

Rapport 1/2 TP3 Numération et Algorithmie

Andrew MARY HUET DE BAROCHEZ

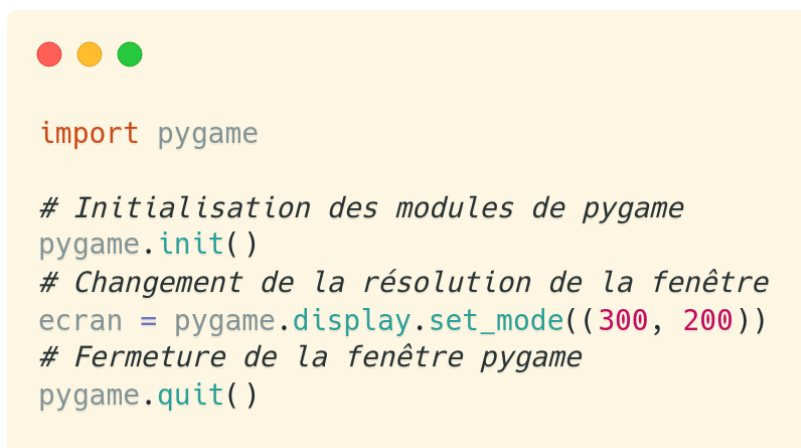
Calvin NGOR

I. Travail Préparatoire

Utilisation de Pygame

1)

Lorsque l'on exécute le code 1, une fenêtre de taille 300x200 s'affiche et se ferme instantanément. C'est tout à fait normal car on appelle la méthode quit() directement après, donc pygame considère qu'il n'a plus de travail à effectuer.

A screenshot of a code editor with a light yellow background. At the top left, there are three colored circles: red, yellow, and green. The code is written in a monospaced font with syntax highlighting. The code is as follows:

```
import pygame

# Initialisation des modules de pygame
pygame.init()
# Changement de la résolution de la fenêtre
ecran = pygame.display.set_mode((300, 200))
# Fermeture de la fenêtre pygame
pygame.quit()
```

Code 1. Début avec pygame

2)

Le code 2 implémente ici une boucle while permettant à la fenêtre de rester ouverte (voir Capture 1) et de ne pas se fermer directement comme pour le code 1.

Dans cette boucle while se trouve une boucle for parcourant la liste d'événements pouvant être obtenue à partir de la méthode get() de la classe event situé dans pygame. Ensuite une condition permet de savoir si un des événements correspond à une pression de touche. Si c'est le cas, la condition de la boucle for passera à False, la boucle s'arrêtera et la fenêtre se fermera.

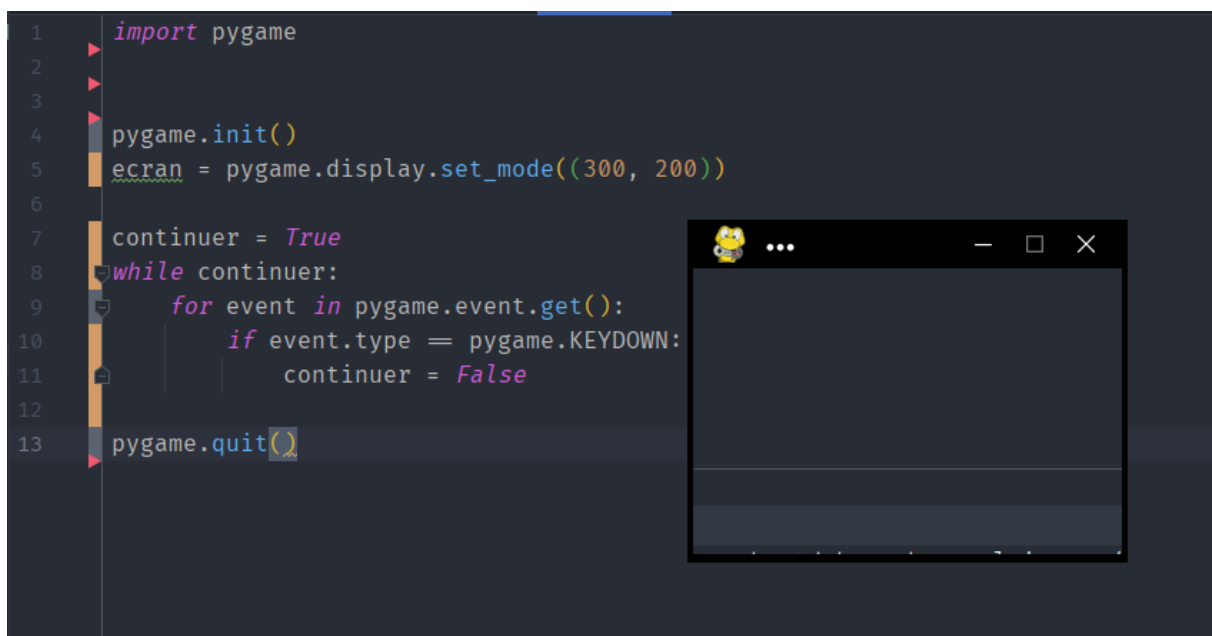
```
import pygame

# Initialisation des modules de pygame
pygame.init()
# Changement de la résolution de la fenêtre
ecran = pygame.display.set_mode((300, 200))

continuer = True
# Boucle du jeu
while continuer:
    # Parcours des différents événements
    for event in pygame.event.get():
        # Arrêt de la boucle dès qu'une touche est pressée
        if event.type == pygame.KEYDOWN:
            continuer = False

# Fermeture de la fenêtre pygame
pygame.quit()
```

Code 2. Ajout d'une boucle pour pygame



Capture 1. Fenêtre pygame vide

Utilisation de Pyopengl pour représenter des objets 3D

1)

En faisant un copier/coller il y a eu une petite erreur d'indentation sur la ligne où se situe le `exit()` (Voir code 3). Cela résultait en la fermeture instantanée de la fenêtre créée, peu importe la situation dans laquelle nous étions alors que nous voulions que la fenêtre ne se ferme seulement au moment où l'on appuie sur la petite croix. C'était tout simplement car le `exit()` n'était pas indenté dans la condition.

On ajoute ensuite un appel à la fonction `gluPerspective` du module `GLU` afin d'initialiser notre matrice perspective. Pour remplir le paramètre `aspect` nous avons effectué une recherche dans la doc de `PyOpenGL`.

On trouve : "Aspect: Specifies the aspect ratio that determines the field of view in the x direction. The aspect ratio is the ratio of x (width) to y (height)." Il suffit donc de passer le ratio de notre fenêtre.

```
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)

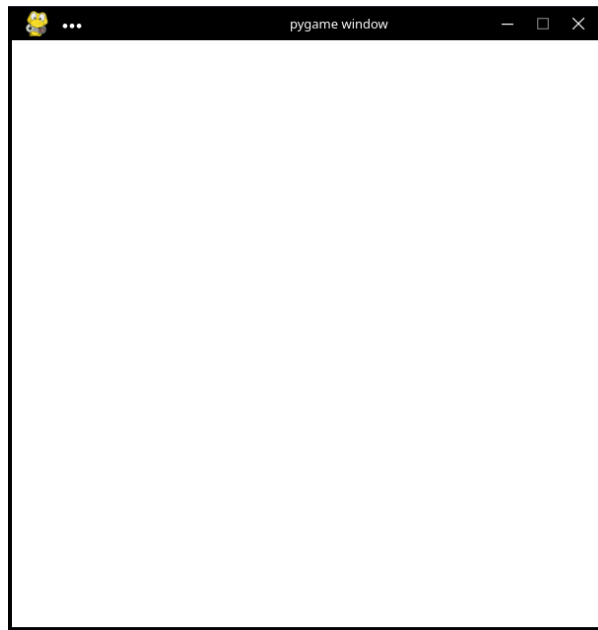
    # Sets the screen color (white)
    gl.glClearColor(1, 1, 1, 1)
    # Clears the buffers and sets DEPTH_TEST to remove hidden surfaces
    gl.glClear(gl.GL_COLOR_BUFFER_BIT | gl.GL_DEPTH_BUFFER_BIT)
    gl.glEnable(gl.GL_DEPTH_TEST)

    glu.gluPerspective(45, (display[0] / display[1]), 0.1, 50.0)

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

Code 3. Initialisation d'OpenGL

Une fenêtre blanche s'affiche à l'exécution du code 3 (voir capture 1).



Capture 1. Fenêtre pygame

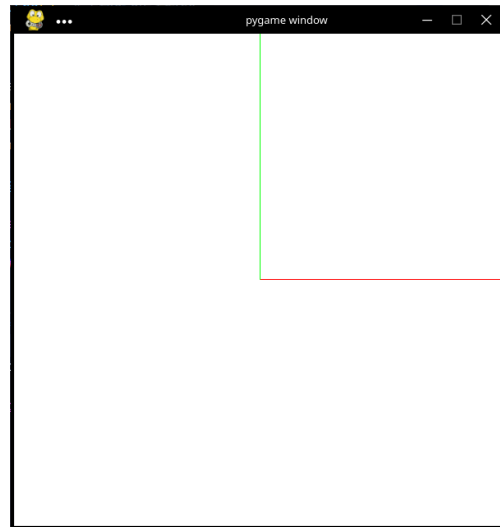
2)

Pour afficher nos trois axes nous créons une nouvelle fonction (Code 4) qui sera exécutée dans la boucle while afin d'alléger cette dernière.

```
def refresh_display():  
    # Sets the screen color (white)  
    gl.glClearColor(1.0, 1.0, 1.0, 1.0)  
    # 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)  
  
    gl.glBegin(gl.GL_LINES) # Indique que l'on va commencer un trace en mode lignes  
  
    # Axe des X  
    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((2, 0, -2)) # Deuxième vertice : fin de la ligne  
  
    # Axe des Y  
    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, 2, -2)) # Deuxième vertice : fin de la ligne  
  
    # Axe des Z  
    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((0, 0, 0)) # Deuxième vertice : fin de la ligne  
  
    gl.glEnd() # Fin du tracé  
    pygame.display.flip() # Met à jour l'affichage de la fenêtre graphique
```

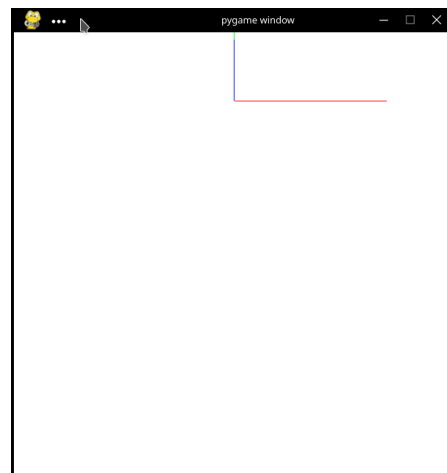
Code 4. Fonction refresh_display affichant nos trois axes

La capture 2 montre l'affichage obtenu. On peut voir l'axe X et Y cependant on ne voit pas l'axe Z en bleu. C'est tout à fait normal car l'axe Z possède la même direction que la caméra et il est placé au milieu. Il apparaît donc comme un point.



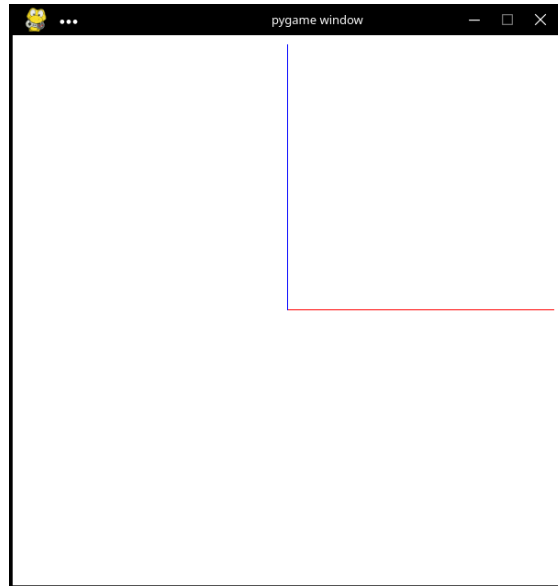
Capture 2. Affichage des axes

Après avoir effectué la translation, on remarque que l'axe Z est maintenant apparent et cache partiellement l'axe Y (Capture 3). L'axe Z est désormais visible car il n'est plus au milieu de l'écran, on peut le voir en perspective.



Capture 3. Affichage des axes après une translation

On effectue ensuite une rotation autour de l'axe X faisant revenir les axes au centre de la caméra (Capture 4). Les axes reviennent au milieu car la rotation est effectuée par rapport à l'axe X de notre plan et non par rapport à l'axe X tracé en rouge. L'axe des Z remplace maintenant l'axe des Y. L'axe des Y est maintenant dans la direction précédente de Z mais avec un sens opposé (les négatifs étant vers la caméra).



Capture 4. Translation et rotation de nos axes.

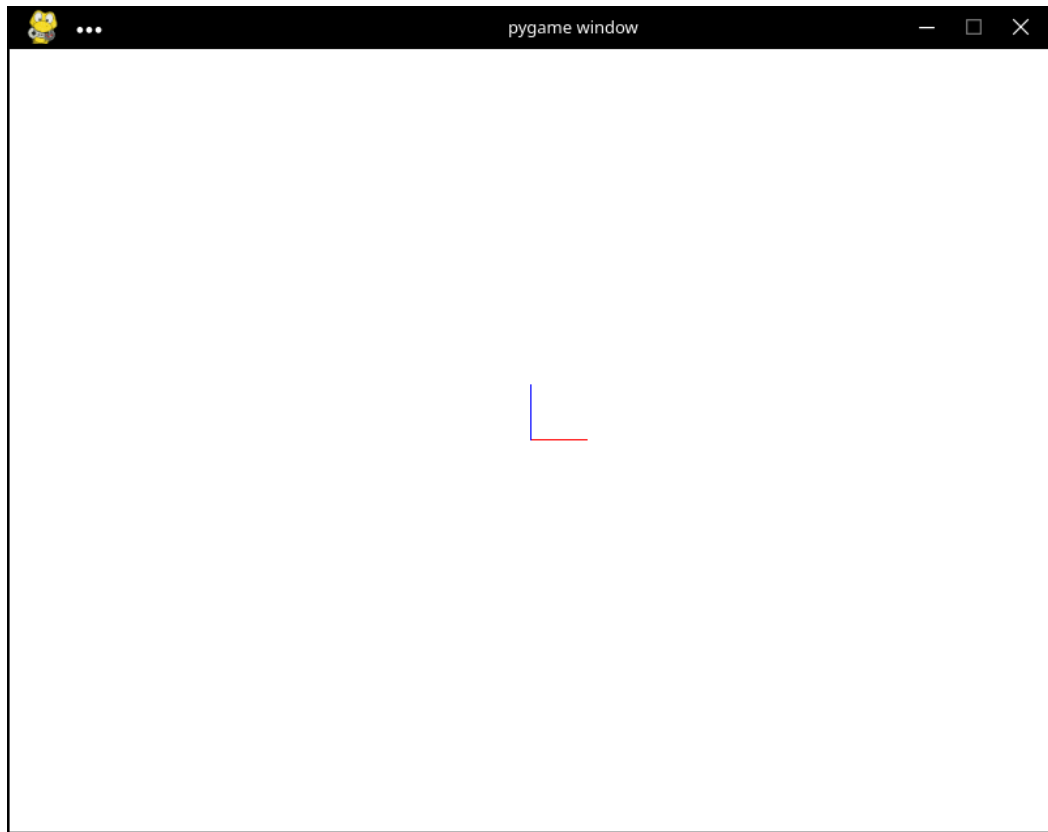
Découverte de l'environnement du travail du TP

1) a)

Le fichier `main.py` contient une multitude de fonctions de questions contenant, ou non, du code à compléter pour nous guider. Il suffit de décommenter l'appel de fonction de la question qui nous intéresse afin de tester.

Le fichier `Configuration.py` contient la classe `Configuration` permettant de contenir et d'appliquer les paramètres souhaités pour la gestion de la caméra, d'OpenGL, des axes... Elle contient également la boucle infinie gérant la détection d'événements.

Lorsqu'on exécute la fonction `Q1a()` en ayant ajouté le retour d'un objet `Configuration()` on constate une fenêtre un peu plus grande que précédemment, avec nos axes (Capture 5). Il s'agit ici de paramètres de base de la fonction `configuration` qu'on peut retrouver dans le constructeur. Ces paramètres sont mis par défaut s'ils ne sont pas passés au constructeur.



Capture 5. Affichage de la configuration par défaut.

b)

On peut donc passer les différents paramètres par le constructeur grâce à un dictionnaire. Les clés de ce dernier représente le nom du paramètre et les valeurs des clés sont les valeurs des paramètres.

Il est également possible de modifier ces paramètres grâce aux setters. Ici il s'agit d'une fonction 'setParameter' qui prend en paramètre le nom du paramètre et sa nouvelle valeur. Il est également possible de chaîner ces appels de méthodes car la fonction 'setParameter' renvoie la référence de l'objet actuel : self.

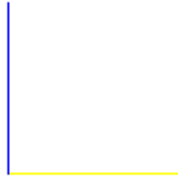
Le paramètre 'screenPosition' demande un traitement supplémentaire car il est nécessaire que PyOpenGL recalcule la matrice de transformation.



Capture 6. Affichage des axes avec les nouveaux paramètres.

c)

Ici même principe que dans la préparation, on ajoute une rotation de -90° sur l'axe X. On voit bien sur la capture 7 que l'axe cyan (précédemment sur Y) est remplacé par l'axe bleu (précédemment sur Z) tout comme dans la préparation du TP.



Capture 7. Affichage des axes après la rotation

II. Mise en place des interactions avec l'utilisateur avec Pygame

1) d)

On ajoute deux conditions (Code 5) à la fonction 'processKeyDownEvent' de la classe configuration. Ces conditions vont nous permettre de détecter si l'événement actuel est la pression de Page Down ou Page Up. On appelle ensuite la fonction glScalef afin d'ajuster le "zoom" en fonction de la touche pressée.

```
elif self.event.key == pygame.K_PAGEDOWN:
    gl.glScalef(1 / 1.1, 1 / 1.1, 1 / 1.1)
elif self.event.key == pygame.K_PAGEUP:
    gl.glScalef(1.1, 1.1, 1.1)
```

Code 5. Conditions permettant de (dé)zoomer avec les touches Page Up / Down.

e)

Ici même principe que la question précédente sauf que le type d'événement est maintenant un 'button' et non plus une 'key'.

```
def processMouseButtonDownEvent(self):
    # Use mouse wheel to zoom out / in
    if self.event.button == pygame.BUTTON_WHEELDOWN:
        gl.glScalef(1 / 1.1, 1 / 1.1, 1 / 1.1)
    elif self.event.button == pygame.BUTTON_WHEELUP:
        gl.glScalef(1.1, 1.1, 1.1)
```

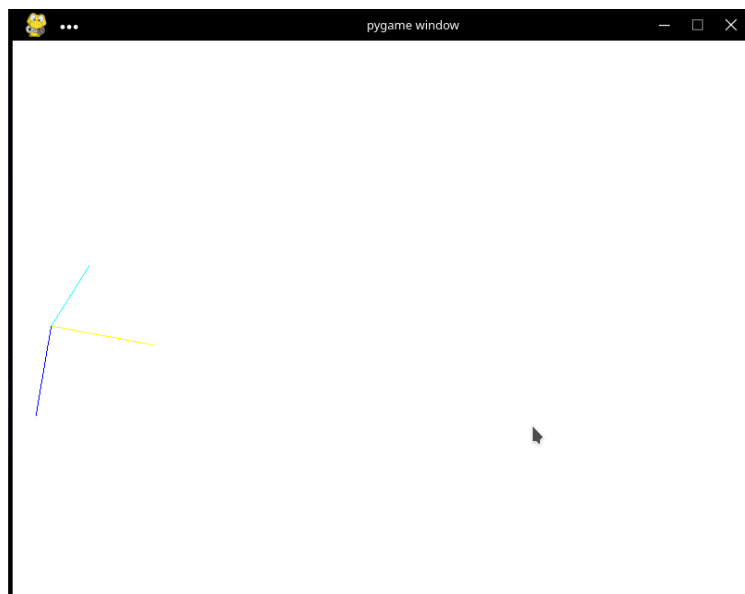
Code 6. Fonction permettant de (dé)zoomer avec la molette.

f)

Le déplacement des objets est implémenté dans la fonction 'processMouseEvent' (Code 7). On utilise les mêmes méthodes utilisées précédemment : 'glRotatef' et 'glTranslatef' afin de déplacer les objets.

```
def processMouseEvent(self):  
    # Clic gauche de la souris pressé  
    if pygame.mouse.get_pressed(3)[0]:  
        # Rotation sur l'axe Z en fonction de la distance  
        # parcourue par la souris sur l'axe X  
        gl.glRotatef(self.event.rel[0], 0.0, 0.0, 1.1)  
        # Rotation sur l'axe X en fonction de la distance  
        # parcourue par la souris sur l'axe Y  
        gl.glRotatef(self.event.rel[1], 1.0, 0.0, 0.0)  
    # Clic droit de la souris pressé  
    elif pygame.mouse.get_pressed(3)[2]:  
        # Translation sur l'axe X en fonction de la distance  
        # parcourue par la souris sur l'axe X  
        gl.glTranslatef(self.event.rel[0] * 0.1, 0.0, 0.0)  
        # Translation sur l'axe Z en fonction de la distance  
        # parcourue par la souris sur l'axe Y  
        gl.glTranslatef(0.0, 0.0, -self.event.rel[1] * 0.1)
```

Code 7. Fonction permettant le “déplacement” d’objets



Capture 8. Exemple de translation et de rotation sur nos axes.

III. Création d'une section

2) a)

On commence par nommer chaque sommet du parallélépipède (Schéma 1).

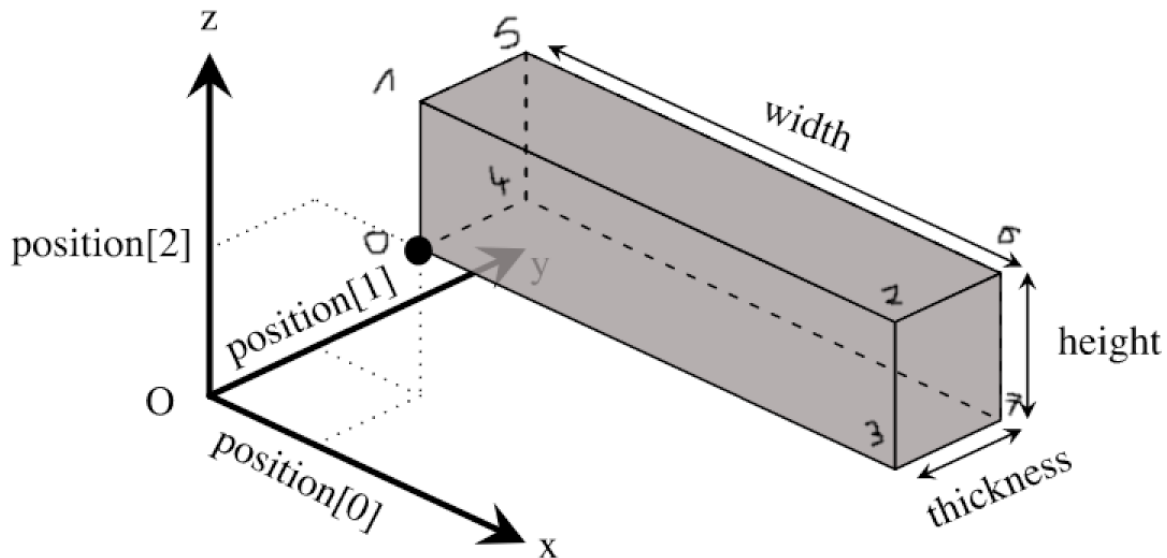


Schéma 1. Nommage des sommets du parallélépipède.

On peut donc plus facilement définir chaque points contenus dans l'attribut vertices. Ensuite on définit les faces en reliant les sommets entre eux (Code 9).

```
self.vertices = [
    [0, 0, 0], # 0
    [0, 0, self.parameters['height']], # 1
    [self.parameters['width'], 0, self.parameters['height']], # 2
    [self.parameters['width'], 0, 0], # 3
    [0, self.parameters['thickness'], 0], # 4
    [0, self.parameters['thickness'], self.parameters['height']], # 5
    [self.parameters['width'], self.parameters['thickness'], self.parameters['height']], # 6
    [self.parameters['width'], self.parameters['thickness'], 0], # 7
]

self.faces = [
    [0, 1, 2, 3], # Front
    [1, 5, 6, 2], # Top
    [5, 6, 7, 4], # Back
    [4, 7, 3, 0], # Bottom
    [0, 1, 5, 4], # Left
    [2, 6, 7, 3] # Right
]
```

Code 9. Définition des sommets et des faces.

b)

Afin de dessiner les différentes faces nous implémentons la méthode 'draw' de la classe 'Section' (Code 10). On commence par réaliser le glPushMatrix pour sauvegarder la matrice actuelle. Ensuite, on appelle la fonction glBegin pour commencer le dessin. La première boucle for parcourt les faces puis la seconde parcourt les points de cette dernière.

Ainsi on peut initialiser nos vertex et leur couleur. Nous avons ajouté un indice *i* incrémenté à chaque tour de boucle afin de faire varier la couleur de notre parallélépipède (Capture 9).

```
def draw(self):
    gl.glPushMatrix()

    if self.parameters['orientation'] != 0:
        gl.glRotate(self.parameters['orientation'], 0.0, 0.0, 1.0)

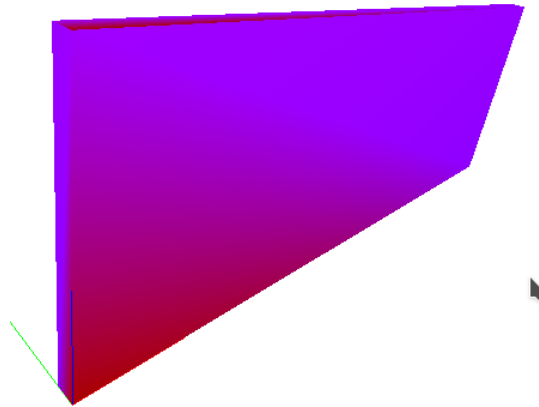
    gl.glPolygonMode(gl.GL_FRONT_AND_BACK, gl.GL_FILL) # on trace les faces : GL_FILL
    gl.glBegin(gl.GL_QUADS) # Tracé d'un quadrilatère

    for face in self.faces:
        for i, point in enumerate(face):
            gl.glColor3fv([10 / (i + 1 * 15), 0.0, 1 * i * 5])
            gl.glVertex3fv(self.vertices[point])

    gl.glEnd()

    gl.glPopMatrix()
```

Code 10. Fonction permettant l'affichage de sections.



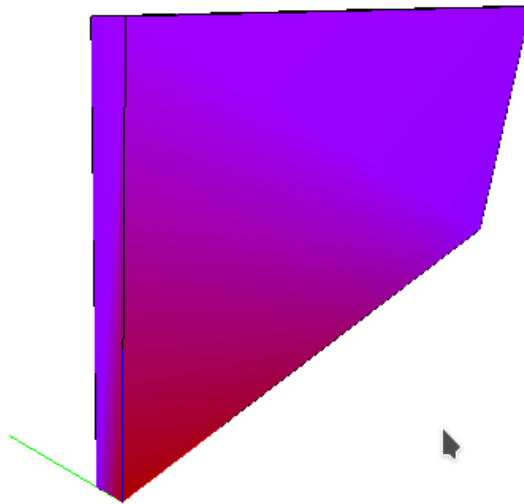
Capture 9. Magnifique parallélépipède.

c)

Pour afficher les arrête, même principe qu'à la question précédente sauf qu'on trace des lignes au lieu de polygones. Cependant il ne faut pas oublier de rajouter l'appel de la fonction 'drawEdges' au début de la fonction 'draw' (Code 11).

```
def draw(self):  
    if self.parameters['edges']:  
        self.drawEdges()  
    ...
```

Code 11. Ajout de l'appel de la fonction drawEdges



Capture 10. Affichage du parallélépipèdes avec ses arêtes.

IV. Création des murs

3) a)

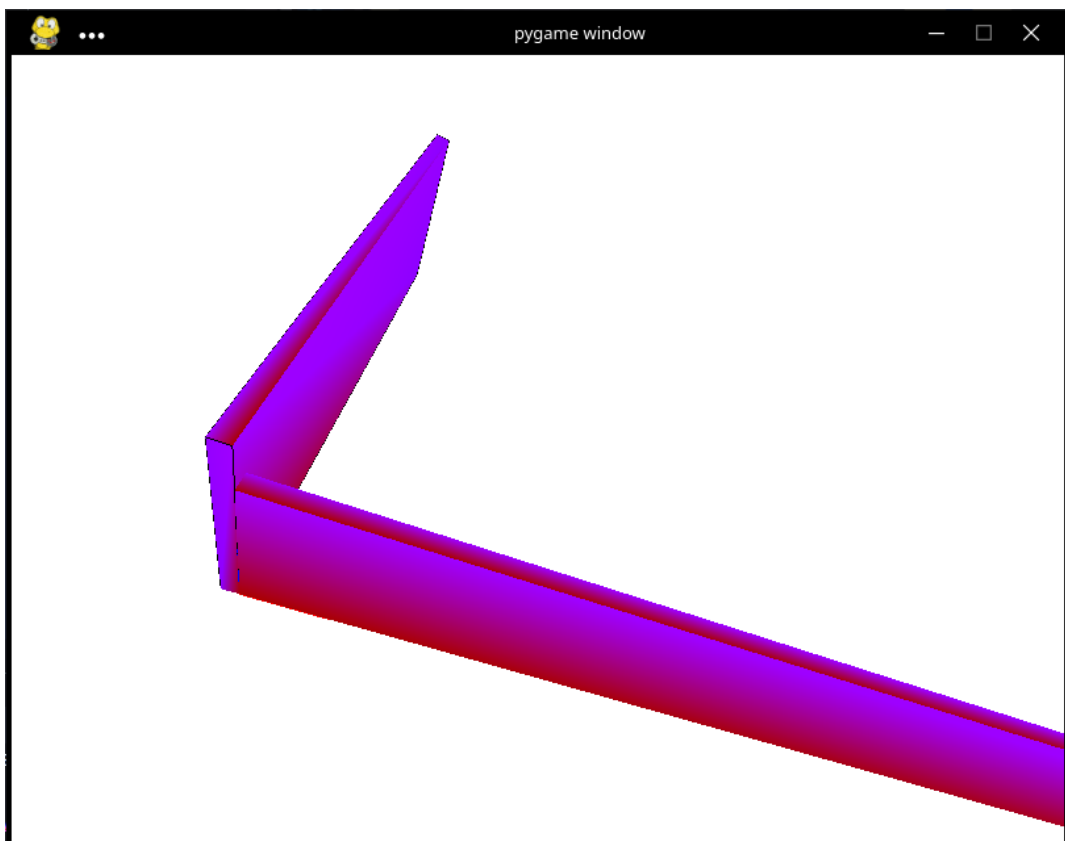
Le fichier Wall.py contient la classe 'Wall' constituée d'un constructeur avec une multitudes de paramètres avec des valeurs par défaut tout comme la classe 'Section'. Le constructeur instancie directement une section parente qui est en fait le mur initial. Cette dernière est ajoutée à la liste des objets de la classe 'Wall'. Pour implémenter la méthode 'draw' on se contente de parcourir la liste d'objet et d'appeler leur fonction 'draw'.



```
# Adds an object
def add(self, x):
    self.objects.append(x)
    return self

# Draws the faces
def draw(self):
    for section in self.objects:
        section.draw()
```

Code 12. Implémentation des méthodes 'add' et 'draw'



Capture 11. Affichage des murs

Tests :

Voici des captures d'écran des différents tests car nous avons remarqué que les tests ne passent pas depuis les actions github.

