INFO 501

TP2 : Représentation visuelle d'informations

Informations préalables à lire attentivement :

Le sujet proposé dans la suite a été **préparé durant la 7ème et dernière séance de TD** (question Q1).

Compte tenu de son ampleur, il peut être vu comme un mini-projet plutôt que comme un TP.

Un dépôt sur l'EAD est prévu à chaque séance de TP dans la zone de dépôt prévue à cet effet.

Ce dépôt sera constitué des fichiers python sur lesquels vous aurez travaillé pendant la séance correspondante. Par exemple, si dans la 2ème séance de TP vous avez travaillé sur les fichiers « Q2Emoticon.py », « Q2GeneralConfiguration.py », « Q2Main.py », « Q3Button.py », « Q3GeneralConfiguration.py », « Q3Main.py », vous déposerez ces 6 fichiers sur l'EAD à la fin de la 2ème séance.

Il ne sera pas nécessaire d'avoir de fichier compte-rendu. Les réponses aux questions, les commentaires que vous pourriez avoir sur le fonctionnement de vos programmes, les résultats d'un test avec l'instruction print, etc. pourront être directement insérés dans vos programmes sous forme de commentaires (avec le symbole #).

Enfin, il ne sera pas nécessaire de faire de dépôt pour la question Q1 (qui sera traitée en TD), et donc les fichiers « Q1GeneralConfiguration.py », « Q1main.py » ne seront pas à déposer.

Représentation visuelle d'informations

Introduction

L'objectif de ce TP est de représenter sur une tablette numérique ou un téléphone fonctionnant avec *android* des informations issues de capteurs sous la forme d'indicateurs visuels permettant de renseigner très simplement et rapidement sur une situation. La représentation visuelle d'informations est la traduction de l'expression anglophone « visual control[1] » inspirée des méthodes japonaises.

L'application que nous allons mettre en œuvre consistera à associer des émoticônes à différents capteurs disponibles sur le site d'Annecy par l'intermédiaire d'une connexion internet. Il s'agit de capteurs de température dans la salle des serveurs et sur le toit et d'un capteur mesurant la puissance fournie par un onduleur.

Il ne s'agit que d'un exemple d'application mais ce type de représentation peut tout aussi bien s'appliquer à la surveillance des paramètres d'un bâtiment intelligent, d'une station d'épuration ou de tout autre dispositif.

Nous allons créer des émoticônes colorées. Une situation totalement normale sera représentée par une émoticône verte, souriante, tandis qu'une situation totalement anormale sera représentée par une émoticône rouge faisant la grimace. Pour les situations intermédiaires la couleur et l'amplitude du sourire ou de la grimace dépendront de la mesure.

Figure 1 : exemples d'émoticône

La figure 2 illustre ce que vous devriez obtenir à la fin de ce TP. Les boutons de la zone supérieure permettent de sélectionner le capteur à afficher. Ils sont centrés horizontalement, tout comme l'émoticône.

1 « **Visual control** is a business management technique employed in many places where information is communicated by using visual signals instead of texts or other written instructions. The design is deliberate in allowing quick recognition of the information being communicated, in order to increase efficiency and clarity. These signals can be of many forms, from different coloured clothing for different teams, to focusing measures upon the size of the problem and not the size of the activity, to kanban, obeya and heijunka boxes and many other diverse examples. In The Toyota Way, it is also known as mieruka » - source Wikipédia

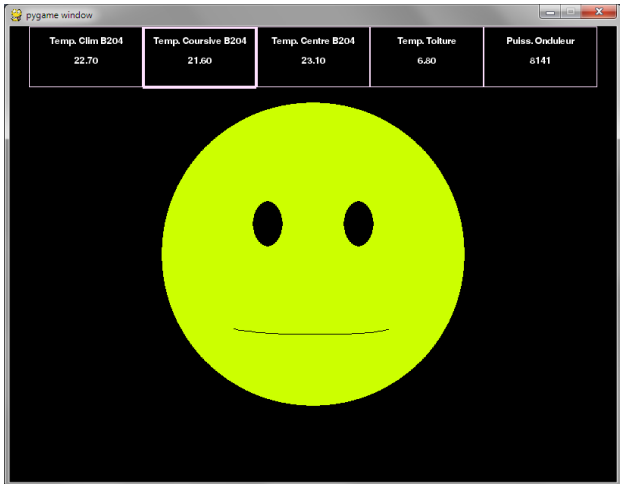

Figure 2 : application en fonctionnement

1) Utilisation de pygame

Pour cette question, il s'agira de tester et de mettre en place ce qui a été fait durant le td7 de préparation dont on rappelle le texte ci-dessous.

Les traitements graphiques seront effectués à l'aide du module *pygame* qui permet, entre autres, de construire des jeux en *Python*. Pour traiter des jeux, le programme doit gérer des événements comme l'appui sur des boutons. Vous trouverez sur le lien suivant la documentation des différents objets et méthodes utilisables dans *pygame* : https://www.pygame.org/docs/.

Q1.a) Pour cette question, on utilisera les fichiers *Q1Main.py* et *Q1GeneralConfiguration.py* donnés en annexe 1 et 2 (ces fichiers sont également disponibles dans le sous-dossier Q1 du sujet TP2 Emoticônes de la plateforme EAD). *Q1Main.py* contient une fonction principale *main()* et son appel. Cette fonction utilise une classe *GeneralConfiguration* définie dans le fichier *Q1GeneralConfiguration.py*. Cette classe contient une partie des fonctions graphiques utiles pour ce TP. L'appel à la méthode *display()* va gérer l'affichage du dessin. Cette méthode appelle la méthode *draw()* qui place les éléments graphiques dans la zone de dessin.

Les couleurs sont représentées dans l'espace dit *RVB* pour *Rouge*, *Vert* et *Bleu* et leur intensité est dans l'intervalle [0, 255]. Ainsi, le triplet [10, 240, 125] désigne une couleur mélange de rouge sombre, de vert intense et de bleu moyen. Les coordonnées sont exprimées en pixel. Pour la suite du TP il est important de bien comprendre les repères associés aux différentes fonctions.

Analyser ce programme en traçant sur une feuille de papier le résultat attendu. En TP, on exécutera le programme en essayant différents jeux de paramètres (en particulier pour les angles associés à l'arc de cercle).

Q1.b) La classe *GeneralConfiguration* contient un attribut *screen* qui est associé à la surface dans laquelle on va dessiner. Étant donné un objet de type *screen*, l'accès à la largeur et à la hauteur de la surface de dessin s'effectue en utilisant respectivement les méthodes *get_width*() et *get_height*(). Donner les instructions permettant d'afficher, à partir de l'objet *generalConfiguration* de la fonction *main()*, la largeur et la hauteur de la zone dessin. En TP, tester cette instruction.

Q1.c) En annexe 3, on donne la documentation de *pygame* pour utiliser la méthode *pygame.draw.rect()*. A l'aide de cette méthode, ajouter un cadre blanc d'épaisseur 1 pixel autour du cercle rouge.

1.1) Ajout de *getters* dans la classe *GeneralConfiguration*

On modifie le constructeur de la classe *GeneralConfiguration* comme suit :

```
# Constructor
def __init__(self) :
    self.initPygame()

    # Parameters for the buttons
    self.buttonWidth = 150
    self.buttonHeight = 80

    # Parameters for the emoticons
    self.emoticonSize = 400
    self.emoticonBorder = 20
```

Le constructeur initialise les paramètres pour les boutons (taille et hauteur), les paramètres pour les émoticônes (taille et bordure). En python les attributs d'une classe sont publics. Par exemple, il est ainsi possible d'écrire :

```
generalConfiguration = GeneralConfiguration()
print(generalConfiguration.buttonWidth)
```

Dans de nombreux langages objets, les attributs d'une classe peuvent être protégés voire privés. Dans ce cas, pour permettre l'accès à l'attribut, on écrit une méthode publique qui retourne la valeur de cet attribut. Ces méthodes sont souvent appelées *getters*. On utilisera la syntaxe *getMyAttribute(self)* pour le *getter* de l'attribut *myAttribute.* Ainsi l'accès à l'attribut *buttonWidth* se fera par le *getter getButtonWidth(self)* et l'on pourra écrire :

```
generalConfiguration = GeneralConfiguration()
print(generalConfiguration.getButtonWidth())
```

Q1.d) Ajouter à la classe *GeneralConfiguration* les *getters* pour les attributs *screen*, *buttonWidth*, *buttonHeight*, *emoticonSize*, *emoticonBorder.* Tester les getters pour vérifier qu'ils retournent bien les valeurs attendues.

De manière symétrique, une méthode publique qui permet de modifier un attribut protégé ou privé est souvent appelée *setter*. Nous verrons une utilisation au paragraphe suivant.

2) L'émoticône

L'émoticône est constituée :

- d'un disque coloré pour la tête ;
- de deux ellipses pour les yeux ;
- d'un arc pour la bouche.

2.1) Dépendance à la classe *GeneralConfiguration*

On utilise maintenant le dossier Q2 disponible sur la plateforme d'EAD qui contient un nouveau fichier *Q2Emoticon.py* contenant la classe *Emoticon.*

L'émoticône doit être inscrite dans un carré dont la longueur des côtés est définie par l'attribut *emoticonSize* de la classe *GeneralConfiguration.* Autour de ce carré, il y a une bordure dont la taille est définie par l'attribut *emoticonBorder,* lui aussi de la classe *GeneralConfiguration.*

La classe *Emoticon* va donc dépendre de la classe *GeneralConfiguration.* Pour gérer ce problème de dépendance, on va effectuer ce que l'on appelle une *injection* dans la classe dépendante. Il existe

plusieurs manières d'injecter une classe dans une autre classe. On verra dans ce TP deux mécanismes d'injection : l'injection par un *setter* et l'injection par le constructeur. Il existe un troisième mécanisme qui consiste à définir ce que l'on appelle une interface. Le choix d'un de ces mécanismes relève du programmeur et dépend du problème.

Remarque importante : L'application considérée est plus complexe que le simple affichage d'une émoticône. Elle comprend en effet la gestion des capteurs et des boutons. Notons, dès à présent, qu'il y aura des dépendances entre les classes associées à cette gestion. Le mécanisme d'injection que nous utilisons dans cette question est donc mis en place à des fins de test et sera revu à la question Q5.

On choisit d'utiliser l'injection dans la classe *Emoticon* par un *setter*. On dote donc la classe *Emoticon* de la méthode *setGeneralConfiguration(self, generalConfiguration)* suivante :

```
def setGeneralConfiguration(self, generalConfiguration) :
    self. generalConfiguration = generalConfiguration
```

Q2.a) Analyser les fichiers *Q2Main.py*, *Q2GeneralConfiguration.py*, *Q2Emoticon.py*.

2.2) La tête

La tête de l'émoticône est un disque inscrit dans le carré dont la longueur des côtés est définie par l'attribut *emoticonSize* de la classe *GeneralConfiguration*. Pour faciliter les dessins de toutes les parties de l'émoticône, on effectue un changement de repère afin de référencer ces parties par rapport au centre de la tête comme le montre la figure 3. La tête est située sous la zone réservée aux boutons (la hauteur des boutons est définie par l'attribut *buttonHeight* de la classe *GeneralConfiguration).* Elle est centrée horizontalement. Une bordure entoure la tête.Sa taille est définie par l'attribut *emoticonBorder* de la classe *GeneralConfiguration*.

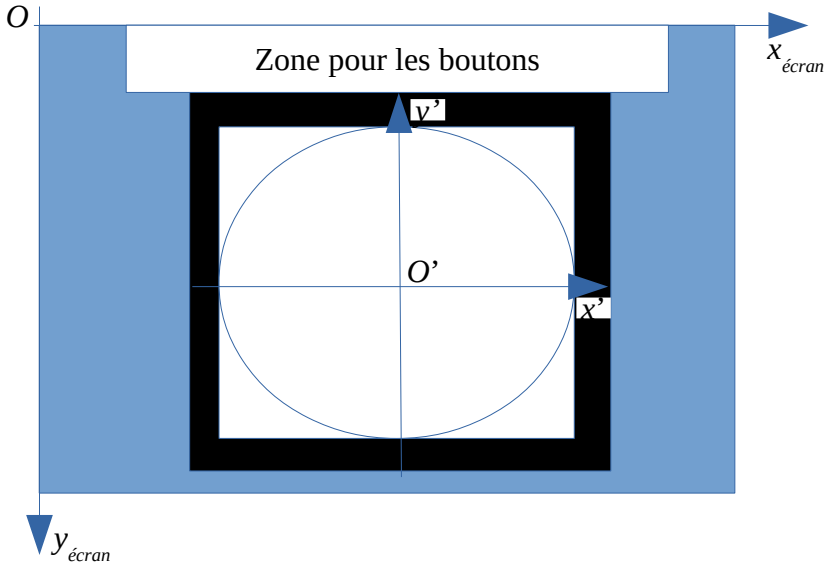

Figure 3 : changement de repère pour le dessin de l'émoticône

Q2.b) Ecrire la méthode *headToArea*(*self*, *position*) de la classe *Emoticon* qui retourne la position, dans le repère de l'écran ($Ox_{écran}y_{écran}$), d'une position exprimée dans le repère de la tête ($O'x'y'$). Les positions seront exprimées sous la forme d'une liste [*x*, *y*] . Tester cette méthode à l'aide d'un *print* dans la fonction *main()*.

Q2.c) On veut gérer maintenant la couleur de la tête de l'émoticône en fonction d'un paramètre *x* prenant ses valeurs dans l'intervalle [-1, 1]. Lorsque ce paramètre vaut -1 la tête doit être totalement

rouge ce qui correspond au vecteur [255, 0, 0] dans l'espace *RVB*. Lorsque ce paramètre vaut 1 la tête doit être totalement verte, ce qui correspond au vecteur [0, 255, 0] dans l'espace *RVB*. Lorsqu'il vaut 0 la tête doit être totalement jaune, ce qui correspond au vecteur [255, 255, 0]. Afin de faire évoluer la couleur de la tête en fonction du paramètre *x* on propose d'utiliser les trois courbes suivantes :

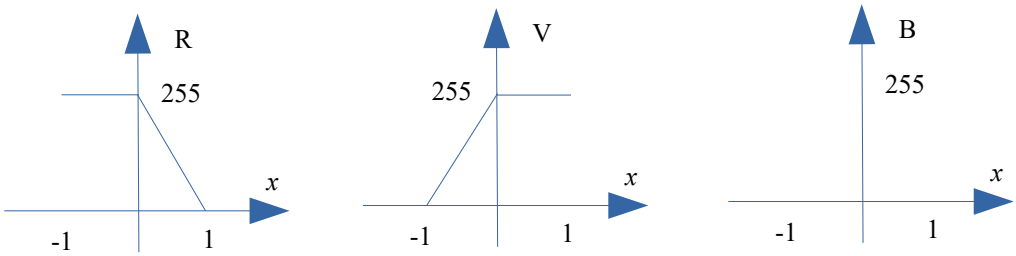

Figure 4 : liens entre le paramètre *x* et les composantes RVB

Ecrire la méthode *color*(*self*, *x*) de la classe *Emoticon* qui retourne la couleur associée au paramètre *x.* Tester cette méthode avec un *print* dans la fonction *main()*.

Q2.d) Ecrire la méthode *head*(*self*, *x*) qui dessine la tête colorée. Remplacer l'instruction *pass* dans la méthode *draw(self)* de la classe *Emoticon* par l'appel à la méthode *head(0)*. Exécuter la fonction *main()*. Tester ensuite avec les paramètres -1 et 1.

2.3) Les yeux

Afin de permettre une modification simple de la taille de l'émoticône, ses paramètres sont définis de manière relative à l'attribut *emoticonSize*, c'est-à-dire du carré dans lequel elle s'inscrit.

Pour cela on dote la classe *Emoticon* de la méthode *setEmoticonParameters(self, size)* suivante :

```
def setEmoticonParameters(self, size) :
    self.eyeWidth = 0.1*size
    self.eyeHeight = 0.15*size
    self.eyeLeftPosition = [-0.15*size, 0.1*size]
    self.eyeRightPosition = [0.15*size, 0.1*size]
    self.mouthPosition = [0, -0.25*size]
    self.mouthMaxHeight = 0.3*size
    self.mouthMaxWidth = 0.55*size
    self.mouthAngle = math.pi/10
```

La figure 5 illustre la position des axes des yeux et de la bouche de l'émoticône.

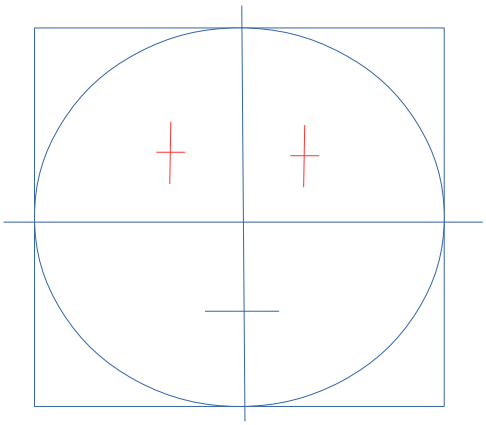

Figure 5 : position des axes des yeux de l'émoticône

En prenant comme repère le centre de la tête, c'est-à-dire le repère *O'x'y'*, les positions relatives sont :

- [0.15**size*, 0.1**size*] pour le centre de l'oeil droit
- [- 0.15**size*, 0.1**size*] pour le centre de l'oeil gauche

Les tailles pour les yeux sont les suivantes :

- 0.15**size* pour la hauteur
- 0.1**size* pour la largeur

Q2.e) Ecrire la méthode *eye*(*self*, *position*) qui dessine un œil noir dont le centre est donné par le paramètre *position* exprimé dans le repère de l'écran. Il est recommandé de revoir la question Q1 pour dessiner correctement cet œil. Insérer l'appel à cette méthode dans la méthode *draw(self)* de la classe *Emoticon* afin de dessiner l'oeil gauche. Dans cet appel, on utilisera méthode *headToArea*(*self*, *position*) pour passer du repère de la tête à celui de l'écran afin que l'oeil soit placé correctement. Faire de même pour l'oeil droit.

2.4) La bouche

La bouche de l'émoticône est dessinée à l'aide d'une portion d'ellipse (revoir la question Q1 pour bien utiliser la méthode *arc*). En prenant comme repère le centre de la tête (*O'x'y'*) la position relative du centre de l'ellipse est [0, -0.25**size*] comme le montre la figure 7.

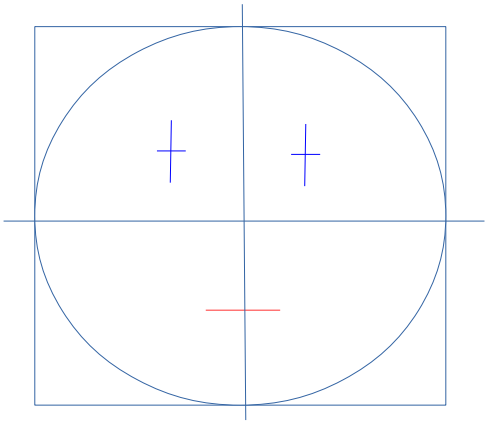

Figure 6 : position de l'axe de la bouche de l'émoticône

Pour réaliser un émoticône totalement souriant, on utilise la portion d'ellipse située entre $\pi + \theta$ et $-\theta$, que l'on translate pour le ramener sur l'axe de la bouche comme le montre la figure 7.

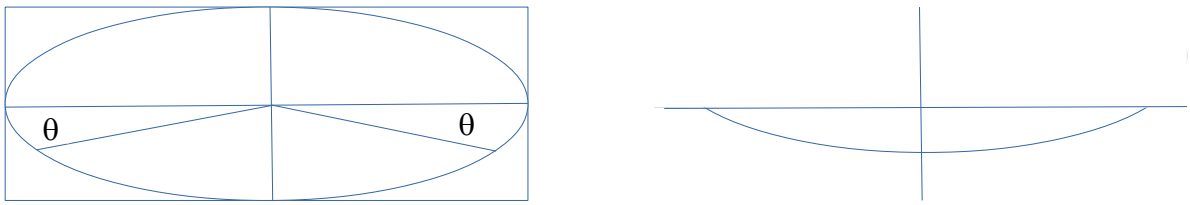

Figure 7 : génération de la bouche

Pour faire évoluer le sourire de l'émoticône en fonction du paramètre *x*, on modifie simplement la hauteur du rectangle dans lequel est inscrite l'ellipse. Le paramètre variant entre -1 et 1, cette opération sera réalisée en multipliant la hauteur maximale par la valeur absolue de *x*. Toutefois, pour des raisons de rendu graphique, si le paramètre *x* est inférieur à 0.15, on dessinera simplement

une ligne sur l'axe de la bouche. La grimace s'engendre de la même manière mais en utilisant un arc entre θ et π - θ. Si le paramètre *x* est supérieur à -0.15, on dessinera aussi une ligne.

On choisit les valeurs suivantes pour les paramètres :

- hauteur maximale du rectangle 0.3**size*
- largeur du rectangle 0.55**size*
- θ = π/10

Q2.f) Ecrire la méthode *mouth*(*self*, *position, x*) qui dessine une bouche en prenant en compte le paramètre *x* qui varie entre -1 et 1. Les coins extérieurs du sourire ou de la grimace sont sur l'axe dont l'origine est définie par le paramètre *position* (voir figure 7), exprimé dans le repère de l'écran. Insérer l'appel à cette méthode dans la méthode *draw(self)* de la classe *Emoticon* afin de dessiner la bouche. On utilisera méthode *headToArea*(*self*, *position*) pour passer du repère de la tête à celui de l'écran afin que la bouche soit placée correctement.

3) Les boutons

On utilise maintenant le dossier Q3 disponible sur la plateforme d'EAD qui contient un nouveau fichier *Q3Button.py* contenant la classe *Button.*

Pour effectuer les tests de cette partie, on utilise le même mécanisme d'injection que pour la classe *Emoticon*.

Les boutons associés à chaque capteur devront afficher le label du capteur ainsi que l'information fournie par le capteur. **Les boutons devront être situés en haut de l'écran et seront centrés**.

3.1) Position des boutons

Dans un premier temps on suppose qu'il n'y a qu'un seul bouton créé comme c'est le cas dans la fonction *main()* du fichier *Q3Main.py* dans le dossier Q3.

Q3.a) Ecrire la méthode *getPosition(self)* de la classe *Button* qui retourne la position du coin supérieur gauche du bouton sous la forme d'une liste [*x*, *y*].

3.2) Affichage d'un texte

Les informations à afficher sont organisées sous forme d'**une liste de labels dont chaque élément correspond à une ligne à afficher**. Par exemple [", 'Temp. Toit', ", '38.4'] devra produire l'affichage suivant (**l'utilisation d'une chaîne vide engendre un saut de ligne**) :

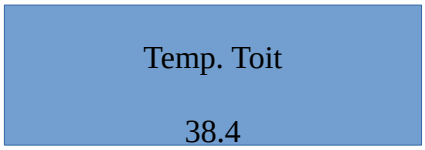

Figure 8 : affichage dans le bouton

L'affichage d'un texte avec *pygame* passe par la création d'une fonte puis d'une image qui va être ensuite collée sur l'écran. L'extrait de code suivant permet de comprendre ce principe.

```
screen = pygame.display.get_surface()
# Creates the font
font = pygame.font.Font(None, 14)
# Creates the text image containing « This is a test » written in white
textImage = font.render('This is a test', 1, [255,255,255])
# Pastes the image on the screen. The upper left corner is at the position 100, 200
```

```
screen.blit(textImage, [100, 200])
```

Deux méthodes associées à l'objet *textImage* sont nécessaires pour permettre le centrage et l'affichage sur plusieurs lignes. Les méthodes *get_width()* et *get_height()* permettent respectivement d'obtenir la largeur et la hauteur de l'image contenant le texte.

Q3.b) Ecrire la méthode *drawLines*(*self*, *lines*) de la classe *Button* qui affiche les lignes passés en paramètre sous forme de liste, dans le bouton.

Q3.c) Ecrire la méthode *draw*(*self)* de la classe *Button* qui dessine le bouton sous la forme d'un cadre blanc de 1 pixel qui contienne les informations à afficher.

4) Gestion des capteurs

On utilise maintenant le dossier Q4 disponible sur la plateforme d'EAD qui contient un nouveau module *Q4Sensor.py* contenant la classe *Sensor.*

Pour effectuer les tests de cette partie, on utilise le même mécanisme d'injection que pour la classe *Emoticon*.

Le fichier *Q4Main.py* illustre la création du capteur mesurant la température de la salle B204 près du climatiseur. Son label est « Temp. Clim B204 », son url est https://www.polytech.univ-smb.fr/apps/myreader/capteur.php?capteur=epua_b204_clim et ses seuils sont [20, 22, 23].

Trois informations sont passées au constructeur de la classe *Sensor* :

- l'url qui sera consultée pour obtenir la mesure ;
- le label qui sera affiché dans le bouton associé au capteur ;
- une liste de 3 valeurs qui vont servir à transformer la mesure en une valeur dans l'intervalle [-1, 1] et permettre ainsi de la représenter visuellement par un émoticône.

Les capteurs disponibles pour ce TP sont les suivants :

Rôle	**Nom pour le paramètre *capteur* de l'url**	**Seuils**
Température dans la salle B204 près du climatiseur	epua_b204_clim	[20, 22, 23]
Température dans la salle B204 au niveau de la coursive	epua_b204_coursive	[20, 22, 23]
Température dans la salle B204 au centre	epua_b204_centre	[20, 22, 23]
Température sur le toit de l'école	epua_toiture	[30, 35, 40]
Puissance délivrée par l'onduleur en W	epua_onduleur1_watts	[10000, 12000, 15000]

Les seuils vont servir à faire une transformation affine par morceaux vers l'intervalle [-1, 1] selon le principe illustré à la figure 9 .

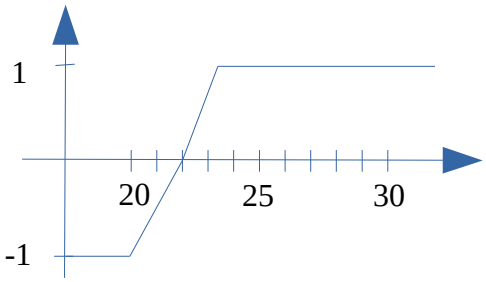

Figure 9 : transformation affine pour le premier capteur

Note : les seuils pour les capteurs de la salle B204 sont volontairement choisis pour engendrer des différences visibles sur les émoticônes et ne correspondent pas au choix qui seraient faits pour une application réelle.

Pour lire ces capteurs, la classe *Sensor* est dotée de méthodes permettant de gérer la connexion internet. L'utilisation de ces méthodes nécessite d'importer les modules *http*, *urllib* comme on peut le voir dans la classe *Sensor*.

Q4.a) Utiliser la méthode *read()* de la classe *Sensor* pour afficher avec un *print* la température près du climatiseur dans la salle B204.

Q4.b) Afficher avec un *print*, le label associé à ce capteur.

Q4.c) Ecrire la méthode *getTransformedValue*(*self*) qui lit le capteur et réalise la transformation affine par morceaux de la mesure.

5) Mise en œuvre de l'application

On utilise maintenant le dossier Q5 disponible sur la plateforme d'EAD.

5.1.) Classes et dépendances

On va maintenant mettre en œuvre l'application complète qui comportera donc quatre classes :

- une classe *GeneralConfiguration* pour gérer la configuration générale et l'ajout de capteur ;
- une classe *Sensor* pour gérer les capteurs ;
- une classe *Emoticon* pour gérer l'émoticône associée à un capteur ;
- une classe *Button* pour gérer le bouton qui permettra de sélectionner le capteur à afficher.

Ainsi les classes *Emoticon* et *Button* ont une dépendance avec la classe *Sensor*. La classe *Sensor* dépend elle-même de la classe *GeneralConfiguration* qui contient la liste des capteurs. Pour gérer ces dépendances, on effectue une injection dans les classes dépendantes. Ainsi, on injectera :

- la classe *GeneralConfiguration* dans la classe *Sensor ;*
- la classe *Sensor* dans les classes *Emoticon* et *Button* .

La figure 10 illustre le sens de l'injection des classes.

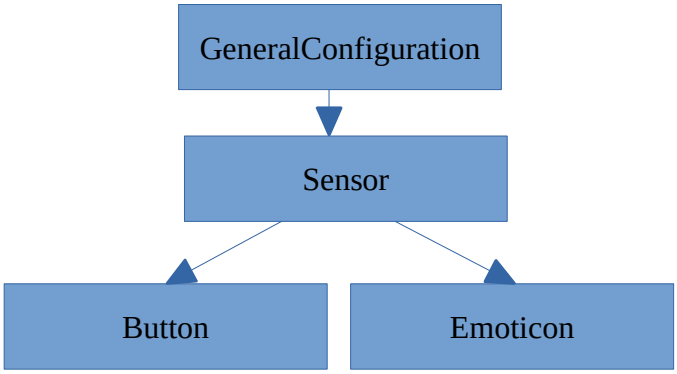

Figure 10 : injection des classes

On utilisera deux mécanismes différents d'injection :

- l'injection par le constructeur de la classe. Dans ce cas, la classe à injecter est passée comme paramètre au constructeur de la classe dépendante. Ce mécanisme sera utilisé pour la dépendance des classes *Emoticon* et *Button* à la classe Sensor. Ainsi on utilise le constructeur suivant pour ces deux classes où le paramètre *sensor* est donc une instance de la classe *Sensor* :

```
def __init__(self, sensor) :
    self.sensor = sensor
```

- l'injection par un *setter*. Dans ce cas la classe à injecter est passée comme paramètre d'une méthode publique. Ce mécanisme sera utilisé pour la dépendance de la classe *Sensor* à la classe *GeneralConfiguration*. Ainsi on utilise le *setter* suivant dans la classe *Sensor :*

```
def setGeneralConfiguration(self, generalConfiguration):
    self.generalConfiguration = generalConfiguration
```

Q5.a) Analyser les fichiers fournis dans le dossier Q5. Dans la fonction *main()* du fichier *Q5Main.py,* on a ajouté un capteur à l'objet *generalConfiguration* à l'aide de la méthode *addSensor*(). Expliquer le fonctionnement de cette méthode. Que vaut l'attribut *sensors* de la classe *GeneralConfiguration* après cet ajout ?

Q5.b) Compléter la classe *GeneralConfiguration* du fichier *Q5GeneralConfiguration.py* avec les *getters* de la question Q1.4. Ajouter les getters pour les attributs *sensors* et *selectedSensor*.

Q5.c) Compléter la classe *Emoticon* du fichier *Q5Emoticon.py* avec les méthodes de la question Q2. On pourra supprimer la méthode *setGeneralConfiguration(self, generalConfiguration)* qui n'est plus utile.

Q5.d) Modifier la méthode *draw(self)* de la classe *GeneralConfiguration* pour qu'elle appelle la méthode *drawEmoticon(self)* de la classe *Sensor*.

Q5.e) Modifier la méthode *drawEmoticon(self)* de la classe *Sensor* pour qu'elle appelle la méthode *draw(self)* de la classe *Emoticon*. Tester le programme.

Q5.f) Compléter la classe *Button* du fichier *Q5Button.py* avec les méthodes de la question Q3. On pourra supprimer la méthode *setGeneralConfiguration(self, generalConfiguration)* qui n'est plus utile.

Q5.g) Modifier la méthode *draw(self)* de la classe *GeneralConfiguration* pour qu'elle appelle la

méthode *drawButton(self)* de la classe *Sensor*, en plus de la méthode *drawEmoticon(self)*.

Q5.h) Modifier la méthode *drawButton(self)* de la classe *Sensor* pour qu'elle appelle la méthode *draw(self)* de la classe *Button.* Tester le programme.

5.2) Prise en compte de la sélection

Q5.i) Pour savoir si un capteur est sélectionné, écrire la méthode *positionToSensorId*(*self*, *position*) dans la classe *GeneralConfiguration* qui retourne le numéro du capteur si le paramètre *position* est dans une position dans la zone du bouton et *None* sinon.

Q5.j) Ecrire la méthode *checkIfSensorChanged*(*self*, *eventPosition*) de la classe *GeneralConfiguration* qui modifie l'attribut *selectedSensor* de cette classe.

Q5.k) Ecrire la méthode *isSelected(self)*, dans la classe *Sensor*, qui retourne *True* si le capteur considéré est le capteur sélectionné. Utiliser cette méthode pour que le cadre du bouton du capteur sélectionné ait une largeur de 3 pixels.

Q5.l) Ajouter les autres capteurs dans la fonction *main()*. Effectuer les quelques modifications nécessaires dans les classes pour prendre en compte le fait que l'attribut *sensors* de la classe *GeneralConfiguration* est maintenant une liste qui contient plusieurs capteurs. Tester le fonctionnent de votre application. L'action sur un bouton doit provoquer l'affichage de l'émoticône associée au capteur auquel ce bouton est lié.

6) Amélioration de l'application

Proposer et implanter toute amélioration que vous jugerez utile pour cette application.

Annexe 1 : fichier Q1main.py

```
import pygame
from Q1GeneralConfiguration import GeneralConfiguration

def main():

    # Creates an instance of the class GeneralConfiguration
    generalConfiguration = GeneralConfiguration()

    # Infinite loop
    while True:

        # Waits for an event
        event = pygame.event.wait()

        # Checks if the user wants to quit
        if event.type == pygame.QUIT:
            pygame.quit()
            break

        # Display the drawing
        elif event.type == pygame.USEREVENT:
            generalConfiguration.display()

        # Checks if the user has clicked with the mouse
        elif event.type == pygame.MOUSEBUTTONDOWN:
            # Just pass
            pass

# Calls the main function
if __name__ == "__main__":
    main()
```

Annexe 2 : fichier Q1GeneralConfiguration.py

```
import pygame
import math

class GeneralConfiguration:
    # Constructor
    def __init__(self) :
        self.initPygame()

    # Initializes pygame
    def initPygame(self):
        #Initialization
        pygame.init()
        # Sets the screen size.
        pygame.display.set_mode((800, 600))
        # Sets the timer to check event every 200 ms
        pygame.time.set_timer(pygame.USEREVENT, 200)
        # Gets pygame screen
        self.screen = pygame.display.get_surface()

    # Draws on pygame screen
    def draw(self):
        # Draws a circle in red with a center in 100, 100
        # and a radius equal to 80
        pygame.draw.circle(self.screen, [255, 0, 0], [100, 100], 80)

        # Draws a black ellipse contained in a rectangle whose left upper
        # corner is 50,60, width is 15 and height is 20
        pygame.draw.ellipse(self.screen, [0,0,0], [50, 60, 15, 20])

        # Draws a black arc contained in a rectangle whose left upper
        # corner is 60,120, width is 80, height is 30,
        # starting angle=5*pi/4, ending angle=7*pi/4

        pygame.draw.arc(self.screen, [0,0,0], [60, 120, 80, 30], \
                        5*math.pi/4, 7*math.pi/4)

        # Draws a black line starting in position 50,100
        # and ending in position 150,100
        pygame.draw.line(self.screen, [0,0,0], [50,100], [150,100])

    # Displays pygame screen
    def display(self):
        # Draws on the screen surface
        self.draw()
        # Updates the displat and clears new timer events
        pygame.display.flip()
        pygame.event.clear(pygame.USEREVENT)
```

Annexe 3 : documentation Pygame sur le tracé d'un rectangle

```
pygame.draw.rect(surface, color, rect, width)
```

Parameters:

```
- surface → surface to draw on
- color ([R, G, B]) →  color to draw (int)
- rect ([x_top_left, y_top_left, width, height]) →  rectangle to draw
(int)
- width (optional, int) → used for line thickness or to indicate that the
rectangle is to be filled (not to be confused with the width value of the
rect parameter)

      if width == 0, (default) fill the rectangle
      if width > 0, used for line thickness
      if width < 0, nothing will be drawn
```