**Modifications dans l’émulateur**

A la base, l’émulateur était conçu en python pour fonctionner avec la librairie Pygame sur pc, il a donc fallu adapter le programme pour notre plateforme.

Ajout d’importations :

from machine import Pin, SPI

from ssd1309 import Display

from utime import ticks\_ms

from buttons import lecture\_touche

Ces librairies permettront de gérer l’écran et les timers.

Suppression des lignes 123 et 124 :

pygame.init()

pygame.time.set\_timer(pygame.USEREVENT+1, int(1000 / 60))

La deuxième ligne est un timer de 16.67ms qui était utilisé pour simuler le timer du CHIP 8

Dictionnaire des entrées ligne 129 :

        self.keyDict = {

            49 : 1,

            50 : 2,

            51 : 3,

…….

            99 : 0xb,

            118 : 0xf

        }

Les codes à gauche correspondent aux codes ASCII renvoyés par Pygame et les codes à droite aux codes hexadécimaux des touches du CHIP 8.

Taille des pixels ligne 158 :

self.size = 10

Cette valeur correspond à la taille des pixels dessinés par les programmes, ici comme nous utilisons un écran de 128\*64 pour une résolution de 64\*32 du CHIP 8 nous mettons cette constante à 2. Ce qui donnera des carrés de 2 pixels de côté.

Initialisation de l’écran lignes 161-163 :

self.screen = pygame.display.set\_mode([width \* self.size, height \* self.size])

self.screen.fill(self.oneColor)

pygame.display.flip()

On a supprimé la première ligne puis remplacé les deux dernières par :

display.clear()

display.present()

qui sont des instructions similaires mais issues de la librairie ssd1309

Modification de la gestion des touches et du timer lignes 599 – 618 :

for event in pygame.event.get():

            if event.type == pygame.QUIT:

                sys.exit()

Si le bouton de fermeture de la fenêtre est cliqué alors quitter le programme, on supprimera ces deux lignes

            elif event.type == pygame.USEREVENT+1

                self.delayTimer.countDown()

Utilisation du timer de 16.67ms pour décrémenter le timer en CHIP 8.

            elif event.type == pygame.KEYDOWN:

                try:

                    targetKey = self.keyDict[event.key]

                    self.keys[targetKey] = True

                except: pass

Contrôle des touches pressées.

            elif event.type == pygame.KEYUP:

                try:

                    targetKey = self.keyDict[event.key]

                    self.keys[targetKey] = False

                except: pass

Contrôle des touches relachées.

Tout d’abord on vient rajouter ligne 125 :

self.lastime = ticks\_ms()

self.lastKey = '0'

ces variables nous serons utile plus tard

if ticks\_ms() - self.lastime >= (1000/60):

self.lastime = ticks\_ms()

self.delayTimer.countDown()

Vient remplacer l’ancienne fonction permettant de gérer le timer.

Pour la gestion des touches, on va réutiliser la librairie buttons, plus précisément sa fonction « lecture\_touche() »

self.pin\_value = lecture\_touche()

On commence par récupérer la touche pressée.

if self.lastKey != self.pin\_value:

targetKey = self.keyDict[self.lastKey]

self.keys[targetKey] = False

Si la dernière touche pressée est différente de la touche actuelle, alors on met la dernière touche à 0.

if self.pin\_value != None:

targetKey = self.keyDict[self.pin\_value]

self.keys[targetKey] = True

self.lastKey = self.pin\_value

Si la touche actuelle est différente de None alors on met l’état de celle-ci à 1, et elle devient la dernière touche pressée.

Suppression de la fonction d’affichage lignes 630 - 640:

Voici la fonction d’affichage prévue pour Pygame

    def display(self):

        for i in range(0, len(self.grid)):

            for j in range(0, len(self.grid[0])):

                cellColor = self.zeroColor

                if self.grid[i][j] == 1:

                    cellColor = self.oneColor

\*               pygame.draw.rect(self.screen, cellColor, [j \* self.size, i \* self.size, self.size, self.size], 0)

        pygame.display.flip()

Concrètement self.grid est une matrice de 64\*32 constituée de 0 et de 1 représentant l’état de chaque pixel, la fonction display() lit tous les éléments de la matrice et dessine chaque pixel avec l’état correspondant. On pourrait modifier cette fonction en remplaçant la ligne marquée d’in astérisque par :

display.fill\_rectangle( j \* self.size, i \* self.size, self.size, self.size, invert=True/False)

Mais après avoir testé nous nous sommes rendu compte que cette méthode était beaucoup trop lente car elle impliquait de redessiner l’entièreté de l’écran à chaque frame ce qui prenait +/- 4 secondes.

Nous sommes donc remontés ligne 541 où il y avait la fonction permettant de mettre à jour la matrice « self.grid » lorsqu’un sprite était dessiné à l’écran.

for i in range(len(spriteBits)):

for j in range(8):

try:

inversion = False

if self.grid[Vy + i][Vx + j] == 1 and int(spriteBits[i][j]) == 1:

collision = True

inversion = True

if int(spriteBits[i][j]) == 1:

display.fill\_rectangle((Vx + j) \* self.size, (Vy + i) \* self.size, self.size, self.size, invert=inversion)

self.grid[Vy + i][Vx + j] = self.grid[Vy + i][Vx + j] ^ int(spriteBits[i][j])

except:

continue

display.present()

return collision

Nous avons rajouté les lignes en vert qui servent à dessiner les pixels du sprite sous forme de carré de 2px de côté. Sans oublier le display.present() qui permet d’envoyer le buffer à l’écran et donc d’afficher les modifications.

La fonction clear lignes 557-560 a elle aussi été modifiée :

    def clear(self):

display.clear()

display.present()

        for i in range(len(self.grid)):

            for j in range(len(self.grid[0])):

                self.grid[i][j] = 0

On a simplement rajouté la fonction display.clear()

Modification du mainLoop() lignes 620-628 :

def mainLoop(self):

        clock = pygame.time.Clock()

        while True:

            clock.tick(300)

            self.keyHandler()

            self.soundTimer.beep()

            self.execution()

            self.display()

La fonction clock.tick() est une fille de pygame.time.Clock(), clock.tick(300) est une instruction permettant de limiter le framerate à 300 fps, nous ne l’utiliserons pas et elle peut donc être supprimée. De même pour « self.display() » qui peut être supprimée étant donné que la fonction self.display() à été supprimée précédemment.

Lancement du jeu ligne 643 :

chip8.readProg(sys.argv[1])

A la base, l’émulateur est censé se lancer par ligne de commande et l’utilisateur doit entrer le nom de la rom après celui du programme. Ici, pour l’instant nous allons simplement remplacer cette ligne par

chip8.readProg("games/Tetris.ch8")

Où “games” est le nom du dossier sur le Raspberry contenant les jeux et « Tetris » peut être remplacé par le nom du jeu auquel on souhaite jouer.