Projet Python

Introduction

La pratique de l'hypnose se répand de plus en plus et l'auto-hypnose gagne ainsi en popularité. De nombreuses méthodes existent, allant des plus basiques – suivre une suite d'instructions donnée à priori – aux plus sophistiquées – e.g l'hypnotiseur expérimenté maîtrisant en temps réel les processus en jeux.
D'une façon générale, ce sont les néophytes qui utilisent les méthodes les plus basiques et les moins subtiles. Ces méthodes ne présentent aucune adaptation aux rythmes et aux capacités inhérentes à chaque individu : la suite d'instructions reste la même quelque soit la personne l'employant. Ainsi, ces méthodes fonctionnent mal et leurs utilisateurs peinent à progresser, perdus qu'ils sont dans la diversité de leurs réactions. Loin de vouloir concurrencer la présence réactive et bienveillante d'un hypnotiseur talentueux, nous souhaitons proposer une assistance à l'auto-hypnose. Cette assistance prend la forme d'un programme informatique distribuant des instructions et calibrant les réactions de l'utilisateur.
Ce genre de projet étant inédit, il nous pousse à nous poser de nombreuses questions. Un programme étant plus rigide qu'un être humain, quelles suites possibles de phénomènes hypnotiques sont le plus adaptées pour entraîner un sujet débutant à développer des phénomènes hypnotiques complexes ? Quels critères utiliser pour déterminer si le sujet réussi/échoue à développer le phénomène hypnotique proposé ? Dans quelle mesure les difficultés techniques rencontrées lors de la conception de ce programme vont-elles limiter les possibilités en matière d'hypnose ?
Si toutes ces questions constituent la motivation principale pour ce projet, le mémoire ne portera pas sur les aspects hypnotiques mais plutôt sur les piliers techniques sur lesquels le programme s'appuie. Ainsi, nous commencerons par présenter l'idée initiale, la forme idéale que devra finalement revêtir le programme. Nous traiterons par la suite l'interface utilisateur, pour enchaîner sur l'utilisation de la voix synthétique. Puis nous aborderons l'une de parties majeures de ce projet : la vision par ordinateur. Enfin, nous discuterons de la façon dont toutes ces composantes sont organisées entre elles.

Idée initiale

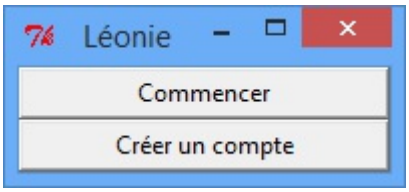
Comme nous l'avons expliqué en introduction, le programme développé doit être en mesure de donner des instructions menant au développement de réactions de plus en plus automatiques, tout cela en fonction du rythme et des capacités du sujet. Il ne doit donc ni donner ses instructions trop tôt, ni les donner trop tard. Dans l'idéal il doit également être en mesure de sélectionner, à l'étape n+1, l'instruction la plus appropriée selon ce qu'il a perçu des réactions du sujet jusqu'à maintenant.
Ainsi, le programme a besoin de feedback pour dérouler les instructions de façon optimale. Ce feedback prend deux formes : la vision par ordinateur, via la Kinect de Microsoft et un système de biofeedback, donnant le rythme cardiaque, le degré de transpiration, le rythme de la respiration etc.. du sujet. Le temps dont nous disposions pour rendre ce projet étant limité, nous n'avons pas abordé la partie biofeedback. De plus, les instructions seront données oralement, l'effort de lecture gênant bien souvent le développement de l'état d'hypnose s'il est mal amené.Voici donc les piliers sur lesquels repose le projet :

- La vision par ordinateur
- Le biofeedback
- L'interface utilisateur
- La voix artificielle
- Le cœur du système

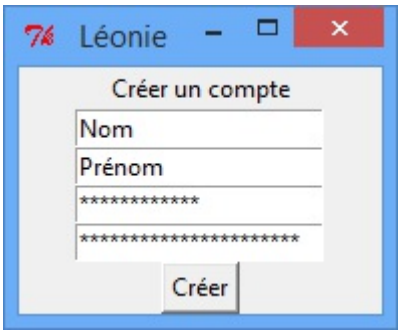
Nous allons commencer par détailler l'interface utilisateur.

Interface utilisateur

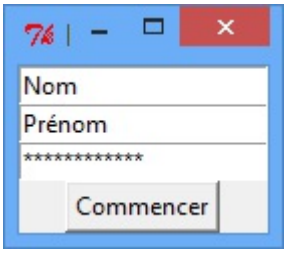
Les fonctionnalités proposées sont relativement simple. L'objectif de cette interface est de garder une trace des utilisateurs du programme, de cette façon lorsqu'ils se reconnectent au programme ce dernier connaît déjà leur vitesse de réaction et jusqu'où ils sont allés précédemment. Ceci évitant de recommencer les séances au tout début et de proposer un contenu destiné aux débutants à un sujet plus expérimenté.
Nous avons utilisé la librairie python Tkinter pour réaliser cette interface graphique, somme toute très simple. Lorsque l'utilisateur lance le programme, une fenêtre s'affiche, lui proposant deux choix : l'inscription, s'il n'a encore jamais utilisé le programme, ou alors simplement de commencer la séance s'il a déjà un compte :



Si l'utilisateur souhaite créer un compte, il doit rentrer plusieurs informations, comme son nom, son prénom, un mot de passe, une vérification de mot de passe – un message d'erreur s'affiche si les deux ne matchent pas. Si le compte est créé avec succès, l'utilisateur est ramené à la fenêtre de log après qu'un message indiquant que tout s'est bien déroulé se soit affiché.



Si l'utilisateur est déjà passé par le programme, il peut commencer directement. S'affiche alors une fenêtre de log, où il doit entrer son nom, son prénom ainsi que son mot de passe. Si ces informations correspondent à un compte déjà créé, le programme se lance.



Comme nous pouvons le voir, peut de temps a été passé sur le design de l'interface graphique. Par manque d'intérêt d'abord, la motivation du projet n'étant pas là. Par faute de goût suffisant pour trouver de bons design ensuite.
Cette partie n'a pas posé de problèmes techniques : Nous avons défini un object Cadre, contenant toutes les fenêtres montrées et permettant de faire le lien entre elles. Un objet a ensuite été créé par fenêtre : FenetreStart pour la fenêtre de log, Inscription pour la fenêtre de création de compte, et Init pour la fenêtre initiale.
De plus, lorsqu'un utilisateur crée un compte, un objet User est créé, contenant son nom, son prénom ainsi que son mot de passe. Cet objet est stocké dans un fichier – un fichier par utilisateur. Le mot de passe est évidemment crypté via le code :

```
password = (hashlib.sha1(self.password1.get().encode(encoding = "UTF-8"))).hexdigest()
```

La fonction `encode()` permet de gérer les accents grâce au paramètre `encoding = "UTF-8"` . La fonction `hashlib.sha1()` est ensuite appelée pour passer le mot de passer par l'algorithme de hashage sha1. Enfin, nous utilisons la fonction `hexdigest()` qui transforme l'encodage en chaîne de caractères.

Abordons maintenant le sujet de la voix artificielle.

Voix synthétique

Nous avons eu beaucoup de problèmes avec la voix artificielle. Nous avons d'abord essayé d'installer pyTTSx mais sans succès. Nous avons alors tenté d'autres packages, notamment l'API de google permettant d'utiliser leur voix artificielle. Mais celui-ci nous posait un problème : une connexion internet était requise pour le faire fonctionner, alors que notre programme est destiné à fonctionner sur un ordinateur sans connexion. De plus, les phrases étaient lues une fois sur deux, la moitié du temps l'API faisaient planter le programme. Nous ne sommes pas parvenus à en trouver la raison. Finalement, nous avons décidé d'utiliser le programme « espeak ». Nous créions alors un fichier wav de la voix artificielle prononçant le discours voulu et prévu à l'avance. Nous stockions ensuite ce fichier dans un dossier data/discours. Ces fichiers étaient ensuite chargés par le programme aux moments convenus et lancés via le lecteur média player. Cette approche posait plusieurs problèmes :

- La voix artificielle était trop peu naturelle et donc inexploitable en situation « réelle ».
- Il fallait lancer le lecteur média à chaque fois. Une fenêtre « parasite » s'affichait donc.
- La procédure de création des discours était lourde. En effet, il fallait d'abord écrire le discours, le copier ensuite dans l'interface espeak, générer le fichier wav, le récupérer et le mettre dans le dossier adéquat. Cette est acceptable dans le cadre restreint du projet pour le cours mais devient vite ingérable s'il prend de l'ampleur.

Finalement, nous avons trouvé une solution. Jusqu'à maintenant nous étions sous Python 3.5. Des problèmes avec la Kinect de Microsoft – que nous détaillerons plus loin – nous ont forcé à passer sous Python 2.7. Il s'est avéré – après quelques jours supplémentaires d'errance sur internet - qu'avec cette version de python, il « suffisait » de copier deux fichier DLL obtenus en téléchargeant le package pyTTSx dans un autre dossier des packages python.
Ceci étant fait, tout est devenu beaucoup plus facile. Le package pyTTSx propose une voix relativement réaliste tout à fait acceptable pour ce projet. La voix artificielle est utilisée pour donner des instructions selon les phénomènes hypnotiques visés. Nous allons présenter la structure des phénomènes.

Nous définissons d'abord une classe appelée Phenomene, contenant deux méthodes :

- La première sert à lancer les instructions relatives au phénomène, donc à utiliser pyTTSx pour lire le texte pertinent pour ce phénomène.
- La seconde sert à générer les encouragements une fois les instructions finies.

Nous allons présenter succinctement ces deux méthodes.

La première méthode est la suivante :

```
def launch(self):
      engine = pyttsx.init()
      engine.setProperty("rate", 150)
      engine.say(self.discours)
      engine.runAndWait()
```

Cette méthode ne prend donc aucun paramètres, elle se contente de prononcer la variable discours, de la même classe, en utilisant pyTTSx.
Nous attirons l'attention du lecteur sur le fait que nous n'utiliserons jamais pyTTSx de façon plus poussée. Le code étant alors toujours le même :

- Nous commençons par créer un objet pyttsx, c'est à dire une voix.
- Nous établissons son rythme : 150 nous paraît parfait, ni trop rapide ni trop lent.
- Nous lui donnons l'instruction de prononcer une chaîne de caractères.
- Nous utilisons la méthode runAndWait().

Désormais nous n'expliquerons plus les codes liés à pyTTSx, ceux-ci étant toujours les mêmes. La seconde méthode est la suivante :

```
def encouragement(self):
      index = randint(0, (len(self.GeneralEncouragements)-1))
      engine = pyttsx.init()
      engine.setProperty("rate", 150)
      engine.say(self.GeneralEncouragements[index])
      engine.runAndWait()
```

Cette méthode a pour rôle de prononcer un encouragement choisi aléatoirement parmi une liste définie à priori dans la variable self.GeneralEncouragements de la même classe. A l'heure où nous écrivons ce rapport, la variable citée est définie comme suit :

```
    self.GeneralEncouragements = ["Très bien", " De plus en plus", "Exactement comme ca", "Tout
a fait, sans aucun effort conscient"]
```

A chaque phénomène que nous intégrons dans le programme correspond alors une classe héritant de la classe Phenomene. De cette façon, pour chaque phénomène particulier, nous n'avons plus qu'à définir les instructions pertinentes qui seront données par le programme. Détaillons à présent la technologie utilisée pour la vision par ordinateur.

Vision par ordinateur

Comme expliqué en introduction, la vision par ordinateur repose sur la caméra Kinect de Microsoft. Il n'existe malheureusement pas de câble officiel pour relier cette caméra à l'ordinateur. Il nous a donc fallu en commander via Amazon en provenance de... Chine. Après un très long délai – au bout duquel nous n'espérions voir ce câble arriver un jour – d'un mois, nous l'avons enfin reçu.
Nous avons alors tenté de faire fonctionner la Kinect avec l'ordinateur. Il existe de multiples SDK et drivers et nous en avons essayé une grande combinaison. Nous sommes passés de python 3.5 à python 2.7 pour faciliter les choses, mais en vain. La seule chose que l'ordinateur détecte chez la Kinect est son entrée audio. Pourtant la Kinect s'allume lorsqu'on la connecte et l'installation de certains drivers se fait automatiquement. Nous avons dépensé beaucoup de temps sur internet, à la recherche d'une solution, sans succès. L'absence de documentation détaillée et d'aide concernant l'utilisation de la Kinect pour python – dont les principaux packages sont pyKinect, pyKinect2 et OpenKinect – nous a empêché de régler les problèmes rencontrés. Ce contretemps s'est révélé assez décourageant. Le choix de la Kinect ne s'était pas fait au hasard. Il correspondait à deux avantages de cette technologie :

- De nombreuses fonctions sont déjà implémentées.
- La Kinect est une caméra profondeur.

Le fait que la Kinect détecte la profondeur de l'image était particulièrement intéressant. En effet, cela permet un skeleton tracking bien plus précis qu'avec une caméra normale. Cela aurai réglé quelques problèmes que nous avons rencontré en faisant de la vision par ordinateur plus classique. Nous les détaillerons plus loin. Par ailleurs, les mouvements automatiques impliqués dans l'émergence d'un phénomène hypnotique sont souvent plus lents et plus subtils que les mouvements habituels. La Kinect avait donc, de par sa précision, nettement l'avantage pour les détecter. Finalement, ne trouvant aucune solution pour faire marcher la caméra de Microsoft, nous nous sommes tournés vers le package cv2, version python d'OpenCV. Nous utilisons alors simplement la webcam de notre ordinateur. Cette approche n'étant clairement pas suffisamment puissant pour la tâche que nous souhaitons effectuer, nous avons programmé un ersatz de détection de phénomène hypnotique : lorsque l'ordinateur donne des instructions, il nous suffit de passer la main dans un carré à gauche de l'écran, puis dans un carré à droite. L'ordinateur estime alors que les instructions ont été réussies et passe à la suite.
Si cela n'a rien à voir avec le projet initial, c'était l'occasion pour nous de pratiquer un peu de vision par ordinateur malgré notre déception suite à l'échec de la Kinect. Nous avons détecté la position de la main en utilisant un histogramme de couleurs calibré préalablement sur notre main. Nous allons d'abord détailler l'algorithme, puis nous soulignerons ses nombreuses limites. Nous avons créé une classe Vision dont le rôle est de gérer l'affichage du retour webcam ainsi que le traitement de l'image pour le tracking de la main. Cette classe a une méthode pour lancer tout ce processus, nommée run – nous expliquerons pourquoi ce nom plus tard.

```
def run(self):
#Cette méthode lance le traitement de l'image.

#Ici on défini les coins des rectangles qui vont nous servir.
    a11 = 100
    a12 = 300
    a21 = 0
    a22 = 128


    b11 = 600
    b12 = 300
    b21 = 500
    b22 = 128
```

```
#On crée les deux variables qui vont nous servir à savor si l'utilisateur a passé la main dans
le rectangle de gauche puis dans celui de droite.
    running1 = True
    running2 = False

#On tente d'obtenir une image via la webcam.
    cv2.namedWindow("preview")
    vc = cv2.VideoCapture(0)
    if vc.isOpened():
        rval, frame = vc.read()
    else:
        rval = False


    while rval:

        c11 = 150
        c12 = 150
        c21 = 300
        c22 = 300

        cv2.rectangle(frame,(c11,c12),(c21,c22),(0, 255, 0),3)
        hand = self.set_hand_hist(frame)
        cv2.imshow("preview", frame)
        rval, frame = vc.read()
        key = cv2.waitKey(20)

        if key == 27:
            self.calibre = True
            break



#Si on a réussi à obtenir une image et tant qu'on réussi à en obtenir.
    while rval:


# Si l'utilisateur est parvenu au bout des instructions, on quitte la boucle: le process de
traitement d'image s'arrête puisque la séance est terminée.
        if self.over:
            break



#Ici on défini les couleurs des rectangles: vert si l'utilisateur n'a pas passé sa main dedans,
rouge s'il l'a fait.
#Le rectangle de droite ne devient rouge que si le rectange de gauche l'est avant lui, comme
expliqué.
        col1 = (0, 255, 0)
        if not running1:
                col1 = (0, 0, 255)


        col2 = (0, 255, 0)
        if running2:
                col2 = (0, 0, 255)


    #On détecte la couleur de peau sur l'image.
        res = self.apply_hist_mask(frame, hand)
    #On obtient le centre de gravité de la main.
        point = self.draw_final(res)
```

```
    #Si on a bien obtenu le centre, on dessine un point noir sur l'écran.
    #Si ce point passe dans le rectangle de gauche, on met running1 à False.
    #Si suite au rectangle de gauche il passe dans celui de droite, on met running2 à True.
        if point is not None:
            cv2.circle(frame,point, 10, (0,0,0), -1)

            if point[0] < a11 and point[0] > a21 and point[1] < a12 and point[1] > a22:
                running1 = False

            if point[0] < b11 and point[0] > b21 and point[1] < b12 and point[1] > b22 and not
running1:
                running2 = True



        #Finalement, si running1 est False et running2 est True, cela signifie que le programme
doit passer à l'instruction suivante.
        #On met alors state à True.
            if not running1 and running2:
                self.state = True
                running1 = True
                running2 = False


        #On dessine les rectangles.
        cv2.rectangle(frame,(a21,a22),(a11,a12),col1,3)
        cv2.rectangle(frame,(b21,b22),(b11,b12),col2,3)

        #On renvoie l'image.
        cv2.imshow("preview", frame)

        #On tente d'obtenir une nouvelle image.
        rval, frame = vc.read()
        key = cv2.waitKey(20)
        #Si on tape sur la touche échape, alors le programme s'arrête: aussi bien les
instructions que le traitement de l'image.
        if key == 27: # exit on ESC
            self.over = True
            self.state = True
            break



    #Finalement, lorsque la séance est terminée on détruit la fenêtre.
    cv2.destroyWindow("preview")
```

Nous commençons donc par ouvrir une fenêtre et essayer d'obtenir une image via la webcam. Puis nous chargeons l'histogramme de couleur de peau calibré au préalable dans la variable « hand ». Si nous avions obtenu les données de la webcam à l'étape précédente nous commençons la boucle de traitement d'image. Nous commençons par dessiner des rectangles de pat et d'autres de l'écran de la couleur appropriée. Nous passons ensuite l'image à deux fonctions de traitement que nous détaillerons plus bas. Enfin, si l'utilisateur passe la main dans le rectangle de gauche, puis dans le rectangle de droite les variables running1 et running2 sont modifiées pour permettre au programme de passer au phénomène hypnotique suivant. Nous dessinons ensuite les rectangles et définissons les actions à suivre si l'utilisateur appuie sur la touche échap. Expliquons maintenant les étapes du processus de tracking de la main. Nous exploitons d'abord la méthode apply_hist_mask, prenant en paramètre l'image ainsi que l'histogramme de couleur de peau calibré auparavant.

- Cette méthode commence par convertir la matrice RGB de l'image obtenue en HSV : Hue Saturation Value – Teinte Saturation Valeur en français.
- On calcule ensuite la probabilité qu'un pixel appartienne
- On crée ensuite un noyau « elliptique » de 11 pixels par 11.

- On vient convoluer l'image HSV avec le noyau elliptique créé précédemment.
- Nous appliquons ensuite un seuil binaire à l'image, avec un seuil automatiquement choisi.
- Nous remettons l'image en noir et blanc.
- Nous appliquons ensuite un filtre gaussien
- Finalement, dans l'image initiale nous remplaçons les pixels où la chair n'est pas détectée par des pixels noirs.

Enfin, cette méthode nous renvoie une image constituée exclusivement des zones détectées comme de la peau. Tout le reste étant noir.

Nous passons maintenant cette image par la méthode `draw_final`.

- D'abord nous détectons les contours de la main.
- Puis nous calculons l'enveloppe convexe de ce contour.
- Enfin, nous trouvons le centre de ce contour et nous le renvoyons.

Cette approche de hand tracking basée sur la couleur de peau n'est pas sans poser quelques problèmes. Tout d'abord, le visage peut être confondu avec la main, puisqu'il n'y aucune notion de profondeur. Ceci peut rendre le contrôle du mouvement de la main difficile. La seconde difficulté réside dans la détection de la peau tout court. En effet, l'histogramme de la couleur de peau étant calibré à l'avance, les conditions de luminosité ne sont pas les mêmes. Ainsi par exemple, lorsque nous avons créé cet histogramme un soir, dans des conditions bien précise, le programme avait le plus grand mal à détecter la couleur de notre peau dans la journée. Et inversement.
Nous avons détaillé les différentes parties du programmes. Il est maintenant temps d'expliquer comment nous les avons rassemblées.

Organisation

Nous avons été confronté à un problème : l'ordinateur s'est avéré incapable de faire tourner la reconnaissance par ordinateur tout en donnant les instructions en parallèle. Il nous a donc fallu remanier la structure du code afin de faire du multi-threading. Ainsi, dans le fichier thread.py, nous déclarons deux classes, Vision et Voice, toutes deux héritant de la classe Thread.
Ceux deux classes correspondent au deux threads parallèles que nous souhaitons lancer : le traitement de la vidéo que nous avons décrit plus haut et la gestion des instructions. De plus, si ces deux threads évoluent en parallèle, ils doivent malgré tout communiquer. En etfet, lorsque l'utilisateur répond positivement à une instruction, le programme doit passer à la suivante. De la même façon, lorsque l'utilisateur arrive au bout des instructions, l'écran de la webcam doit se couper. Pour ce faire, nous avons défini deux variable dans la classe Vision : over, qui vaut FALSE tant que l'utilisateur n'est pas arrivé au bout des instructions de la séance et state qui vaut FALSE tant que l'utilisateur n'a pas réussi à suivre les instructions données. Ainsi, un objet Voice vérifie, à la fin de l'instructions, si state vaut True. Si c'est le cas, il passe à l'instruction suivante. Si ce n'est pas le cas, il donne des encouragements à intervalles aléatoires. Après chaque encouragement, il vérifie si l'utilisateur a réussi a suivre l'instruction.
Dans le fichier Phenomenon.py sont définies les classes correspondant aux phénomènes, dans GUI.py sont définies toutes les classes nécessaires à l'affichage des fenêtres, dans Seance.py est définie la classe Seance qui lance les deux threads parallèles. Enfin, le fichier image_analysis.py contient des fonctions utiles aux traitement des images de la classe Vision.

Conclusion

Nous avons rencontré de nombreux problèmes au cours de la réalisation de ce projet. D'abord des problèmes techniques, liés au technologies utilisées. Par exemple la Kinect, ou encore la voix artificielle. Il nous a également fallu reconsidérer la structure du code : au début, les portions concernant la vision par ordinateur et la voix artificielle étaient imbriquées. Comme nous sommes finalement passés par pyTTSx, nous avons dû créer deux threads différents pour chacun de ces processus tout en les faisant communiquer entre eux. Finalement, du fait de ces nombreux problèmes pas toujours intéressants, ce projet s'est révélé frustrant dans son déroulé. Malgré tout, nous avons eu l'occasion de toucher un peu à toutes les composantes que nous avions prévues, particulièrement la vision par ordinateur. Finalement ce projet est loin d'être arrivé à son terme. Il nous reste essentiellement à bien développer cette partie vision par ordinateur – dans l'idéal réussir à faire fonctionner la Kinect avec python. Sans cela, il est inutile de continuer le

développement du programme. Le travail restant à faire n'est pas tant technique que conceptuel : il consiste essentiellement à apporter une réponse satisfaisante aux questions posées en introduction.