WOLF Manon
GIGANTE Elodie
Date du TP : 08/12/2021

Rapport de TP2 – Lecture automatique de chiffres par analyse d'image

III. Travail préparatoire

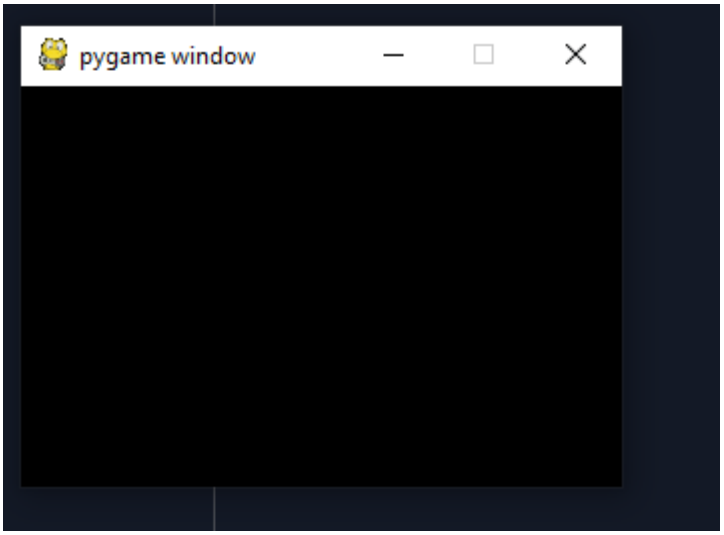
Utilisation de Pygame

1) Pygame crée une fenêtre et la ferme directement

2)

```
:linenos:

import pygame


pygame.init()
ecran = pygame.display.set_mode((300, 200))

continuer = True
while continuer:
    for event in pygame.event.get():
        if event.type == pygame.KEYDOWN:
            continuer = False

pygame.quit()
```

Pygame crée une fenêtre et la maintient ouvert tant que l'on appui pas sur une touche



WOLF Manon
GIGANTE Elodie
Date du TP : 08/12/2021

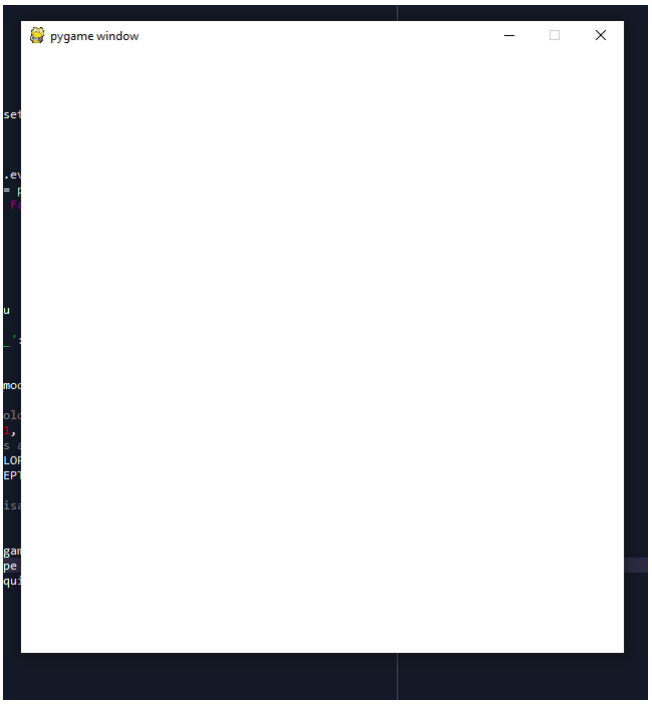
Utilisation de Pyopengl pour représenter des objets 3D

1)

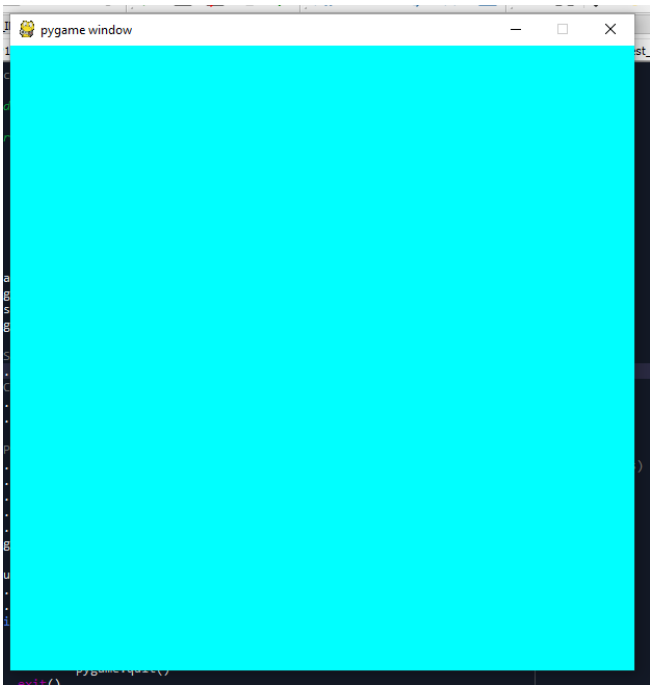
```
:linenos:

import pygame

import OpenGL.GL as gl

import OpenGL.GLU as glu

if __name__ == '__main__':

    pygame.init()

    display=(600,600)

    pygame.display.set_mode(display, pygame.DOUBLEBUF | pygame.OPENGL)

    # Sets the screen color (white)

    gl.glClearColor(1, 1, 1, 1)

    # Clears the buffers and sets DEPTH_TEST to remove hidden surfaces

    gl.glClear(gl.GL_COLOR_BUFFER_BIT | gl.GL_DEPTH_BUFFER_BIT)

    gl.glEnable(gl.GL_DEPTH_TEST)

    # Placer ici l'utilisation de gluPerspective.

    while True:

        for event in pygame.event.get():

            if event.type == pygame.QUIT:

                  pygame.quit()

              exit()
```

Crée une fenêtre d’une certaine taille

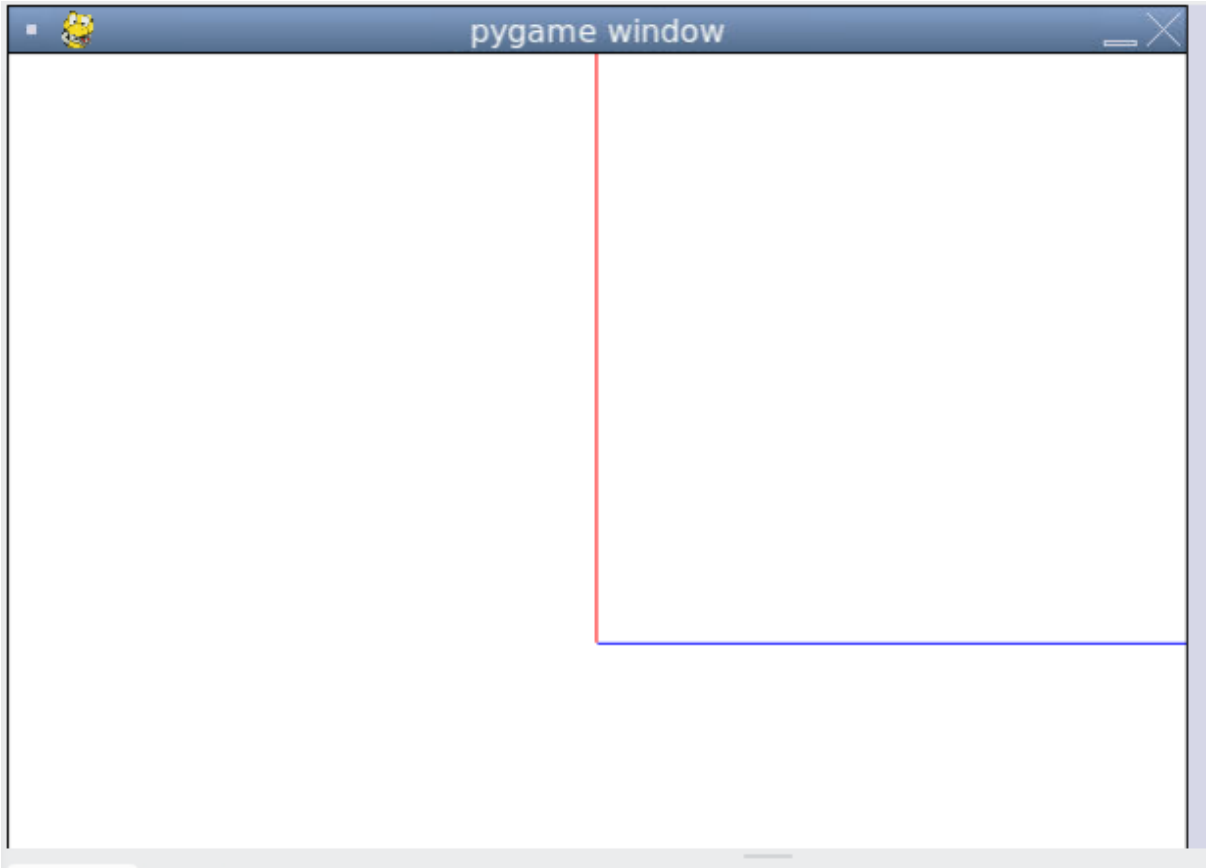
WOLF Manon
GIGANTE Elodie
Date du TP : 08/12/2021



On peut changer la couleur en changeant le code dans la fonction



2) Créer un repère

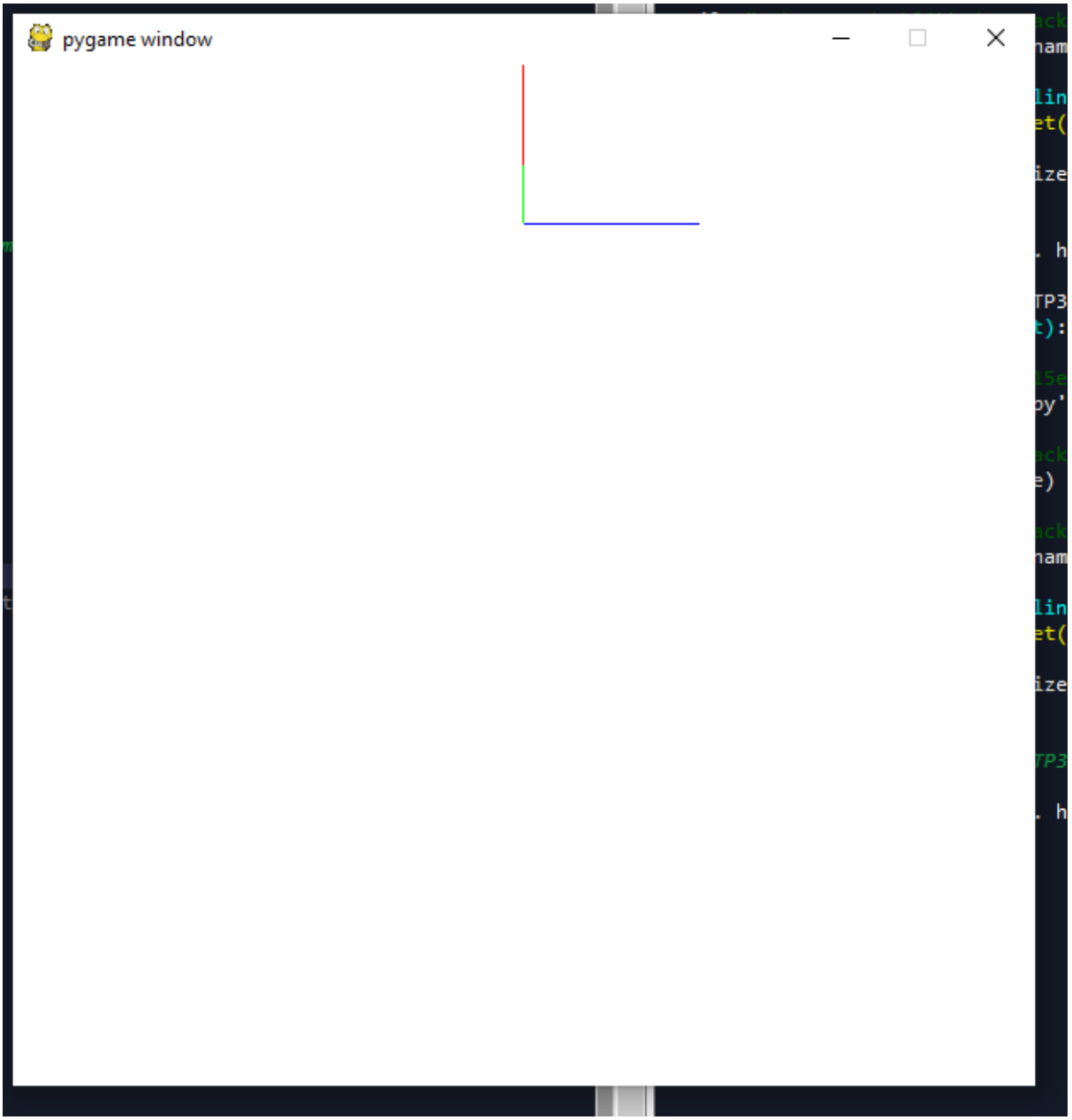



```
gl.glBegin(gl.GL_LINES) # Indique que l'on va commencer un trace en
mode lignes (segments)
   gl.glColor3fv([1, 0, 0]) # Indique la couleur du prochain segment en
RGB

   gl.glVertex3fv((0,0, -2)) # Premier vertice : départ de la ligne
   gl.glVertex3fv((0, 1, -2)) # Deuxième vertice : fin de la ligne
    gl.glColor3fv([0, 0, 1]) # Indique la couleur du prochain segment
en RGB
   gl.glVertex3fv((0,0, -2)) # Premier vertice : départ de la ligne
   gl.glVertex3fv((1, 0, -2)) # Deuxième vertice : fin de la ligne

   gl.glColor3fv([0, 1, 0]) # Indique la couleur du prochain segment en
RGB
   gl.glVertex3fv((0,0, -2)) # Premier vertice : départ de la ligne
   gl.glVertex3fv((0, 0, -1)) # Deuxième vertice : fin de la ligne

   gl.glEnd() # Find du tracé
   pygame.display.flip() # Met à jour l'affichage de la fenêtre
graphique
```

WOLF Manon
GIGANTE Elodie
Date du TP : 08/12/2021

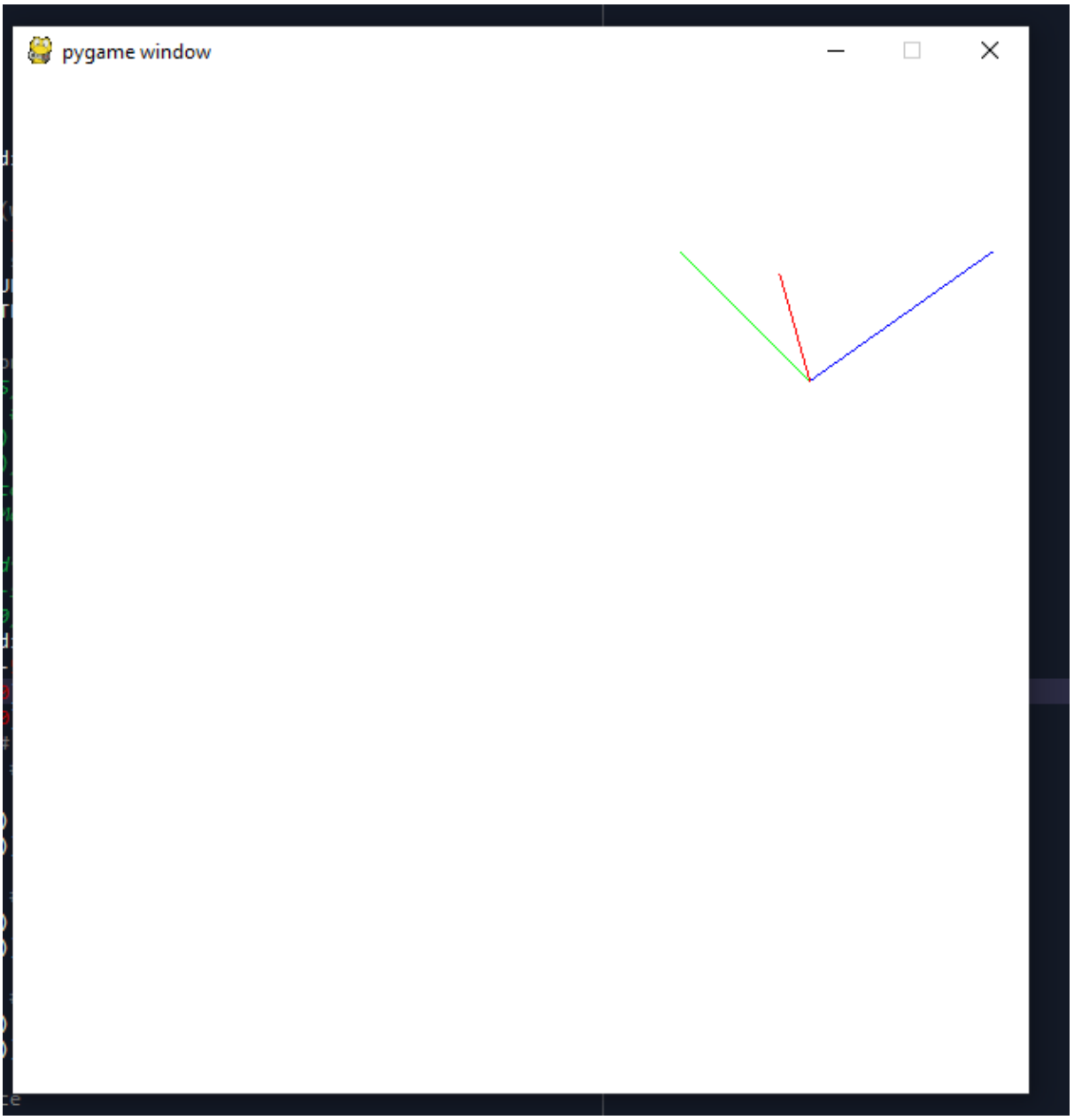
3) Crée des translations et rotations

Translation de vecteur (0,2,-5)




```
gl.glTranslatef(0.0, 2, -5)
```

Rotations de -45° autour de l'axe x et de de l'axe y :




```
gl.glRotatef(-45, 1, 0, 0)
gl.glRotatef(-45, 0, 1, 0)
```

Remarque : Il faut aussi effectuer une translation en même temps sinon la fenêtre n'affiche rien

Code complet pour cette question :

```
import pygame
import OpenGL.GL as gl
import OpenGL.GLU as glu
```

```
if __name__ == '__main__':
    pygame.init()
    display=(600,600)
    pygame.display.set_mode(display, pygame.DOUBLEBUF |
pygame.OPENGL)

    # Sets the screen color (white)
    gl.glClearColor(1, 1, 1, 1)
    # Clears the buffers and sets DEPTH_TEST to remove hidden
surfaces
    gl.glClear(gl.GL_COLOR_BUFFER_BIT |
gl.GL_DEPTH_BUFFER_BIT)
    gl.glEnable(gl.GL_DEPTH_TEST)

    # Placer ici l'utilisation de gluPerspective.
    glu.gluPerspective(45, (display[0] / display[1]), 0.1,
50.0)
    gl.glTranslatef(0.0, 2, -5)
    gl.glRotatef(-45, 1, 0, 0)
    gl.glRotatef(-60, 0, 1, 0)
    gl.glBegin(gl.GL_LINES) # Indique que l'on va commencer un trace en
mode lignes (segments)
    gl.glColor3fv([0, 0, 0]) # Indique la couleur du prochian segment en
RGB

    gl.glVertex3fv((0,0, -2)) # Premier vertice : départ de la ligne
    gl.glVertex3fv((0, 1, -2)) # Deuxième vertice : fin de la ligne

    gl.glColor3fv([0, 0, 1]) # Indique la couleur du prochian segment en
RGB
    gl.glVertex3fv((0,0, -2)) # Premier vertice : départ de la ligne
    gl.glVertex3fv((1, 0, -2)) # Deuxième vertice : fin de la ligne

    gl.glColor3fv([0, 1, 0]) # Indique la couleur du prochain segment en
RGB
    gl.glVertex3fv((0,0, -2)) # Premier vertice : départ de la ligne
    gl.glVertex3fv((0, 0, -1)) # Deuxième vertice : fin de la ligne

    gl.glEnd() # Fin du tracé
    pygame.display.flip() # Met à jour l'affichage de la fenêtre graphique

    while True:
```

```
        for event in pygame.event.get():
          if event.type == pygame.QUIT:
               pygame.quit()
        exit()
```

Découverte de l'environnement du travail du TP

```
Configuration({'screenPosition': -5, 'xAxisColor': [1, 1, 0]}). \
    setParameter('xAxisColor', [1, 1, 0]). \
    setParameter('yAxisCo lor', [0,1,1]). \
    display()
```

- Expliquer pourquoi le chaînage de l'appel des méthodes *setParameter()* et *display()* est possible.

On peut chaîner les deux, car Configuration initie des axes, puis setParameter va les modifier, et enfin display affiche la configuration modifiée. Les trois méthodes se succèdent dans l'ordre.

- Un traitement particulier est effectué dans le « setter » pour le paramètre *screenPosition*. Expliquer pourquoi ce traitement particulier doit être effectué.

-> `'screenPosition': -5` permet d'afficher le résultat "plus proche" de l'écran et donc de mieux voir les axes

(1). c) Ajouter une seule instruction à la méthode *initializeTransformationMatrix()* pour que l'axe *z* soit représenté verticalement sur l'écran et que l'axe *x* soit représenté horizontalement. L'axe *y* devient alors la profondeur. Garder cette transformation pour la suite.

```
gl.glRotatef(-90, 1, 0, 0)
```

(1). d)

```
def processKeyDownEvent(self):
       # Rotates around the z-axis
       if self.event.dict['unicode'] == 'Z' or (self.event.mod &
pygame.KMOD_SHIFT and self.event.key == pygame.K_z):
           gl.glRotate(-2.5, 0, 0, 1)
       elif self.event.dict['unicode'] == 'z' or self.event.key ==
pygame.K_z:
           gl.glRotate(2.5, 0, 0, 1)
```

```
        elif self.event.dict['unicode'] == 'u':
          gl.glScalef(1.1,1.1,1.1)
        elif self.event.dict['unicode'] == 'd':
          gl.glScalef(1/(1.1),1/(1.1),1/(1.1))
```

Pour simplifier la réalisation, nous avons choisi les touches “u” et “d” (pour up et down) à la place de pageUp et pageDown. On associe un grossissement de 1.1 dans toutes les directions à la touche u, et un grossissement de 1/1.1 pour la touche d.

(1). e)

```
def processMouseButtonDownEvent(self):
       if self.event.button==4:

           gl.glScalef(1.1,1.1,1.1)
       if self.event.button==5:
           gl.glScalef(1/(1.1),1/(1.1),1/(1.1))
```

Le principe est le même qu’avant, mais on associe maintenant la commande à la molette de la souris.

(1). f)

```
def processMouseMotionEvent(self):
       if pygame.mouse.get_pressed()[0]==1 :
         pygame.MOUSEMOTION
         x=self.event.rel[0]
         z=self.event.rel[1]
         gl.glRotate(4,x,0,z)
       if pygame.mouse.get_pressed()[2]==1 :
         pygame.MOUSEMOTION
         x=self.event.rel[0]
         z=self.event.rel[1]
         gl.glTranslatef(-x/20,0,-z/20)
```

On crée les deux cas : bouton gauche pressé puis bouton droit pressé.

Pour la rotation, on indique un mouvement selon x et z, en donnant les valeurs de déplacement qui vont déterminer la vitesse de rotation. Nous avons fixé à 4 car cela nous semblait le déplacement le plus agréable.

Pour la translation, le principe est similaire, mais nous avons eu d'abord du mal à comprendre comment fonctionnait la fonction ; nous pensions qu'il fallait indiquer une valeur de déplacement en x et en z, mais il faut en fait multiplier ou diviser les valeurs des coordonnées. Là encore, nous avons choisi une "vitesse" de 1/20.

Remarque : la translation est en fait une translation circulaire autour de l'origine, et non une translation linéaire.

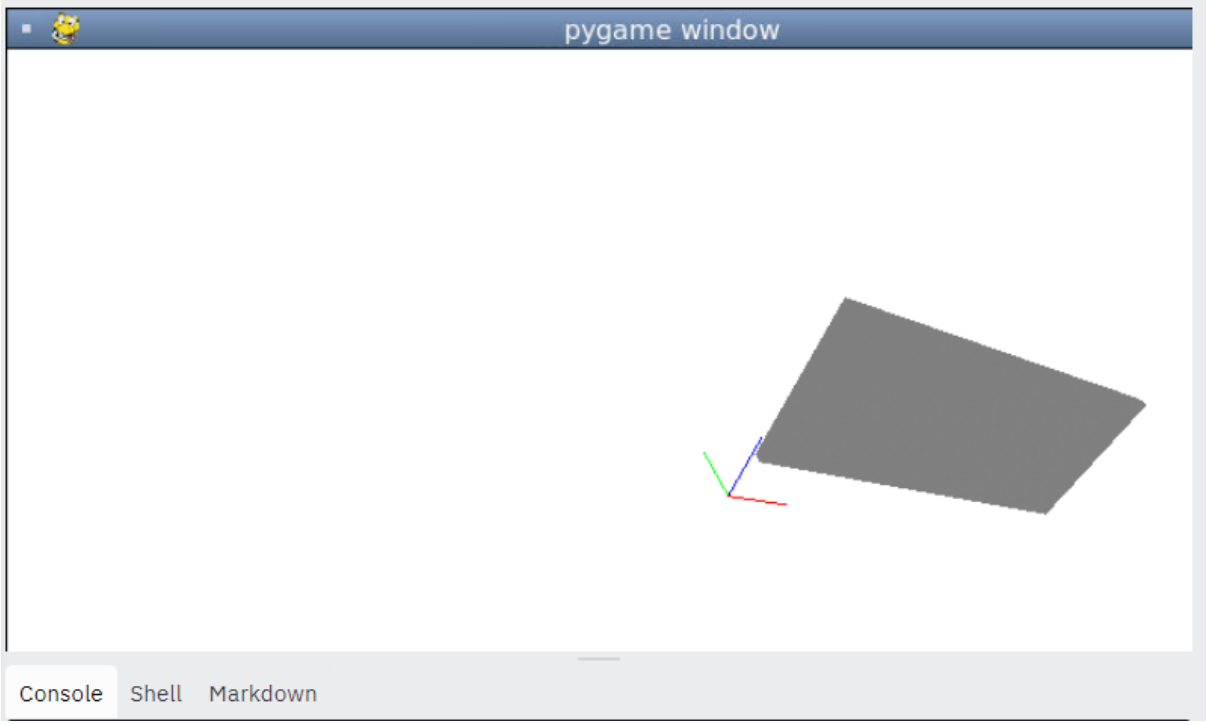
IV - Création d'une section

(2). a)

```
   def generate(self):
       self.vertices = [
         [0, 0, 0 ],
         [0, 0, self.parameters['height']],
         [self.parameters['width'], 0, self.parameters['height']],
         [self.parameters['width'], 0, 0],
         [0, self.parameters['thickness'], 0],
         [0,self.parameters['thickness'],self.parameters['height']],
         [self.parameters['width'],self.parameters['thickness'],
self.parameters['height']],
         [self.parameters['width'], self.parameters['thickness'], 0]
               ]
       self.faces = [
               [0, 3, 2, 1],
               [0, 4, 5, 1],
               [4, 5, 6, 7],
               [3, 2, 6, 7],
               [1, 5, 6, 2],
               [0, 4, 7, 3]
               ]
```

On utilise le modèle de base en créant tous les sommets à partir des coordonnées, et on fait de même avec les faces formées par différentes combinaisons de sommets.

(2). b) Méthode draw




```
def draw(self):
        gl.glPushMatrix()
        gl.glTranslate(self.parameters['position'][0],
self.parameters['position'][1], self.parameters['position'][2])
        gl.glPolygonMode(gl.GL_FRONT_AND_BACK, gl.GL_FILL) # on trace
les faces : GL_FILL
        gl.glBegin(gl.GL_QUADS) # Tracé d'un quadrilatère
        gl.glColor3fv([0.5, 0.5, 0.5]) # Couleur gris moyen
        for i in self.faces:
          for j in i:
            gl.glVertex3fv(self.vertices[j])
        gl.glEnd()
        gl.glPopMatrix()
```

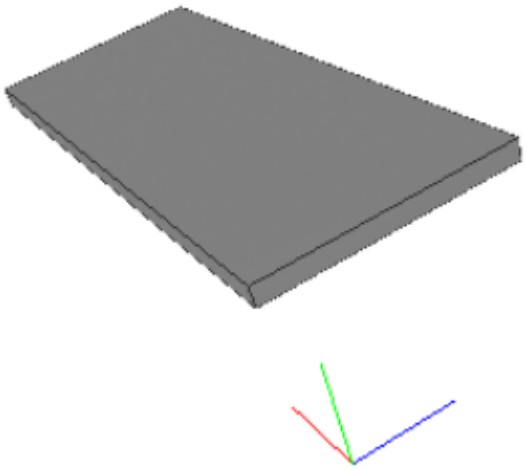
On utilise PushMatrix et Popmatrix pour le stockage et le rappel des informations.
La translation avec les paramètres position permet de placer chaque élément au bon endroit.
Ensuite on définit la couleur, puis on fait une double boucle : on parcourt toutes les faces, et pour chaque face on place tous les points.

(2).c)

```
def drawEdges(self):
```

```
        # A compléter en remplaçant pass par votre code
        gl.glPushMatrix()
        gl.glTranslate(self.parameters['position'][0],
self.parameters['position'][1], self.parameters['position'][2])
        gl.glPolygonMode(gl.GL_FRONT_AND_BACK, gl.GL_LINE)
        gl.glBegin(gl.GL_QUADS) # Tracé d'un quadrilatère
        gl.glColor3fv([0.2, 0.2, 0.2])
        for i in self.faces:
          for j in i:
            gl.glVertex3fv(self.vertices[j])
        gl.glEnd()
        gl.glPopMatrix()
```

On écrit drawEdges de façon similaire à draw, mais en traçant les lignes au lieu de remplir les faces.



On modifie ensuite la méthode draw dans Configuration pour ajouter les arêtes et la condition:

```
    # Draws the objects if any
        for x in self.objects:
            if x.parameters['edges'] == True:
              x.drawEdges()
            x.draw()
```

(3).a)

```
 def draw(self):
     gl.glPushMatrix()
     gl.glRotatef(self.parameters['orientation'],0,0,1)
     self.parentSection.drawEdges()
     for x in self.objects:
       x.draw()
     gl.glPopMatrix()
```

On définit le mur de façon similaire aux sections, en ajoutant une rotation correspondant à l'orientation.

VI - Création d'une maison

(4).a)

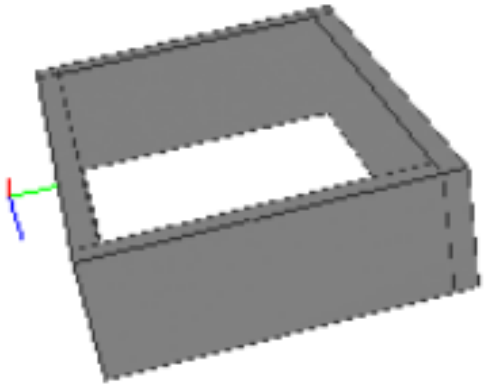
```
# Draws the house
   def draw(self):
       gl.glPushMatrix()
       gl.glTranslate(self.parameters['position'][0],
self.parameters['position'][1],
       self.parameters['position'][2])
       gl.glRotatef(self.parameters['orientation'],0,0,1)
       for x in self.objects:
         x.draw()
       gl.glPopMatrix()
```

On définit la fonction draw de house de la même façon que pour le mur, en ajoutant une boucle pour intégrer tous les éléments.

```
  wall = Wall({'position': [0, 0, 0], 'width':10, 'height':2.6,
'thickness':0.5, 'edges':True})
   wall2 =
Wall({'position':[0,0,0],'width':7,'thickness':0.5,'height':2.6,
'orientation':90})
   wall3 = Wall({'position':[-0.5,7,0],'width':10,
'thickness':0.5,'height':2.6})
   wall4 =
Wall({'position':[0,-10,0],'width':7.5,'thickness':0.5,'height':2.6,'or
ientation':90})
   house = House({'position': [-3, 1, 0], 'orientation':0})
```

```
house.add(wall).add(wall3).add(wall4).add(wall2)
return Configuration().add(house)
```

On crée 4 murs dont on définit les paramètres de façon à être bien alignés, et on les regroupe dans une maison, que l'on affiche ensuite.



Conclusion :

Ce TP nous a permis de renforcer notre compréhension de l'utilisation des fonctions, méthodes et classes sans les confondre, ainsi que de se rendre compte de l'utilité de la programmation appliquée à un problème assez complet. Comme lors du TP précédent, nous avons pu aussi observer les résultats de nos codes grâce aux images, ce qui rendait le problème concret.

Il était intéressant de voir comment fonctionne et est programmé un logiciel de construction 3D.

Nous avons rencontré quelques difficultés à certains moments, quand il fallait comprendre et utiliser les fonctions déjà existantes et que nous n'avions pas écrites nous-mêmes. La gestion des différents fichiers et de leurs interactions était aussi assez difficile.