row in range(height):
      sprite = memory[selfi+row]
      for col in range(width):
        if (sprite & 0x80) > 0:
          if drawPixel(v[x]+col, v[y]+row):
            v[0xF] = 1
        sprite <<= 1
```

We're finally at Dxyn. This one's really important: drawing pixels. Each sprite is 8 pixels wide, so we can hardcode that in. We're getting n for the height, which is opcode & 0xF. Set v[0xF] to 0 for now to signify drawing pixels, to signify erasing we're going to do 1. Now, CHIP-8 sprites look like this:

```
11110000
10000000
10000000
11110000
11110000
```

We're cycling through each row, and sprite = memory[selfi+row] gives us the single row of the sprite. From there check the left bit is greater than 0, if so, set the mode to erase. Now, get the IBM CHIP-8 ROM (you can find it online) and plug it in. Boom! You got IBM running! The source code of what we just built is https://github.com/0xilis/HSF8/blob/main/JEFIBM.py. For JEFIBM, read the docs for the opcodes and implement them - i feel like I explained enough. Though, for future use, he're some big things we're going to run in to:

Keyboard Input

Pretty easy. screen.bind('<Key>',inputEventDown) to run the inputEventDown() function when a key is down, and screen.bind('<KeyRelease>',inputEventUp) to run inputEventUp() when a key is up. For def inputEventDown(event):, you'll have to change the key presses to what CHIP-8 keycodes are, and set it in our keypressed variable. On inputEventUp(event), change keypressed to None. Here's a snippet from HSF8 that should give you the idea of what keycodes to use:

```
keys = {
   "1":0x1,
   "2":0x2,
   "3":0x3,
   "4":0xc,
   "q":0x4,
   "w":0x5,
   "e":0x6,
   "r":0xD,
   "a":0x7,
   "s":0x8,
   "d":0x9,
   "f":0xE,
   "z":0xA,
   "x":0x0,
   "c":0xB,
   "v":0xF
 }
```

What's wrong with our renderer?

Hey, what's wrong with our renderer, anyway? Well, if you didn't notice - in the example I showed for the instruction that draws sprites (Dxyn), it sets a mode to 1 if we should erase a sprite. But did we take this into account when we made our renderer? No! So we have to rewrite the entire renderer… how fun… ugh. Don't worry, if you have been following this so far, you should have a pretty good idea for what to do in the rewrite. I should note that if a pixel is attempted to draw out of bounds, it wraps it around to the display, so I would recommend taking that into account (although to be honest I'm not really sure if any games actually rely on this…).

Final Thoughts

Holy cow, that was awesome! I've always wanted to try making an emulator, and well, now I guess I sort of have. Will I be attempting one again? I don't know. But I gotta say, I love the result. I half want to do a try for NES / Gameboy. You can find HSF8 on GitHub here: https://github.com/0xilis/HSF8/blob/main/emu.py