Pyray et Spyder : Instructions

Linux

Dans le terminal, exécutez :

python3 -m venv ~/localdir/spyder-env

source ~/localdir/spyder-env/bin/activate

pip install numpy trimesh raylib

ensuite à chaque fois qu'un nouveau terminal est ouvert, il faut exécuter :

source ~/localdir/spyder-env/bin/activate

et pas des outres commandes.

Windows/ Mac OS

Installer miniconda :

https://docs.anaconda.com/miniconda/install/

Pyray

La bibliothèque pyray est une interface Python de la bibliothèque graphique Raylib, utilisée pour créer des applications graphiques en 2D et en 3D. Elle est idéale pour le développement de jeux et de simulations, permettant aux utilisateurs de gérer facilement les objets, les caméras, les couleurs et les animations dans un environnement interactif.

Pyray offre des fonctions simples pour dessiner des formes, des images, des modèles 3D, et pour détecter des collisions, ce qui en fait un bon choix pour les débutants comme pour ceux qui travaillent sur des projets rapides en graphique. Elle permet aussi de créer des animations et des scènes interactives, de sorte que les étudiants peuvent expérimenter avec des concepts comme les vecteurs, la perspective et la transformation d'objets.

Pour apprendre à utiliser pyray, vous pouvez consulter la documentation officielle de Raylib, qui décrit aussi bien les fonctions de base que les éléments avancés.

Liens :

1. https://electronstudio.github.io/raylib-python-cffi/pyray.html

Spyder

Spyder est un environnement de développement intégré (IDE) pour Python, conçu spécialement pour les scientifiques, les ingénieurs, et les étudiants en science des données. Il offre une interface intuitive et des fonctionnalités puissantes comme l'édition de code avec coloration syntaxique, l'exécution interactive de scripts, et des outils de débogage. L'interface est modulable, avec des panneaux comme l'Explorateur de variables, qui permet de visualiser et de manipuler les données et les objets en mémoire, ce qui facilite l'analyse et le développement de projets en Python.

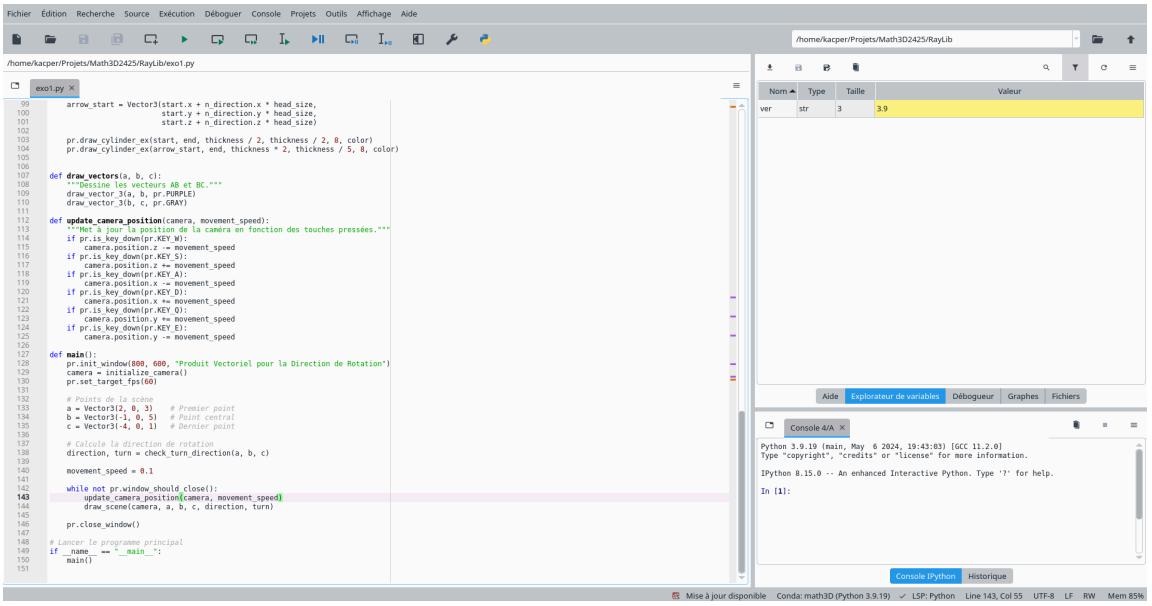


Fig 1. Fenêtre principale de Spyder

Exercice 1

Dans cet exercice, nous allons nous concentrer sur la mise en œuvre d'une calculatrice de base qui nous dira si nous détectons un virage sur un chemin défini par des vecteurs.

Le code de base est ici : lien vers le code

1. Trouvez la fonction : “dot_product” et implémentez le calcul du produit scalaire.
2. Trouvez la fonction “vector_length” et implémentez le calcul de la longueur d'un vecteur en utilisant le produit scalaire.
3. Trouvez la fonction “vector_normalize” et implémentez la fonction de normalisation du vecteur.

Une fois ces trois fonctions corrigées, le rendu devrait fonctionner correctement.

4. Trouvez la fonction : “cross_product” et implémentez le calcul du produit vectoriel entre deux vecteurs.

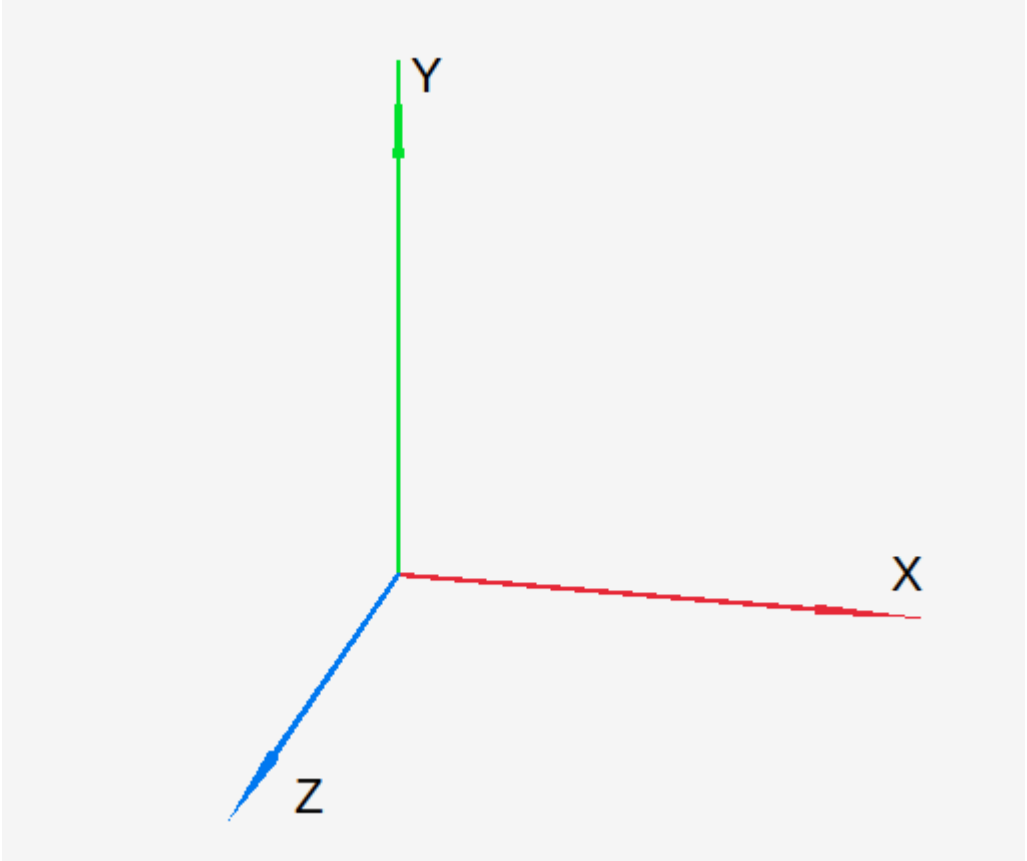
5. Trouvez la fonction : “check_turn_direction” et implémentez le contrôle entre deux paires de vecteurs définies par trois points d'entrée. Cette fonction doit retourner le résultat du produit vectoriel entre les deux vecteurs, ainsi qu'un texte informant sur le virage (horaire, antihoraire, collinéaire).
6.
7. 

Fig 2. Le système de coordonnées utilisé dans raylib. Notez qu'il s'agit d'un système de coordonnées de droite.

Exercice 2 : Le labyrinthe

1. Décommentez la ligne 189 : “points = generate_maze_path(50, int(grid_size/2), 1.0, False)”. Si tout est bien implémenté jusqu'à présent, vous devriez voir quelque chose de similaire au chemin montré dans la Fig. 3. Dans cet exercice, vous allez implémenter une fonction qui peut analyser une liste de points et indiquer quand des virages à gauche et à droite se produisent.
2. Modifiez la fonction de dessin et affichez les résultats du produit vectoriel pour l'ensemble du chemin. Vous pouvez également imprimer la sortie dans la fenêtre de ligne de commande.

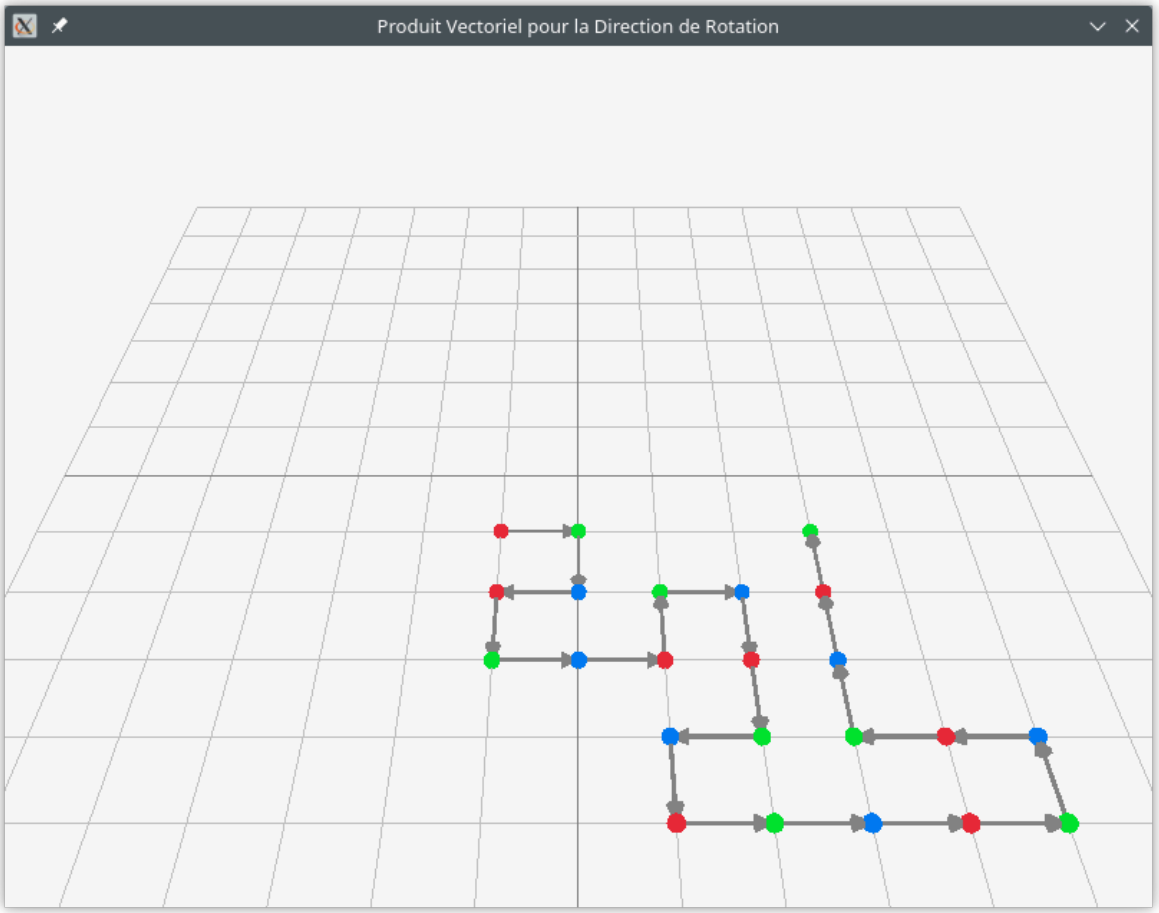


Fig 3. Labyrinthe généré aléatoirement

Bonus !

Modifiez la ligne suivante :

```
points = generate_maze_path(50, int(grid_size/2), 1.0, False)
```

par :

```
points = generate_maze_path(50, int(grid_size/2), 1.0, True)
```

Maintenant, le labyrinthe est généré en 3D. Pouvez-vous implémenter un moyen de détecter le changement de chemin dans le labyrinthe 3D ?

Exercice 3

Le code de base est ici : lien vers le code

Implémentez la fonction “is_point_in_fov” et testez-la sur un certain nombre d'images. Collectez des captures d'écran pour tous les exemples.

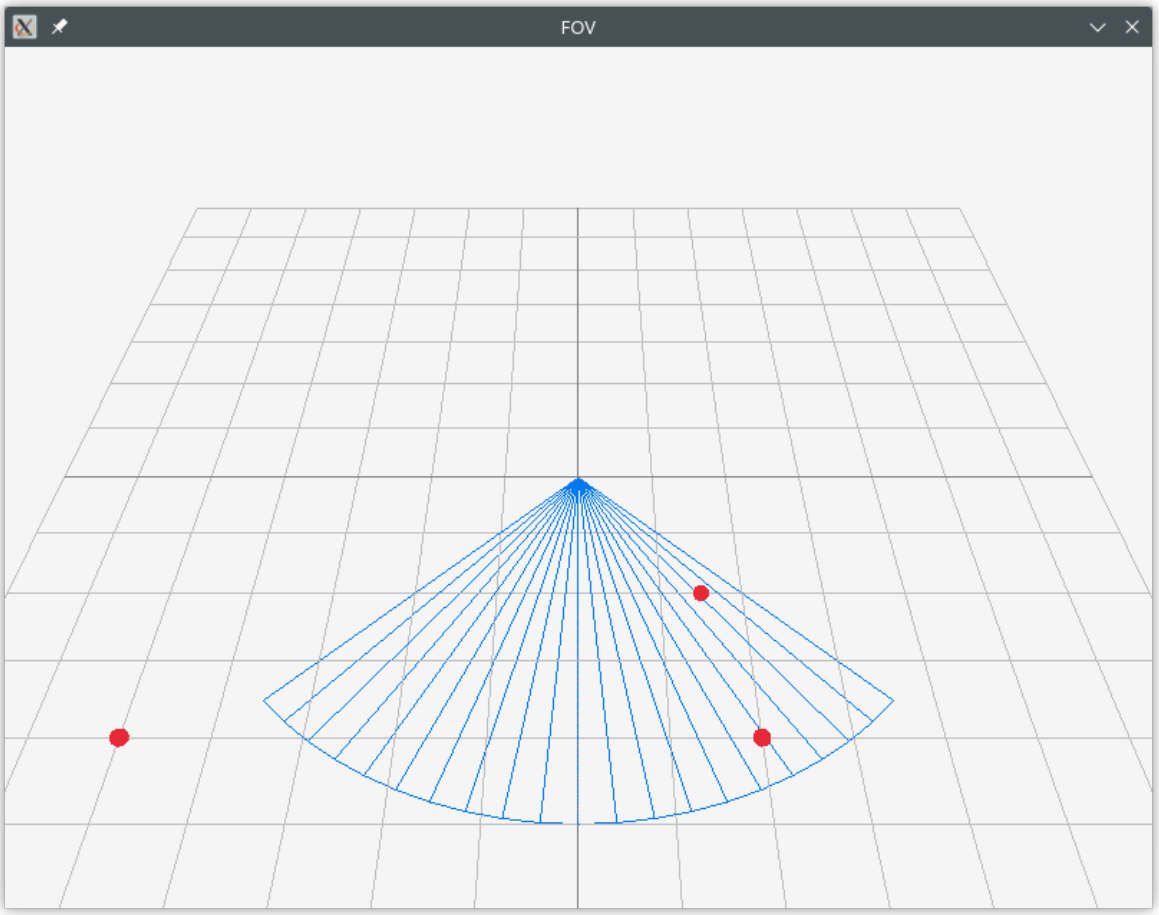


Fig. 4. Visualisation du FOV et de quelques points.

Exercice 4

Utilisez le code créé jusqu'à présent et implémentez un code qui prend deux vecteurs 3D, puis visualisez le parallélogramme défini par ceux-ci et génère sa surface.

Exercice 5

Dans cet exercice, nous allons travailler avec des maillages polyédriques en 3D, représentés par des sommets, des faces et des arêtes. Comme illustré dans la Fig. 4 :

- Les points rouges représentent les sommets.

- Les polygones gris représentent les faces.
- Les segments de ligne noirs représentent les arêtes.

L'objectif de cet exercice est de calculer les vecteurs normaux pour les faces et les sommets. Pour calculer un vecteur normal pour chaque face, suivez ces étapes :

1. Choisissez deux vecteurs consécutifs **a** et **b,** le long des arêtes d'une face.
2. Calculez le vecteur normal **n,** en utilisant le produit vectoriel
   n = **a** x **b** / ||**a** x **b**||

Pour afficher les vecteurs normaux sur le maillage :

1. Déterminez le centre de gravité de chaque face en faisant la moyenne des coordonnées des sommets qui définissent la face.
2. Tracez le vecteur normal unitaire **n** au centre de gravité de chaque face.

Pour les sommets, le vecteur normal est calculé comme la moyenne des vecteurs normaux des faces adjacentes.

Le code de base est ici : lien vers le code et le fichier du maillage

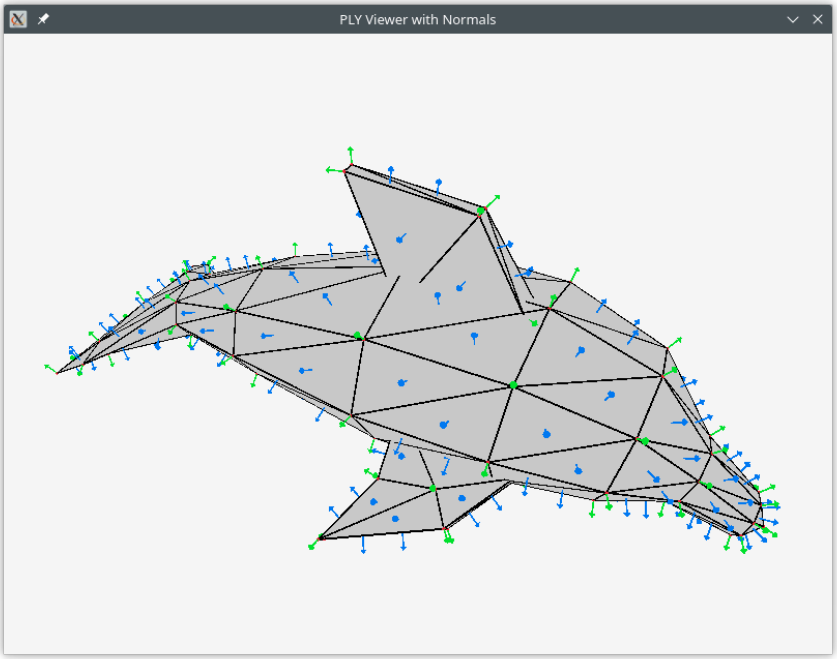


Fig. 5. Visualisation d'un maillage avec des vecteurs normaux calculés par face et sommet.