

Traitemet et Synthèse d'Images Projet 2103 Racer (Jeu OpenGL):

Sommaire:

I) Introduction	2
II) Objectifs du jeu	2
III) Commandes	3
Vaisseau	3
View Mode	3
Menu Pause	3
Récapitulatif	3
IV) Mécaniques de Gameplay	4
Mise en place de la fenêtre, de la scène et de la caméra	4
Interactions entre l'utilisateur et le jeu	6
Mouvement du vaisseau et de la caméra en course	7
Système de checkpoints et de tours	8
Mode View Mode	9
Menu Pause	11
V) Physique	15
Système de collision	15
Système de vitesse, principes d'inertie et d'accélération	17
Système d'arrêt du véhicule	18
Synchronisation en temps réel	19
Synchronisation prenant en compte le menu pause	20
Système de vitesse pour les véhicules	20
VI) Traitement des Données	21
Configuration des statistiques pour les véhicules	21
Configuration des paramètres du jeu	23
Configuration du chargement de la carte	24
VII) Interfaces	29
Système d'affichage de données à l'écran	29
Système de cinématique de fin de course	30
VIII) Conclusion:	31

I) Introduction

Plongez-vous en 2103 pour vous préparer au Grand Prix de course automobile! Un Grand Prix c'est comme un partie de TSI, ça se prépare, donc entraînez-vous dans une première version du jeu *2103 Racer* à bord de l'un des 3 véhicules sur l'un des 5 circuits pour battre des temps records!

Ce jeu est codé en Python et utilise OpenGL pour l'affichage graphique. Nous avons commencé avec des ressources fournies par l'E-Campus de CPE Lyon, qui comprenaient une scène, une caméra, des déplacements et un éclairage de base, que nous avons modifiés pour créer ce jeu de course.

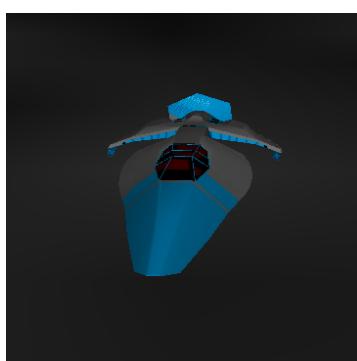
II) Objectifs du jeu

Au lancement du programme, vous serez placé sur un circuit à bord d'un vaisseau, tous deux configurés dans le fichier texte *Data.txt*. Pour changer votre configuration sur le vaisseau, le circuit ou de manière générale un paramètre du jeu au lancement de celui-ci, veuillez le faire ici.

Vous devrez compléter un certain nombre de tours en respectant le tracé du circuit en déplaçant votre vaisseau. Essayez de le compléter le plus rapidement possible pour tenter de devenir le vrai roi de la course!

Véhicules disponibles

AG-SYS



Feiser

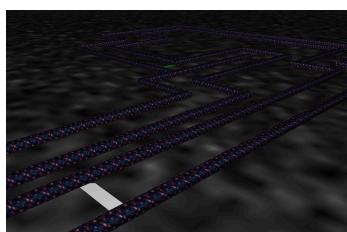


EGREQ4303



Circuits disponibles

Metropolis



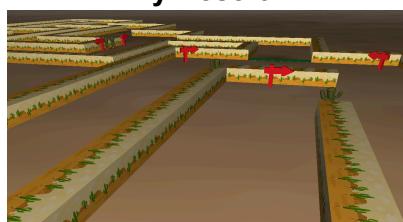
Eight



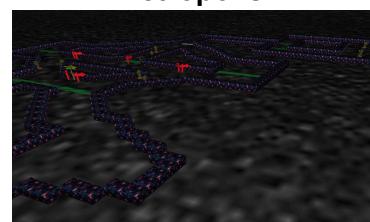
Jungle Forest



Dry Desert



Metropolis 2



III) Commandes

Vaisseau

Déplacez votre vaisseau à l'aide des flèches directionnelles. Le déplacement se fait au sol et vous devez maintenir l'entrée pour continuer à vous déplacer dans cette direction.

View Mode

Si vous avez activé la possibilité de passer en *View Mode* en mettant *Free Camera Mode* à *True* dans le *Data.txt*, vous pouvez enclencher ce mode à tout moment (hors menu Pause) en appuyant sur la touche *V*. Dans ce mode, vous pouvez déplacer la caméra avec les touches *ZQSD*, tourner la caméra avec les touches *IJKL* et à tout moment passer en mode cinématique en cachant toutes interfaces utilisateurs avec la touche *T*. Vous pouvez toujours déplacer votre vaisseau. Réappuyer sur *V* vous recentra automatiquement sur le vaisseau et vous fera quitter le mode cinématique.

Menu Pause

Appuyer sur le bouton *P* pour mettre en Pause le jeu. Dans ce menu, vous pouvez bien entendu continuer votre course, mais également, configurer votre véhicule ou le circuit que vous souhaitez. En confirmant la régénération vous apparaîtrez dans le nouveau circuit à bord de votre nouveau véhicule. Pour ce faire, déplacez-vous dans le menu avec les flèches haut et bas, et appuyez sur droite et gauche pour vous déplacer dans les sous-menus (pour choisir votre véhicule, votre course ou la confirmation de génération du circuit). Appuyez sur Entrée pour confirmer l'action du sous-menu (vous devez appuyer sur Entrée pour confirmer la sélection d'un véhicule ou d'un circuit). Dans la sélection du Circuit, appuyez sur *M* (resp. *V*) pour activer/désactiver le mode Miroir (resp. Reverse). Vous pouvez à tout moment quitter le menu pause en appuyant sur *P*.

Récapitulatif



- ↑ : Avancer / Aller au menu précédent
- ↓ : Reculer / Aller au menu suivant
- → : Tourner à droite / Aller au sous-menu suivant
- ← : Tourner à gauche / Aller au sous menu précédent
- P : Activer/Désactiver le menu Pause
- V : Activer/Désactiver le View Mode
- Z : Avancer la caméra
- S : Reculer la caméra
- Q : Déplacer à gauche la caméra
- D : Déplacer à droite la caméra
- I : Tourner en haut la caméra
- K : Tourner en bas la caméra
- J : Tourner à gauche la caméra
- L : Tourner à droite la caméra
- T : Passer en mode Cinématique
- Entrée: Valider une sélection dans le menu
- M: Activer/Désactiver Mode Miroir (seulement dans le sous-menu Course)
- R: Activer/Désactiver Mode Reverse (seulement dans le sous-menu Course)

Légende: Instruction: Seulement dans le mode *View Mode*

Instruction: Seulement dans le menu *Pause*

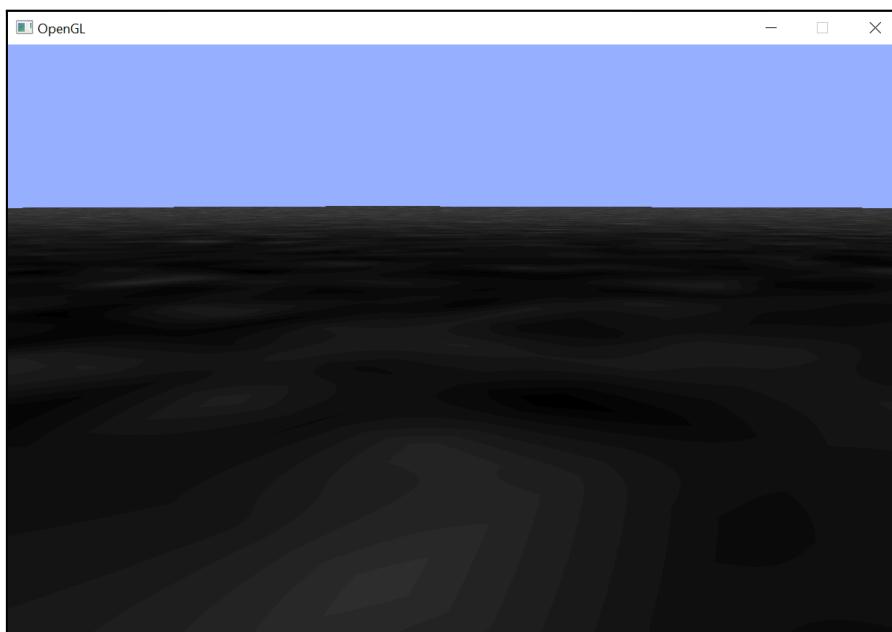
IV) Mécaniques de Gameplay

Mise en place de la fenêtre, de la scène et de la caméra

Pour commencer, il est essentiel de définir la scène de jeu dans OpenGL. Nous définissons la fenêtre, le contexte, la caméra, les programmes et les données via un objet de classe `ViewerGL`. Nous le ferons à travers un objet de classe `ViewerGL` qui va contenir l'essentiel de notre programme.

On définit notre fenêtre puis notre contexte avec la bibliothèque `glfw` (on définit ici la taille de la fenêtre `1200px800p`, la couleur de fond, ici en RGB normalisé (0.6,0.7,1.0) soit [cette couleur](#)).

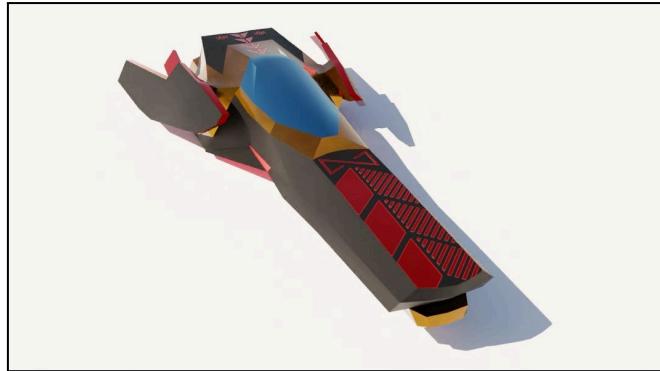
Il est maintenant temps de configurer la scène. Tout d'abord, on définit notre **Caméra** en lançant une fonction qui va créer un objet de classe `Camera` dans le fichier `cpe3d.py`. Cela va permettre de définir une **transformation** et une **projection**. La transformation correspond aux caractéristiques de notre objet ici considéré en 3D, on fait appel à la classe `Transformation3D` qui va permettre de définir ses **caractéristiques de transformation** qui sont: la **translation** (coordonnées “xyz”), des **angles de rotations d’euler** et le **centre de rotation**. La projection est une matrice de protection créée à partir de la bibliothèque `pyrr`. On initialise ensuite les différents programmes que nous utiliserons lors du projet grâce au fichier importé `glutils.py`. Nous utiliserons un programme `program3d_id` pour les objets en 3D dans la scène avec les fichiers `shader.vert` et `shader.frag`, `programGUI_id` pour les objets type texte qui est comme son nom l’indique du GUI (Graphic User Interface) grâce aux fichiers `phong.vert` et `phong.frag`.



```
class Transformation3D:  
    def __init__(self, euler = pyrr.euler.create(), center = pyrr.Vector3(), translation = pyrr.Vector3()):  
        self.rotation_euler = euler.copy()  
        self.rotation_center = center.copy()  
        self.translation = translation.copy()
```

Par la suite on peut charger nos objets sous format *.obj* grâce au fichier *mesh.py* qui importe une classe *Mesh* pour tout objet qu'on souhaite importer. Cela nous donne le **maillage** de notre objet (avec les sommets et les caractéristiques du sommet), ce sont les *datas* qu'on doit exploiter pour l'afficher via OpenGL.

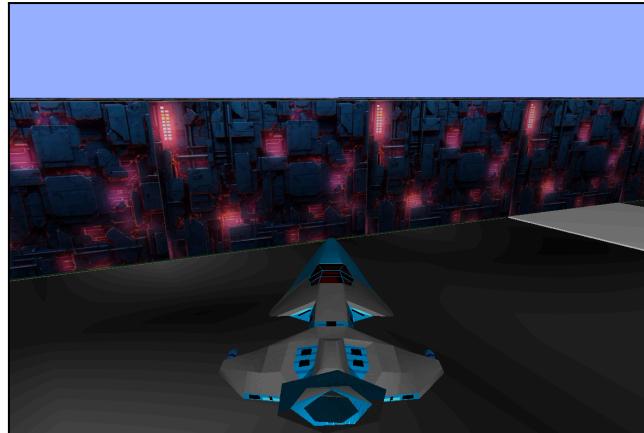
Après avoir normaliser ces caractéristiques, on peut ensuite changer sa taille en donnant une matrice en paramètre qui va allonger/rétrécir les proportions pour chaque vecteur directeur (par exemple une matrice [2,1,3] va multiplier la longueur de l'objet en x par 2, celle de y par 1 et celle de z par 3). Après cela on peut créer **un VAO pour chaque fichier .obj** qu'on importe (si on utilise le même objet plusieurs fois, on utilisera le même VAO).



```
m = Mesh.load_obj(Ship_file)           TristanP, 2 weeks ago • Fix Camera + Text + Pause V1
m.normalize()
m.apply_matrix(pyrr.matrix44.create_from_scale([self.ship_size_x, self.ship_size_y, self.ship_size_z, 1]))
tr = Transformation3D()
tr.translation.y = -np.amin(m.vertices, axis=0)[1]
tr.translation.x = ship_spawn_location[0][0]
tr.translation.z = ship_spawn_location[0][1]
tr.rotation_center.z = 0.2
tr.rotation_euler[pyrr.euler.index().yaw] = ship_spawn_location[0][2]
texture = glutils.load_texture('textures/'+self.ship+'/Ship.png')
o = Object3D(m.load_to_gpu(), m.get_nb_triangles(), program3d_id, texture, tr)
```

On peut enfin définir les caractéristiques de chaque objet 3D comme la caméra en réglant ses coordonnées, ses angles d'euler et son centre de rotation. Après avoir charger la texture via *glutils.py*, on peut ajouter notre objet 3D, grâce à la classe de *ObjectCube* de *cpe3d.py*, qui va pouvoir être dessiné par le GPU après projection grâce à *program3d_id* et ses caractéristiques de transformation (qui va être traduit par une variable uniforme envoyé au vertex shader et fragment shader).

Pour les objets GUI textes, il suffit de définir le texte à afficher, les coordonnées du coin inférieur gauche et supérieur droit, appliquer la texture qui correspond à une image avec tous les caractères d'une police d'écriture et utiliser un VAO pour le texte. Ce VAO s'initialise avec la classe *Text* de *cpe3.py* en initialisant une **géométrie rectangulaire** (avec 2 triangles) sur $[0;1]^2$ (pour les coordonnées envoyées au fragment shader, soit $[-1;1]^2$ dans notre fenêtre). Enfin, on peut ajouter un objet texte qui sera de classe *Text* et qui peut être dessiné par le GPU grâce à *programGUI_id*.



Chaque objet de notre scène a une caractéristique *visible* qui quand passée à *False* permet d'empêcher le GPU de dessiner l'objet en question.

Nous stockerons tous les objets dans une liste *objs* propre à notre objet *ViewerGL* et tous les objets textes dans un dictionnaire avec comme clé la signification du texte et en valeur l'objet *Texte*.

Une fois que la scène est mise en place, nous pouvons générer le circuit et lancer la fonction *run* du *ViewerGL* pour faire une boucle infini où le GPU va effacer l'écran et redessiner chaque objet.

```
for obj in self.objs:
    try:
        GL.glUseProgram(obj.program)
        if isinstance(obj, Object3D):
            self.update_camera(obj.program)
            obj.draw()
    except AttributeError:
        pass
```

Interactions entre l'utilisateur et le jeu

Nous codons les interactions initiales entre l'utilisateur et le contexte *OpenGL* en utilisant la fonction *poll_events* de *glfw* pour capturer les événements de clavier. La bibliothèque *glfw* a une fonction *poll_events* qui permet de récupérer les événements comme les touches claviers enclenchées. Il est possible d'envoyer ces informations dans une méthode de notre *ViewerGL* qu'on nommera *key_callback* après avoir configuré avec la fonction *set_key_callback* de *glfw*. Les informations récupérées sont la touche du clavier et l'action. Si l'action est à 0, la touche n'est pas enclenchée, si elle est à 1, elle est enclenchée et si elle est à 2 elle est enfoncee. On enregistre cette action dans un dictionnaire *touch* avec comme clé la touche du clavier.

```
def key_callback(self, win, key, scancode, action, mods):
    # sortie du programme si appui sur la touche 'échappement'
    if key == glfw.KEY_ESCAPE and action == glfw.PRESS:
        glfw.set_window_should_close(win, glfw.TRUE)
    self.touch[key] = action
```

Mouvement du vaisseau et de la caméra en course

Grâce aux interactions capturées, nous pouvons **déplacer le véhicule** avec les flèches directionnelles. Le déplacement du vaisseau est réalisé en appliquant une matrice de rotation à un vecteur de translation en fonction des angles d'Euler du vaisseau. Pour cela dans la boucle d'affichage on fait appel à la fonction *update_key* et on vérifie l'état des entrées des flèches directionnelles dans la liste *touch* (si les clés existent et si l'action est bien celle où on appuie sur le bouton). Pour avoir le nom de la clé, il suffit de mettre *glfw.KEY_UP* par exemple pour la flèche du haut. On vérifie l'action en mettant *glfw.PRESS*.

Ensuite on peut changer la caractéristique **transformation** du vaisseau si ces conditions sont respectées. Si on appuie sur les flèches du côté, on augmente (ou diminue selon la direction) l'angle euler *yaw* d'un petit angle. À chaque boucle dans la boucle d'affichage, tant qu'on maintient la même direction, notre vaisseau tournera donc. Pour les translations avant et arrière, on calcule la vitesse de déplacement (cf Partie Physique *Système de Vitesse, Principe d'Inertie et Principe d'Accélération*) et on applique une matrice 3x3 à notre caractéristique transformation. Cette matrice se compose en 3 coordonnées [x, y, z].

Quand on veut aller tout droit, c'est comme si on voulait mettre [0,0,v] avec v notre vitesse. Cependant nous voulons **aller tout droit quelle que soit la direction**. Grâce à la bibliothèque *pyrr* et la sous-bibliothèque *matrix33*, nous pouvons faire appliquer une **rotation à notre vecteur idéal** par produit matriciel avec un vecteur comprenant les angles d'euler de notre objet. Pour notre vaisseau, elles sont accessibles dans la liste *objs* à l'indice 0 car nous ferons en sorte que le **premier objet à charger soit le vaisseau**. Le vecteur résultant de cette opération est la matrice que nous devons ajouter à la translation de notre objet pour qu'il aille tout droit.

Une fois que nous avons fait ceci, nous avons fait bouger notre vaisseau, il faut désormais bouger la caméra. Pour cela rien de plus simple, on copie les caractéristiques du vaisseau (*translation, angles d'euler, centre de rotation*) en ajoutant une **matrice offset pour les translations** pour se trouver derrière le vaisseau et une rotation pour regarder dans la direction du vaisseau (avec l'angle *yaw*) avec une inclinaison verticale légère (avec l'angle *roll*). Pour ajouter du réalisme, cette matrice offset sera composée d'une matrice constante et on la multipliera par $1 + v/v_{max}$ ce qui donne un effet d'éloignement quand le vaisseau avance vite. Quand le vaisseau tourne, on rajoute un **angle offset sur yaw** pour donner l'impression qu'on tourne bel et bien.

La mise à jour de la translation et de l'angle doit absolument se faire avant que le GPU redessine tous les objets de la scène, donc il est préférable de le faire juste après le déplacement du véhicule.

```
if self.collision_rotation(-self.TurnSpeed_real_time) and not(self.race_finished):
    self.objs[0].transformation.rotation_euler[pyrr.euler.index().yaw] -= self.TurnSpeed_real_time
if self.camera_quick_follow_rotation and not(self.free_mode):
    self.cam.transformation.rotation_euler = self.objs[0].transformation.rotation_euler.copy()
    self.cam.transformation.rotation_euler[pyrr.euler.index().yaw] += np.pi + self.camera_turning_offset_quick_follow
    self.cam.transformation.rotation_euler[pyrr.euler.index().roll] += np.pi/10
    self.update_checkpoint_lap()

glfw.KEY_RIGHT in self.touch and self.touch[glfw.KEY_RIGHT] > 0:
```

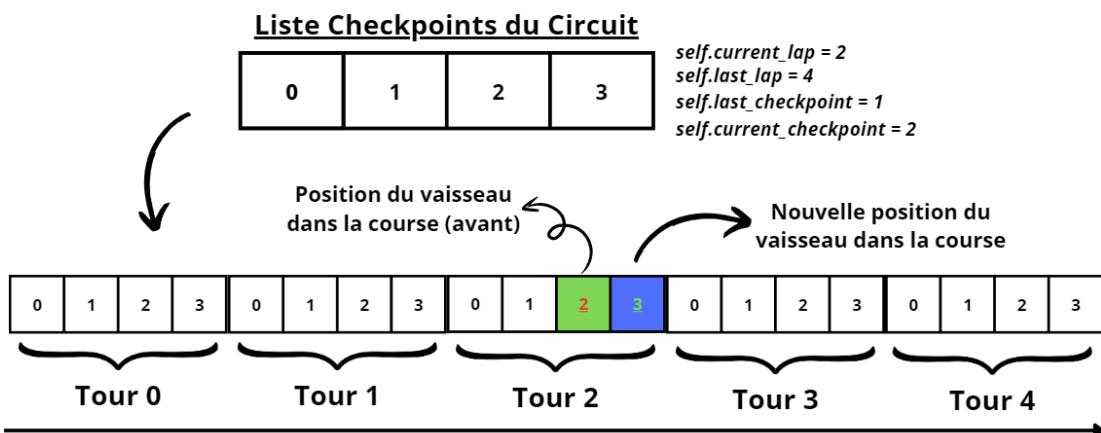
Système de checkpoints et de tours

Le circuit comporte des **checkpoints invisibles** que le vaisseau doit traverser dans un ordre séquentiel pour compléter un tour. Si le vaisseau traverse le checkpoint 0 (la ligne de départ) après avoir parcouru le dernier checkpoint (resp. premier checkpoint), le nombre de tour augmente (resp. diminue) jusqu'à la fin de la course (resp. jusqu'au tour 0).

Chaque circuit est configuré dans *ViewerGL* par une liste `self.checkpoint_list` contenant les checkpoints autres que celle de la ligne de départ et le nombre de tours nécessaires `self.last_lap` pour finir le circuit. On note le tour actuel dans `self.current_lap` et la section actuelle dans `self.last_checkpoint`.

On importe le fichier *checkpoints.py* pour gérer le système de checkpoint. On crée une liste de longueur $L = (\text{self.last_lap} + 1) \times \text{len}(\text{self.checkpoint_list})$ dans le but de représenter l'ensemble des checkpoints à parcourir entre le tour 0 et le tour 3 (par défaut le joueur commence au tour 1 mais rien ne vous empêche de faire le tour à l'envers si vous le souhaitez).

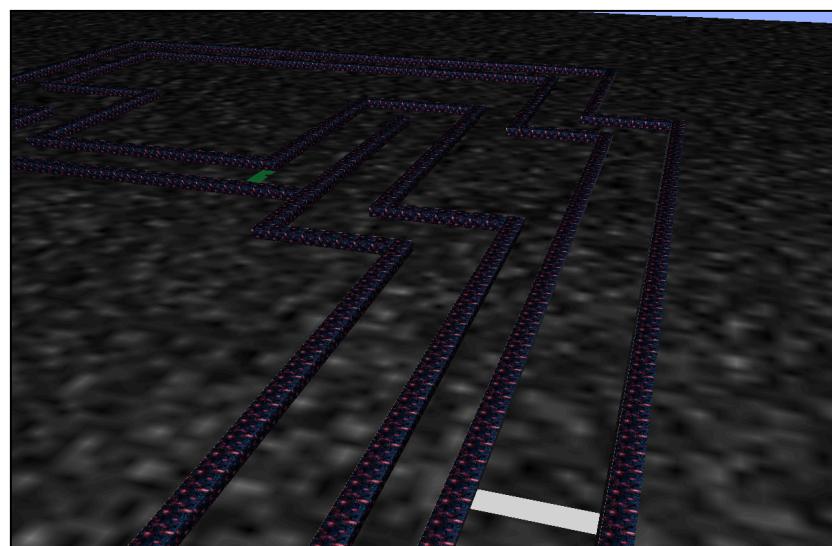
On sait alors l'avancement du joueur dans la course car le dernier checkpoint qu'il a parcouru se situe à l'indice `self.current_lap` \times `len(self.checkpoint_list)` + `self.last_checkpoint`.



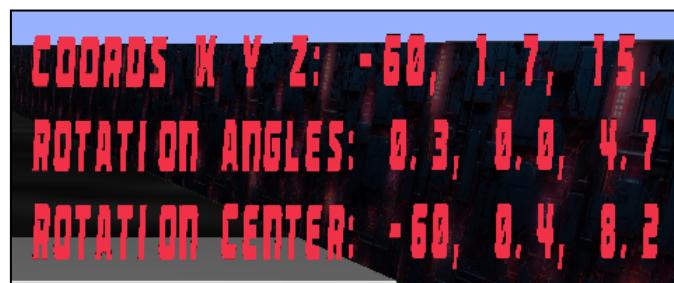
Quand on rentrera en collision avec un checkpoint, il faudra regarder ce qu'on fait: si on arrive bien au checkpoint suivant ou précédent, si on augmente ou diminue le numéro de la section ou également le nombre de tours. La fonction `checkpoint_check_system` prend en paramètre les valeurs nécessaires à la création de cette liste et le checkpoint qui a été entré en collision avec le vaisseau `current_checkpoint`. On regarde si en incrémentant (resp. diminuant) de 1 l'indice de notre avancement dans la course, on tombe sur ce checkpoint, si oui, on actualise le `last_checkpoint` par `current_checkpoint` et si `current_checkpoint` est égal à 0, on augmente (resp. diminue) le nombre de tours `current_lap`. On ne fait rien si `current_lap` est supérieur à `last_lap` et on ne regarde pas si on parcours le circuit à l'envers si le vaisseau est au tour 0 et au checkpoint 0 (on n'actualise pas ces valeurs, à part si on parcourt le circuit dans le bon sens).

Mode View Mode

En passant la valeur *Free Camera Mode* à *True* dans le *Data.txt*, il est possible d'appuyer sur la touche *V* pour que les **caractéristiques transformation** de la caméra ne dépendent plus de celle du vaisseau, c'est-à-dire qu'on ne fait plus la synchronisation comme expliqué dans le déplacement du vaisseau en course, grâce à une valeur *self.free_mode* qui indique si on est dans ce mode. Dans ce mode uniquement, de nouvelles touches de claviers sont prises en compte et permettent le déplacement de la caméra. Pour les mouvements de translation avant et arrière et les déplacements verticaux, elles sont similaires à celui du vaisseau, on applique un vecteur de déplacement, auquel on appliquera d'abord une rotation en fonction des angles d'euler, avant d'effectuer le déplacement. Cependant pour les déplacements latéraux, on recopie les angles d'euler de la caméra mais on passe **l'angle roll à 0** pour que le déplacement se fasse sans que la caméra se déplace verticalement. Pour les touches de rotation, on incrémente ou diminue **l'angle yaw** pour les rotations Gauche/Droite et **l'angle roll** pour les rotations Haut/Bas.



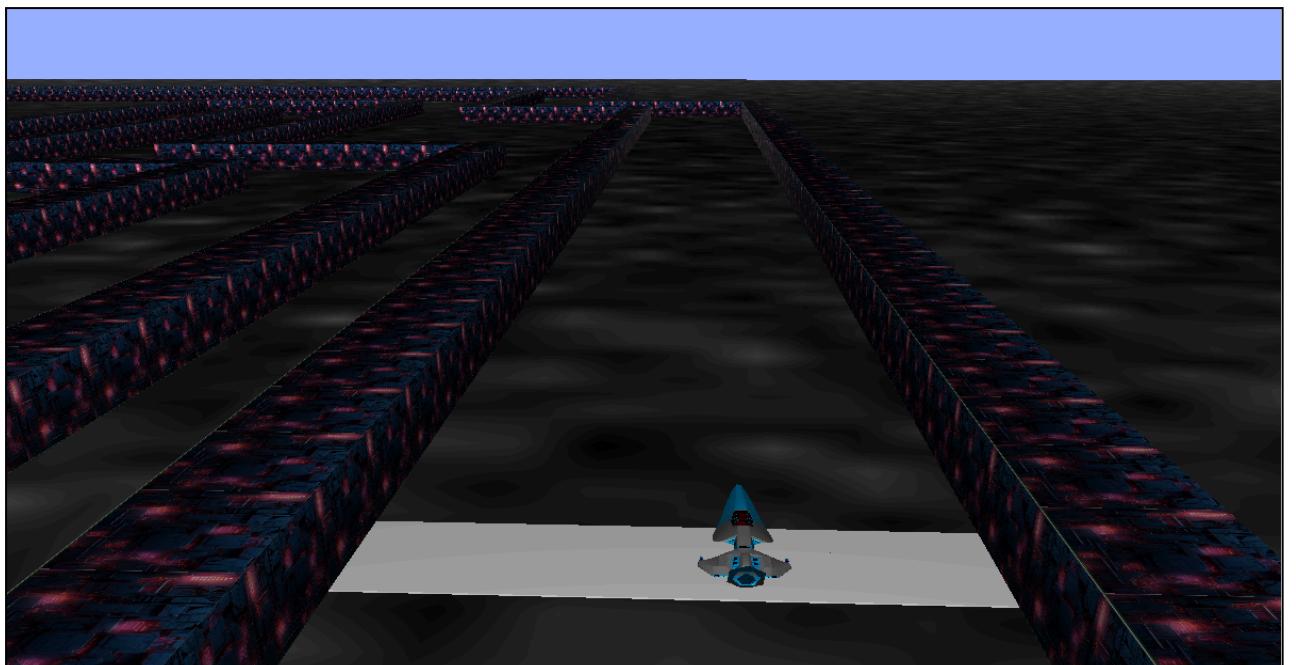
Quand on enclenche ce mode, on fait apparaître des nouveaux objets textes qui affichent les caractéristiques transformation de la caméra. Pour cela, quand on appuie sur la touche *V*, on fait passer *self.free_mode* à *not(self.free_mode)*. Si la valeur passe à *True*, alors on passe les caractéristiques **visible** des textes à *True* (elles sont donc initialisées à *False*). Cette caractéristique intervient dans la **fonction draw** de l'objet et exécute les lignes permettant l'affichage seulement si la caractéristique est à *True*. Si *self.free_mode* passe à *False* (en quittant), on fait disparaître ces textes.



Enfin, il est possible de passer en **mode cinématique** en cachant tous les objets textes de l'écran en appuyant sur *T*. En appuyant sur ce bouton, si on passe en mode cinématique, on parcourt tous les objets textes, on enregistre dans un dictionnaire *self.text_list_visible_copy* la clé et la valeur actuelle de la caractéristique *visible* et on passe à *False* cette caractéristique dans l'objet texte. Si on appuie à nouveau sur *T* ou si on appuie sur *V* pour quitter le mode *View Mode*, on quitte le mode cinématique et on parcourt tous les objets textes en remettant les caractéristiques à ce qu'elles étaient avant de passer en mode cinématique. On définit aussi *self.free_mode_input* et *self.text_display_input* qui correspondent aux derniers instants où on a respectivement appuyé sur la touche *V* et la touche *T* pour qu'on n'active ou désactive ces modes toutes les 0.2s.

```
if glfw.KEY_T in self.touch and self.free_mode:           TristanP, 2 weeks ago • Data+Machine file texts + Free View Mode
    if self.touch[glfw.KEY_T] and (glfw.get_time()-self.text_display_input - self.timer_dict["text_display_input"])>0.2: # Remove Text
        if self.show_text:
            self.text_list_visible_copy = {}
            for meaning in self.text_dict.keys():
                text = self.text_dict[meaning]
                self.text_list_visible_copy[meaning]=text.visible
                if meaning != "Finish" and meaning != "Pause":
                    text.visible = False
        else:
            for meaning in self.text_dict.keys():
                text = self.text_dict[meaning]
                if meaning != "Finish" and meaning != "Pause":
                    text.visible = self.text_list_visible_copy[meaning]

    self.show_text = not(self.show_text)
    self.text_display_input = glfw.get_time()
    self.timer_dict["text_display_input"] = 0
```



Menu Pause

Il est possible à tout moment d'appuyer sur le bouton *P* pour mettre en Pause le jeu. Cela veut dire qu'on passe un paramètre *self.pause* à *not(self.pause)* (initialement à *False*). Si la valeur passe à *True* alors on n'effectue plus les méthodes et fonctions qui sont liées au déroulement du jeu (déplacement, rotation, collision...) et on exécute de nouvelles méthodes qui sont liées au menu Pause. On effectue le test de l'action du bouton *P* dans la boucle d'affichage en permanence.

```
self.check_pause()          TristanP, 2 weeks ago •

if not self.pause:
    self.updating_text()
    self.update_key()
    #print(1/(glfw.get_time()-self.last_frame))
    self.last_frame = glfw.get_time()
    self.timer_dict["last_frame"] = 0
else:
    self.update_key_pause()
    self.update_display_list()
    self.menu_display()

for obj in self objs:
    try:
        GL.glUseProgram(obj.program)
        if isinstance(obj, Object3D):
            self.update_camera(obj.program)
        obj.draw()
    except AttributeError:
        pass
```

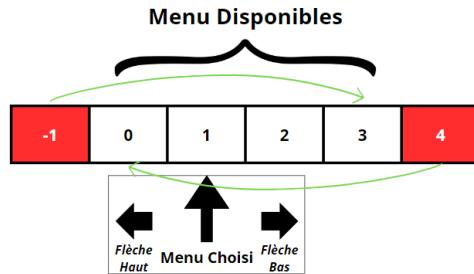
Dans le menu pause on peut utiliser les flèches haut et bas pour changer de menu parmis "Continue", "Ships", "Tracks" et "Restart" et les flèches droite et gauches pour changer de sous-menu comme pour la sélection du véhicule, du circuit ou la confirmation ou non de la régénération de la carte.



On stocke chaque menu dans un **dictionnaire** *self.menu_dict* avec comme clé le nom du menu et comme valeur la liste des sous menus dans le menu.

La flèche haut (resp. bas) incrémente (resp. diminue) la valeur `self.pause_menu_select` de 1. On regarde ensuite le nombre de menus et on passe la valeur par modulo du nombre de menus. C'est-à-dire que s'il y a 4 menus et que la valeur est à 4, elle passera à 0 car les menus vont de 0 à 3 (de même on passe de -1 à 3).

La flèche droite (resp. gauche) incrémente (resp. diminue) la valeur `self.pause_menu_select` de 1. On regarde ensuite le nombre de sous-menu selon le menu choisi et de la même manière on passe la valeur par modulo du nombre de sous-menu.



Le menu *Continue* n'a pas de sous-menu, le menu *Restart* contient les sous-menu *Yes* et *No*, enfin les menus *Ships* et *Tracks* contiennent la liste des circuits et vaisseaux disponibles (extraites respectivement de *Vehicles_Stats.csv* et de *Data.txt*). Quand on appuie sur *Entrée*, on réalise une action qui va dépendre du menu dans lequel on est. Si on est dans *Continue*, on désactive le mode Pause. Si on est dans le menu *Ships* ou *Tracks*, on met à jour respectivement `self.ship` et `self.track`. Si on est dans *Restart*, si on a choisi *No*, alors on passe le menu de sélection à 0 (là où on continue). Si on a choisi *Yes*, on réinitialise la carte en réinitialisant la liste contenant nos objets et en régénérant la carte en fonction de `self.ship` et `self.track`.

Par défaut quand on lance le menu pause, `self.pause_menu_select` est par défaut à 0, cependant **le sous-menu de sélection par menu peut être personnalisé grâce au dictionnaire `self.sub_menu_by_default`**. Pour le menu *Restart* elle est à 1 donc sur *No*. Pour *Ships* et *Tracks*, elle est sur le numéro associée au vaisseau de `self.ship` et le circuit de `self.track` (cela permet de voir si la sélection a bien été prise en compte après avoir appuyer sur *Entrée*).



Pour l'affichage du texte, on fait appel à la méthode `menu_display` qui met à jour 2 objets textes dans le menu: **la flèche de sélection du menu** et **le texte “Select” pour la sélection du sous-menu**. Nous avons du texte pour chaque menu avec des coordonnées (x_{1_a}, y_{1_a}) pour le coin bas-gauche et (x_{2_a}, y_{2_a}) pour le coin haut-droit (avec a le numéro du menu). La flèche de menu à des coordonnées (x, y) pour le coin bas-gauche et (x', y') pour le coin haut-droit où $(y, y') = (y_{1_a}, y_{2_a})$. Ces coordonnées qui dépendent du numéro du menu sont stockées dans une liste `self.pause_menu_arrow_y`.



Pour l'affichage de l'objet texte `Select` pour la sélection du sous-menu, pour le menu `Restart`, le fonctionnement est le même mais on copie les coordonnées x et non y cette fois-ci.

Pour le menu `Continue`, on fait disparaître l'objet.



Pour les menus `Ships` et `Tracks` on décide d'avoir 4 sélections possibles visibles sur l'écran (donc 4 objets textes). Si le nombre de sous-menu est égal à 4, alors tout se passe comme pour le menu `Restart` mais avec 4 sous-menus. Si le nombre de sous-menu est inférieur à 4, alors on cache ceux qui sont inutiles et tout se passe comme si on avait 4 sous-menu mais avec des emplacements vides. C'est le cas du menu `Ships`.

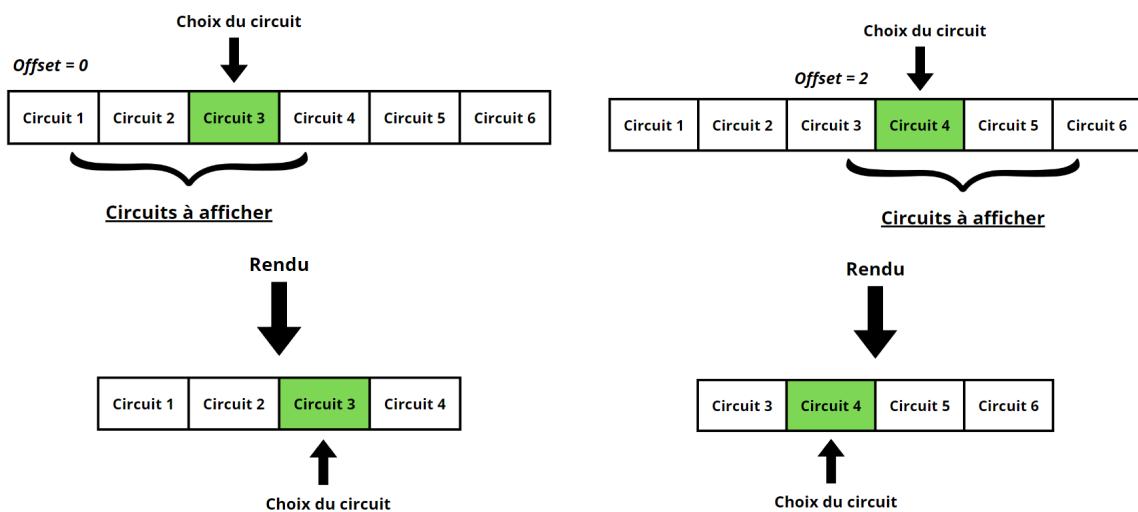


Si le nombre de sous-menu est supérieur à 4 alors on définit un offset dans chaque sous-menu. On n'affichera que 4 éléments issue d'une liste `self.ship_menu_list` ou `self.track_menu_list`. Ces listes correspondent au 4 premiers éléments de l'ensemble des véhicules et circuits à partir de l'indice d'offset.

L'offset est par défaut initialisé à 0 au lancement du jeu et est mis à jour quand le sous-menu choisi n'est plus dans la liste à afficher. En notant o pour l'offset, s le numéro du sous-menu on obtient que:

Si $s - o > 3$, alors tant que $s - o > 3$, on incrémente o de 1.

Si $s - o < 0$, alors tant que $s - o < 0$, on diminue o de 1.



Exemple où on lance le jeu sur le circuit 1 puis on appuie 2 fois vers la droite

Exemple où on lance le jeu sur le circuit 1 puis on appuie 3 fois vers la gauche

Ensuite chaque objet texte i prend le i ème élément de la liste à afficher et met à jour sa caractéristique `value`.

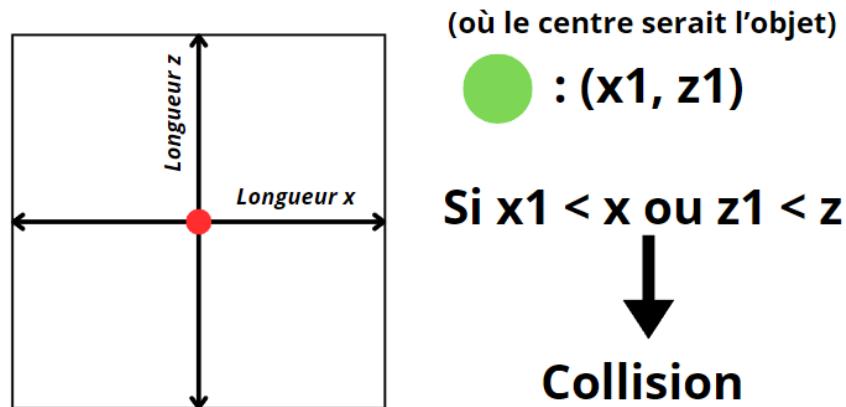


V) Physique

Système de collision

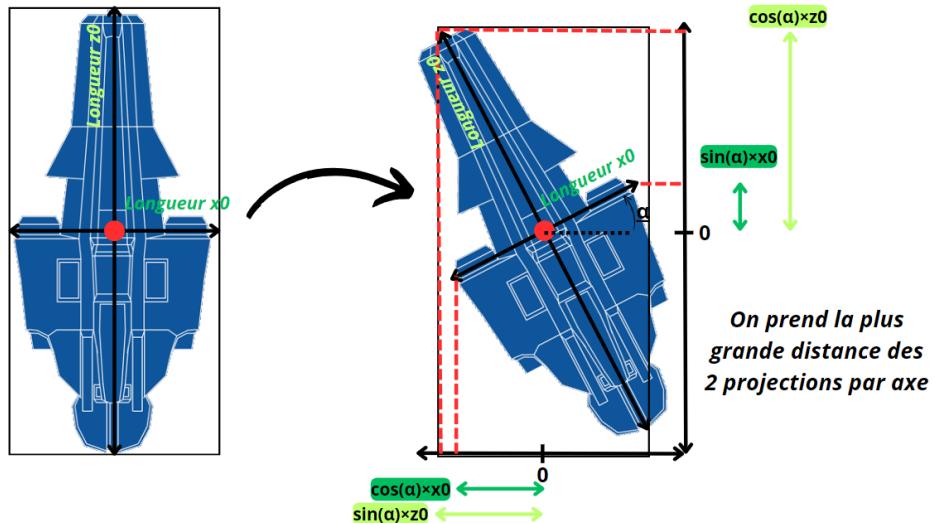
Le système de collision consiste à regarder les coordonnées du centre de notre vaisseau et de l'objet qu'on souhaite regarder et le comparer à des distances fixes ou variables selon la méthode de définition de notre hitbox de notre objet qu'on souhaite appliquer.

La méthode de **hitbox rectangulaire fixe** est très simpliste: on prend les coordonnées du centre de notre objet et on définit une **longueur x** et une **longueur z** de la taille de notre hitbox selon les vecteurs de la scène x et z (il n'y a pas besoin de le faire en fonction de y ici vu que notre vaisseau reste au sol). On a ainsi un **squelette en croix** et on obtient donc la hitbox ci dessous:



La méthode de **hitbox rectangulaire variable** est plus complexe et fait importer une fonction dans le fichier *collision_angle.py*, mais permet de prendre en compte comment est orienté l'objet si ses angles d'euler ont changé depuis le chargement de celui-ci au début de la scène. Les paramètres à prendre en compte sont la **longueur x** et la **longueur z** à **l'angle 0°** ainsi que **l'angle d'euler yaw** qui ici est le seul à prendre en compte vu que le vaisseau reste toujours parallèle au sol. Cette angle yaw va ainsi faire changer la longueur x et la longueur z en la diminuant tout d'abord. La longueur séparant le centre et l'un des extrémités de notre squelette en croix va dépendre de l'angle. On peut **projeter cette longueur** pour revenir à la méthode hitbox rectangulaire fixe en multipliant la longueur x (resp. z) à l'angle 0° par $\cos(\text{yaw})$ pour obtenir la nouvelle longueur. Cependant il y a un **début d'ambiguïté** car à l'angle 0° il n'y a qu'une valeur possible pour chaque longueur mais avec un angle non nul, il y en a 2. Par exemple pour la nouvelle longueur x, viendra un moment donné où quand l'angle se rapprochera de 90°, $\cos(\text{yaw}) \times \text{longueur } x_0$ sera plus petit que $\sin(\text{yaw}) \times \text{longueur } z_0$. Ainsi il faut prendre **la plus grande valeur entre les 2 pour chaque angle** d'où:

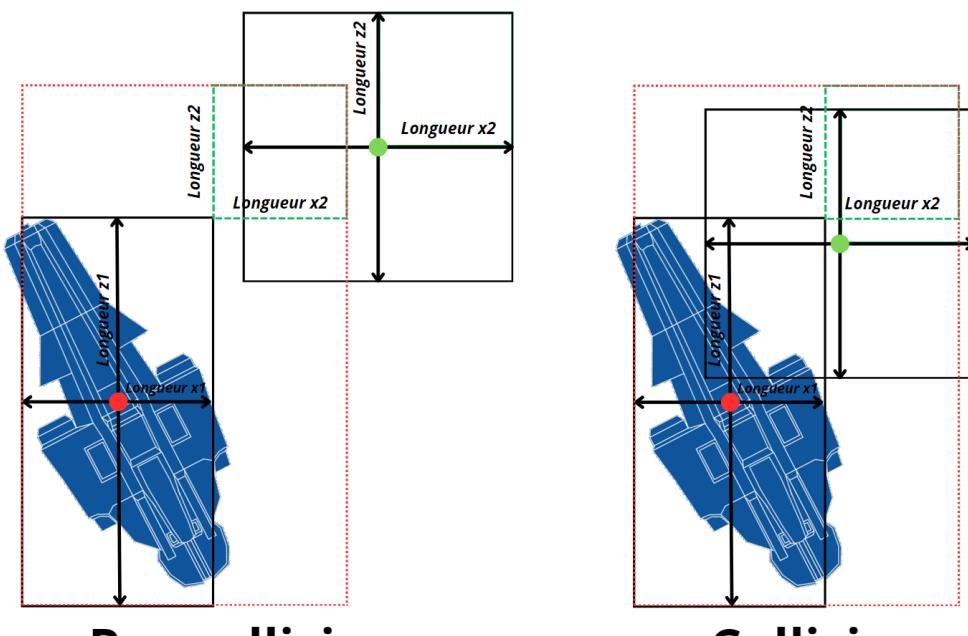
- longueur x = $\max (\cos(\text{yaw}) \times \text{longueur } x_0, \sin(\text{yaw}) \times \text{longueur } z_0)$
- longueur z = $\max (\cos(\text{yaw}) \times \text{longueur } z_0, \sin(\text{yaw}) \times \text{longueur } x_0)$



La méthode hitbox rectangulaire fixe est idéale pour tous les objets de la scène qui restent immobiles (cubes et checkpoints) car **elle est simple et elle ne changera pas**.

La méthode hitbox rectangulaire variable est idéale pour notre vaisseau car celui-ci va **changer d'angle** pendant la course. Cette méthode a tout de même ses limites. En effet la longueur qu'on projette est la distance est celle d'un des points composant notre croix (et non l'un des coins de notre rectangle). Ainsi **une partie de notre rectangle de départ** ne sera **pas comprise dans notre nouveau rectangle** trouvé après les projections.

Enfin une fois que les hitbox sont définies (ou plutôt les longueurs de notre squelette en croix), il suffit de regarder si la **différence absolue des positions** en x (resp. en z) entre le vaisseau et l'objet est plus **petite que la somme des longueurs des hitboxs** en x (resp. en z) du vaisseau et de l'objet.

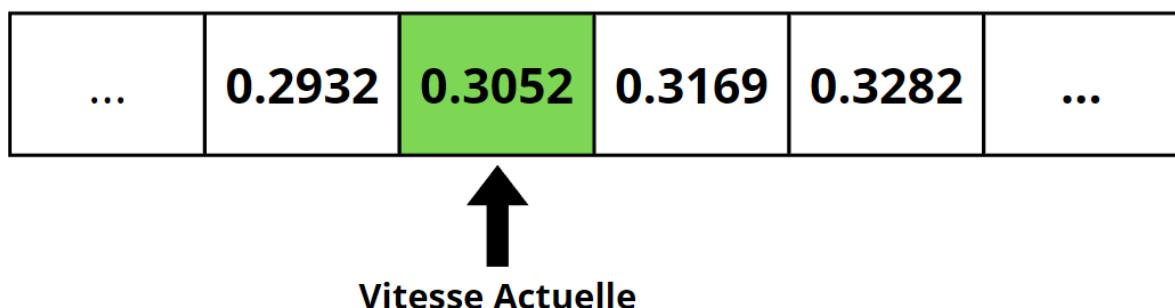


Pour voir s'il y a collision, on regarde si le point vers est dans le cadre rouge en pointillé

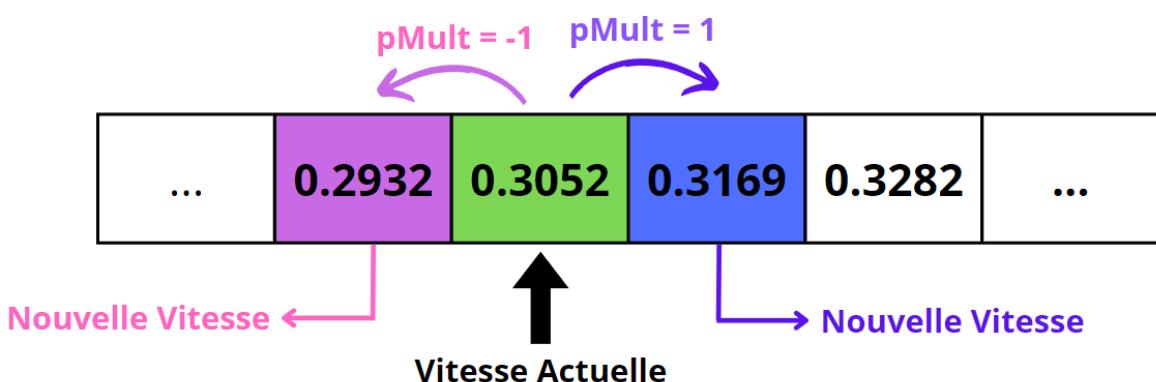
Si les deux conditions sont remplies, alors il y a **collision**. À partir de là, il est possible de faire des événements comme actualiser les checkpoints et le nombre de tours comme vu précédemment ou alors d'empêcher le déplacement de notre vaisseau. Pour cela il faut regarder, avant que le vaisseau se déplace, **simuler le déplacement** et regarder s'il y a collision (ainsi les coordonnées qu'on envoie dans la fonction sont celles qui sont simulées), s'il y a collision avec un seul objet de type *Cube*, on empêche le déplacement (et on actualise la vitesse à 0 par simplicité), sinon on l'autorise.

Système de vitesse, principes d'inertie et d'accélération

Nous configurons 2 vitesses avec inertie dans ce jeu, la vitesse de déplacement positive et la vitesse de déplacement négative. Nous séparons les deux pour bien distinguer des cas comme nous l'expliquerons ensuite. Chaque vitesse sera en réalité une valeur parmi une **liste de valeur** de vitesse possible (stockée dans une liste). Le but est de pouvoir rendre la nouvelle vitesse à la frame suivante dépendante de l'emplacement de la vitesse actuelle dans cette liste.

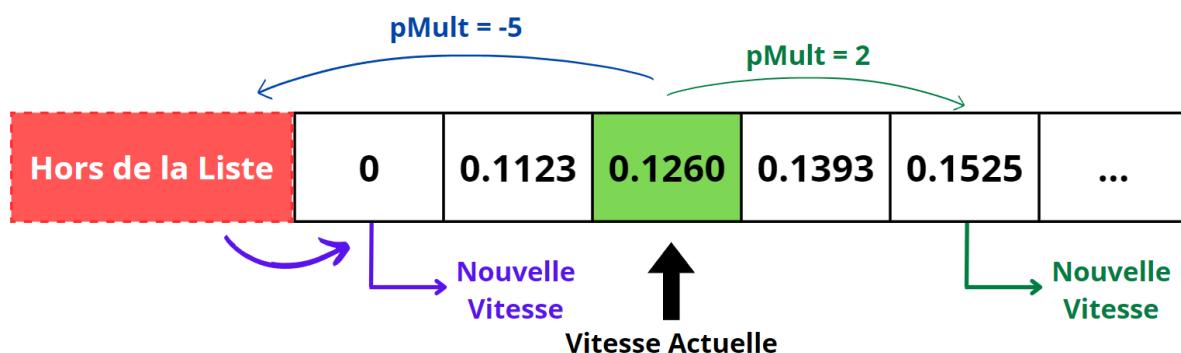


Par exemple, dans un cas simple, en maintenant la flèche avant, la valeur de la nouvelle vitesse sera déterminée par l'emplacement de la vitesse actuelle dans la liste puis on prend la valeur de l'index suivant. En définissant **pMult** le nombre de déplacements algébrique dans la liste, pMult vaudra 1 ici. Si ensuite on lâche la flèche du haut, alors la vitesse la frame suivante sera à l'emplacement de la vitesse actuelle moins 1 donc pMult vaudra -1. **pMult** ici permet de simuler l'accélération (positive et négative) et donc par la même occasion un effet d'inertie.



De même si on constate que notre vitesse de déplacement positive est non nulle mais qu'on appuie sur la flèche arrière, on peut modifier la valeur de pMult pour la passer à -5 pour décélérer plus vite. Cependant si on est en marche arrière, appuyer sur la flèche arrière ne devrait pas décélérer notre vitesse (réduire notre vitesse en valeur absolue). Cette accélération pMult pourrait être utilisée aussi pour créer potentiellement des événements qui viendrait augmenter notre accélération.

Si la valeur de notre indice dépasse 0 en négatif ou la longueur de notre liste en positif, la valeur sera automatiquement celle de l'extrême dépassée (vitesse maximale ou minimale). On réalise exactement la même chose pour la vitesse négative, mais ayant la possibilité de configurer cette vitesse à partir d'une toute autre liste, cela permet d'avoir plus de flexibilité sur la configuration de notre véhicule. On fait cela en important le fichier `speed_calculator.py`.



Système d'arrêt du véhicule

Pour ajouter du réalisme dans la vitesse de déplacement, nous avons ajouté un **temps d'arrêt obligatoire** avant de pouvoir redémarrer. Ainsi il n'est pas possible de passer d'une marche arrière à une marche avant instantanément. Pour cela on doit repérer la dernière fois que le véhicule s'arrête, c'est-à-dire que la vitesse du véhicule est égale à 0 et la vitesse précédente est non nulle. Ensuite, on n'autorise le déplacement, à travers une variable, seulement quand le temps actuel moins le temps du dernier arrêt est supérieur à une différence de temps qu'on définit **temps lag à 0** que nous avons défini à 0,2s. Ainsi le véhicule ne peut se déplacer qu'à partir de 0,2s.

```
self.move_perm = (glfw.get_time() - self.last_0 - self.timer_dict["last0"]) > self.move_backward_lag
```

```
if previous_speed - self.translation_speed_dec == previous_speed and previous_speed != 0:
    self.last_0 = glfw.get_time()
    self.timer_dict["last0"] = 0
```

Synchronisation en temps réel

Il est nécessaire pour notre jeu de faire une synchronisation en temps réel car chaque action est réalisée à chaque rafraîchissement de la fenêtre ce qui peut poser du in-game lag. On préférera du real time lag où on aura l'impression lors d'un déplacement de ne pas voir l'intégralité du déplacement mais celui-ci semblera la même que sans le lag (on n'aura pas l'impression d'aller plus lentement). Pour cela juste avant l'affichage on récupère le temps avec la fonction `glfw.get_time` et on la stock dans `self.last_frame`.

```
self.updating_text()          TristanP, 2 weeks
self.update_key()
#print(1/(glfw.get_time()-self.last_frame))
self.last_frame = glfw.get_time()
self.timer_dict["last_frame"] = 0
```

On fait l'affichage et au moment où on souhaite faire un déplacement ou appliquer `pMult` (notre accélération), on applique la méthode `self.speed_real_time`. Le déplacement ou l'accélération qu'on note p qu'on souhaite appliquer est celle qu'on veut appliquer sur un intervalle de temps arbitraire `self.one_frame_rule` qu'on note f paramétrable dans `Data.txt` qui vaut par défaut 0,016s. On calcule la différence de temps Δt entre `glfw.get_time` et `self.last_frame` et on réalise un produit en croix pour déterminer $p_{réel}$, ainsi $p_{réel} = p \times \Delta t / f$. Vu que `pMult` est supposé être un indice, on prendra la partie entière du résultat pour ce cas particulier. D'ailleurs bien que la fonction ne s'en serve pas, il est possible de multiplier $p_{réel}$ par un coefficient `pCoeff`. Cela peut s'avérer utile si plus tard dans le développement du jeu, nous mettons des boosters (on augmentera la vitesse de déplacement en le multipliant par un `pCoeff` qui partira de 1 à un instant t et augmentera jusqu'à un `pCoeff_max` avant de redescendre à 1), ou des options de freinage plus puissante (on augmentera `pMult` en multipliant par un `pCoeff` constant). On peut à tout moment se passer de cette synchronisation en temps réel si le paramètre `pBool` est à `False`.

```
def speed_real_time(self,speed,pBool,pCoeff=1):
    if pBool:
        time_diff = glfw.get_time()-(self.last_frame+self.timer_dict["last_frame"])
        coeff = time_diff/self.one_frame_rule
        self.coeff_compensate_time = coeff
        return (speed *coeff*pCoeff)
    else:
        return speed*self.not_sync_mult*pCoeff
```

Synchronisation prenant en compte le menu pause

Le menu pause apporte un nouveau problème à la physique du jeu. En effet, son système repose sur le fait qu'on n'exécute pas les fonctions reliées au reste de la physique du jeu, mais un paramètre changera tout de même, **le temps qui se défile**. Dans notre jeu nous avons plusieurs repères temporelles (la dernière frame de rafraîchissement in-game, la dernière frame de temps lag à 0, le temps du début de la course et du tour courant, le temps de la fin de la course et les dernières frames d'interactions dans le menu *View Mode*, que ce soit accéder ou quitter le menu ou passer en mode cinématique) qui seront décalés à cause du mode Pause.

Pour compenser cela on introduit un **dictionnaire *timer_dict*** qui a pour clé la valeur temporelle à compenser et pour valeur la valeur qu'elle doit compenser. Cette valeur est initiallement à 0 et est augmentée, quand on quitte le menu Pause, par le temps passé dans le menu pause (il faut donc des repères temporels pour savoir quand on accède au menu pause et quand on le quitte). Cette valeur est remise à 0 quand la valeur qu'elle doit compenser est actualisée. La valeur du dictionnaire doit toujours correspondre au **temps total de pause** depuis la dernière actualisation de la valeur temporelle.

```
self.timer_dict = {}          TristanP, 2 weeks ago • Fina

self.timer_dict["last0"] = 0
self.timer_dict["free_mode_input"] = 0
self.timer_dict["text_display_input"] = 0
self.timer_dict["time_finished"] = 0
self.timer_dict["last_frame"] = 0
self.timer_dict["time_delay_track"] = 0
for i in range(self.last_lap):
    self.timer_dict["time_delay_lap"+str(i+1)] = 0

if self.touch[glfw.KEY_V] == glfw.PRESS \
    and(glfw.get_time()-self.free_mode_input - self.timer_dict["free_mode_input"])>0.2:      TristanP, yesterday • Quelques
    self.free_mode = not(self.free_mode)
    if not self.free_mode:
        self.cam.transformation.rotation_center = self.objs[0].transformation.translation + self.objs[0].transformation.rotat
        self.cam.transformation.translation = self.objs[0].transformation.translation + pyrr.Vector3(self.camera_offset_trans
    self.free_mode_input = glfw.get_time()
    self.timer_dict["free_mode_input"] = 0
```

Système de vitesse pour les véhicules

Pour modéliser l'évolution de la vitesse de déplacement du véhicule, nous nous sommes appuyés sur une fonction $\text{Arctan} \circ \ln$ pour avoir une fonction avec une faible valeur dérivée vers 0, puis une valeur dérivée plutôt importante avant de stagner et de tendre vers une constante.

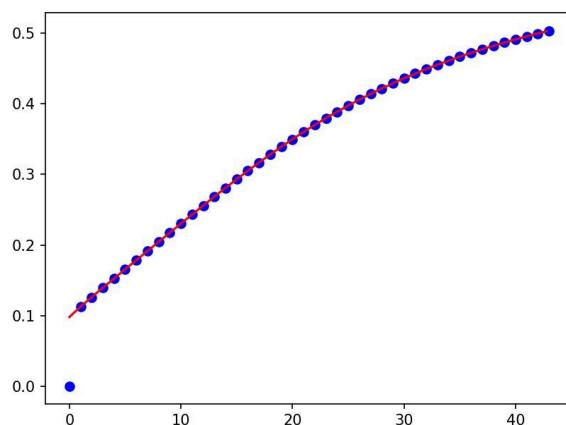
Ainsi la fonction que nous avons utilisé est la suivante:

$$f(x) = \frac{(A+0,2)}{\pi} \times \left(\frac{\pi}{2} + \text{Arctan}(\ln((x \times \tau) + 0,1)) \right)$$

Avec A la vitesse maximale de notre véhicule et τ un coefficient influençant sur la vitesse de convergence vers A , ainsi on le considéra comme notre statistique d'accélération du véhicule.

Ensuite, on importe depuis le fichier *machines_stats.py* la fonction *Speed_translation_tab_maker*, qui en fonction des paramètres de vitesse maximale et d'accélération renvoie une liste qui est un échantillonnage de cette fonction. On ajoute dans la liste, toutes les valeurs de vitesse $f(x)$ avec x initialisé à 0 et qui augmente de 1 à chaque boucle, tant que $A - f(x) > \varepsilon$ avec ε un paramètre qu'on choisit arbitrairement à 0.05. La première de la valeur de la liste ne sera pas nul car il n'est pas possible de faire $\ln(0)$ donc on change manuellement à la fin la première valeur de la liste.

```
while speed_max - Speed > epsilon:
    Speed = f1(speed_max,i,accelerating)
    Speed_translation_tab.append(Speed)
    i+=1
Speed_translation_tab[0] = 0
return Speed_translation_tab
```



Courbe de la vitesse et échantillonnage (points bleus) de celle-ci

VI) Traitement des Données

Configuration des statistiques pour les véhicules

On configure **les statistiques de chaque véhicule dans un fichier *Vehicule_Sats.csv***. Le format du fichier permet de le modifier facilement sur Excel et de l'importer facilement sur Python. On importe le fichier *data_reader.py* pour importer les données dans notre jeu. Quand on lit le fichier sur Python, grâce à la fonction *all_vehicules_stat_reader*, on met dans une liste chaque ligne qui est elle-même une liste. Quand on souhaite appeler aux statistiques d'un véhicule, en appelant à la fonction *vehicule_reader* et en donnant le nom du véhicule, on reçoit un **dictionnaire contenant les caractéristiques du véhicule** avec comme clé le nom de la caractéristique et en valeur la valeur associée à la caractéristique.

```
Vehicules;Speed;Acceleration;Reverse;RAcc;Turn;Brake;RBrake;Lag0;SizeX;SizeZ
Ship0;9;9;9;9;9;9;9;0.2;1.1;2.8
AG-SYS;6;8;2.5;8;7;8;6;0.2;1.1;2.8
Ship1;6;8;2.5;8;7;8;6;0.2;1.1;2.8
Ship2;9;9;9;9;9;9;9;0.2;1.1;2.8
```

```

def vehicule_reader(vehicule_name):
    all_vehicules_stats = all_vehicules_stat_reader()
    line = 1
    while all_vehicules_stats[line][0] != vehicule_name and line < len(all_vehicules_stats)-1:
        line+=1
    if all_vehicules_stats[line][0] != vehicule_name:
        print("Error: Vehicule not found!")
        return None
    else:
        vehicule_stat={}
        vehicule_stat["Name"] = all_vehicules_stats[line][0]
        for ind in range(1,len(all_vehicules_stats[0])):
            vehicule_stat[all_vehicules_stats[0][ind]] = float(all_vehicules_stats[line][ind])
    return vehicule_stat

```

Les caractéristiques du véhicules sont: la vitesse maximale en marche avant, l'accélération du véhicule en marche avant, la vitesse maximale en marche arrière, l'accélération du véhicule en marche arrière, le coefficient pour tourner, l'accélération de freinage en marche avant, l'accélération de freinage en marche arrière, le temps nécessaire d'arrêt du véhicule à vitesse nulle, sa longueur x à 0° et sa longueur en z à 0°.

L'accélération de freinage *Brake* correspond au coefficient de *pMult* quand on veut augmenter notre décélération lors d'un déplacement, par exemple si *Brake* vaut 5, que je suis en train d'avancer tout droit et que j'appuie sur la flèche du bas, *pMult* passera à -5.

Les longueurs *x* et *z* sont utiles pour la définition de la hitbox du vaisseau et donc la détection de collision. Le coefficient pour tourner va juste plus ou moins faire tourner le vaisseau quand on appuiera sur les flèches droites et gauches. Il est d'abord multiplié par une valeur arbitraire de $\frac{0.05}{7}$. Enfin les vitesses maximales et accélérations permettent de créer les listes de vitesse positives et négatives (cf Système de statistiques pour les véhicules). On divisera la vitesse maximale par 10 et l'accélération par 200 avant de l'appliquer dans la fonction que nous avons définie plus tôt.

```

def f1(A,x,to):
    return (A+0.2)*((math.pi)/2+math.atan(math.log((x*to)+0.1)))/math.pi

def Speed_translation_tab_maker(speed_stat,acceleration_stat):
    Speed_translation_tab = []
    Speed = 0
    speed_max = speed_stat/10
    epsilon = 0.05
    i = 0
    accelerating = acceleration_stat/200

```

```

# Initialisation of the Machine Stats
self.Speed_translation_tab = Speed_translation_tab_maker(speed_forward_stat,acceleration_forward_stat) # Forward Stats; Basic S
self.Speed_translation_tab_backward = Speed_translation_tab_maker(speed_backward_stat,acceleration_backward_stat) # Backward Sta
self.TurnSpeed = 0.05 * turning_stat/7 # Initialisation of the turning speed when pressing a direction; Basic Stats 7
self.pMultDecrease_forward = (2*forward_brake)//5 # Says how much we decrease if we try to run back when moving forward; Basic
self.pMultDecrease_back = (2*backward_brake)//5 # Says how much we increase if we try to run forward when moving backward; Basi
self.move_backward_lag = vehicule_stat["Lag0"] # Says how much time we wait when we reach Speed 0 before moving again

self.ship_hitbox_x = vehicule_stat["SizeX"]
self.ship_hitbox_z = vehicule_stat["SizeZ"]

```

Configuration des paramètres du jeu

Nous souhaitons maintenant placer tous les paramètres du jeu (certaines constantes, certains Booléens...) dans un **fichier Data.txt** pour que ces variables soient facilement accessibles et changeable si on veut changer la configuration d'un objet, sans tout changer dans le code ou alors sans devoir retrouver la valeur dans le code.

Pour cela on souhaite les **trier par catégorie**, chaque catégorie commencera par une ligne dont le premier caractère est #. Pour chaque catégorie on crée un dictionnaire et on ajoute dans le dictionnaire chaque ligne non vide avec comme clé une chaîne de caractère qui décrit la valeur de la ligne et la valeur est la valeur écrite en chaîne de caractère de la ligne. Par exemple, *Track* dans la catégorie *Customisation Settings* signifie que le dictionnaire *Customisation Settings* contient à la clé *Track* le nom du circuit au lancement du jeu. La clé et la valeur sont pour le moment forcément des str. On les extrait en lisant la ligne de la manière suivante: Tant qu'on ne rencontre pas le caractère “：“, le caractère appartient à la clé. À partir de “：“, elle appartient à la valeur. Dès qu'on rencontre un #, on suppose que la ligne est finie (cela permet de mettre des commentaires dans notre fichier texte à la fin de la ligne, comme en Python). Attention il est important de mettre un espace après “：“ pour que cela fonctionne correctement !

```
# Customisation Settings
Track: Metropolis
Ship: AG-SYS      TristanP, 2 weeks ago • Data+Machine file texts + Free View Mode

# Camera Settings
Camera Quick Follow Translation: True # Says if the camera will try to follow the ship when moving as soon as possible (not waiting the n
Camera Quick Follow Rotation: True # Says if the camera will try to follow the ship when turning as soon as possible (not waiting the nex
Camera Offset Translation: [0, 1.25, 7.5] # Offset of the camera compared to the ship when quick follow
Camera Turning Offset Quick Follow: 0.1 # Offset of the camera when the ship is turning
Camera Moving Offset Speed Forward: 0.2
Camera Moving Offset Speed Backward: 0.1
```

On importe le fichier *data_reader.py* pour pouvoir extraire tous les dictionnaires de catégories et associer chaque dictionnaire à une valeur dans notre *ViewerGL*. Ensuite on demande d'extraire la valeur qui nous intéresse en faisant par exemple le code ci-dessous:

```
All_data = data_reader.data_dump_by_category()
customisation_settings = data_reader.data_dump_a_category(All_data,0)
camera_settings = data_reader.data_dump_a_category(All_data,1)
synchronisation_settings = data_reader.data_dump_a_category(All_data,2)
gameplay_settings = data_reader.data_dump_a_category(All_data,3)
tracks_settings = data_reader.data_dump_a_category(All_data,5)

self.track = data_reader.data_dump(customisation_settings,"Track","s")
```

On doit décrire dans la fonction *data_dump* **le type de valeur qu'on souhaite avoir** (une chaîne de caractère est “s”, un nombre sera “i”, un Booléen sera “d”et une liste “l”).

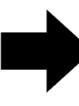
Avoir une valeur de type “s” revient juste à renvoyer la valeur du dictionnaire. Avoir une valeur de type “i” ou “b” revient à renvoyer le retour de la fonction eval de Python avec comme argument la valeur du dictionnaire. Pour les valeurs de type “l”, suite à des mauvaises expériences personnelles du passé, nous nous passerons de la fonction eval de

On parcourt cette matrice et on regarde la valeur dans chaque case de cette matrice. On distingue chaque type d'objet et on ajoute ses coordonnées dans une liste (on fera bien attention, l'emplacement sur la carte devra être multiplié par la taille d'un cube, on suppose ici qu'un cube prend l'emplacement total d'une case dans notre matrice). On prendra bien en compte l'activation ou non du **mode Miroir** (droite et gauche inversée) et du **mode Reverse** (on peut faire le circuit à l'envers). Ces modes peuvent être activés à tout moment via le fichier *Data.txt*. Il y a donc une liste d'objets pour:

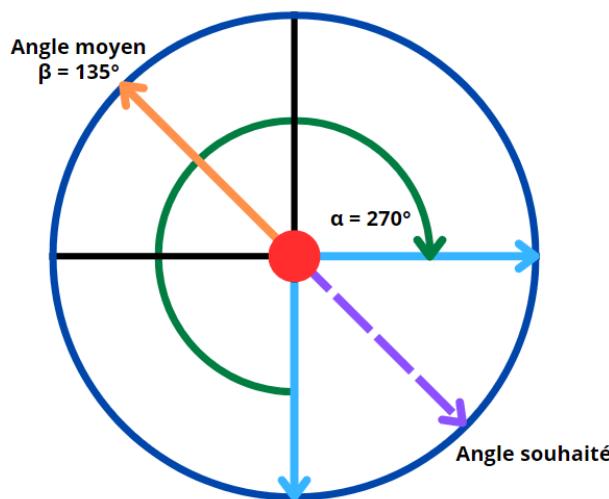
- Les paramètres du spawn du vaisseau
- Les Blocs “B”
- Les Cactus “C”
- Les Palmiers “P”
- Les Flèches “F”
- Les Checkpoints avec un numéro

Pour le vaisseau, on considère un **lieu de spawn** une case qui commence par *N* ou *R*, qui contient l'un des caractères de la liste suivante $L = ["v", "<", "^", ">"]$ (et $L = ["v", ">", "^", "<"]$ en mode *Miroir*), et qui contient un numéro (le numéro du vaisseau qui doit spawn, ici comme on est sur un contre-la-montre, on considère que l'emplacement de notre vaisseau est particulière donc on le place sur l'emplacement 0). Ainsi “**Nv0**” est un milieu de spawn, tout comme “**R^0**”. Chaque carte contient au moins 2 milieux de spawn, un contenant *N* pour le spawn en mode Normal et *R* pour le spawn en mode Reverse.

Les caractères sont des flèches qui indiquent le sens dans lequel le véhicule doit être orienté. L'interprétation dans notre code est que par défaut le vaisseau sera dirigé vers le bas avec un **angle yaw de 0 rad**. Ensuite, l'angle augmentera de $\frac{\pi}{2}$ rad dans le sens d'une aiguille d'une montre. Ainsi, en connaissant le caractère qui symbolise notre flèche, on peut déterminer **son indice d'emplacement** dans la liste *L*, ce qui fait que “R^0” se traduit par un angle de $2 \times \frac{\pi}{2} = \pi$ rad. Lors du spawn du vaisseau, on fera attention de bien lui appliquer cet angle-ci.

v	<	^	>
			
0°	90°	180°	270°

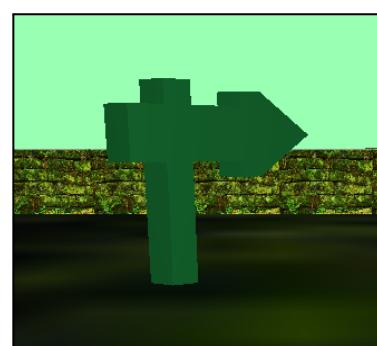
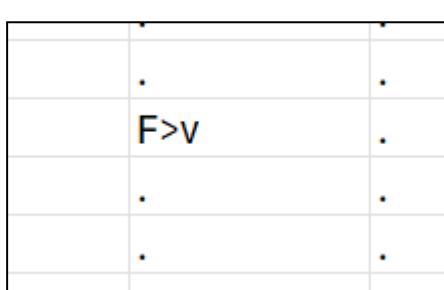
Nous sommes allés encore plus loin pour l'orientation de certains objets. Les **objets 3D Flèches**, **Cactus** et **Palmiers** peuvent eux aussi être orientés avec plusieurs flèches (d'ailleurs les objets Cube ne peuvent pas tourner dans la scène). Par exemple "C>v" est un cactus avec un angle *yaw* de $\frac{\pi}{4}$ rad, tandis que "F^^>" est un objet Flèche avec un angle *yaw* de $\frac{15\pi}{8}$ rad. Ici la méthode débute comme celle du vaisseau on relève chaque angle grâce à l'indice du caractère dans la liste. Cependant il ne suffit pas de faire la moyenne des angles pour obtenir le résultat, en effet cela ne marche que si **la différence des angles est inférieur π** (sachant qu'on ne parcourt que les angles positifs). Ainsi "Fv>" donnerait une direction vers en haut à gauche avec cette méthode car "v" correspond à un angle de 0° et ">" de 135° comme le montre l'illustration ci-dessous. Si on considérait ">" avec un angle de -90° on aurait eu un problème avec "F^>" qui donnerait une direction bas gauche.



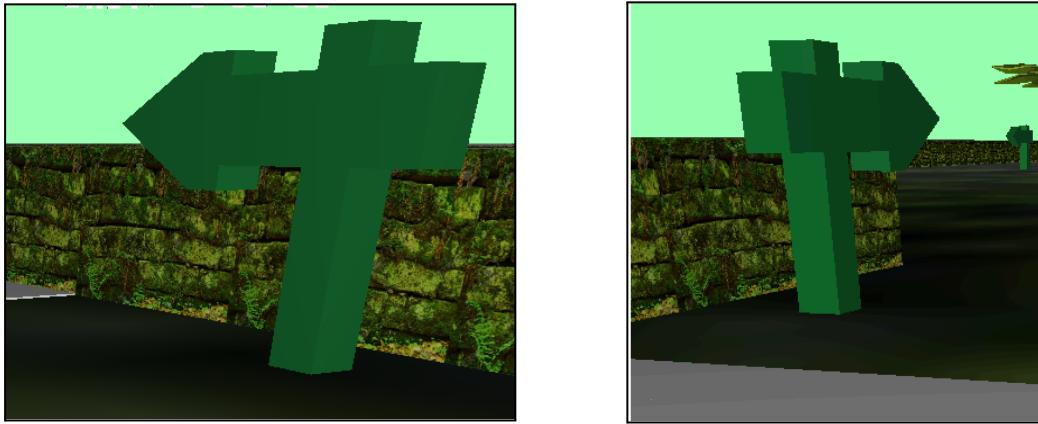
On souhaite l'angle moyen des 2 flèches bleus (l'angle 0° est vers le bas)

Nous devons donc trouver une méthode qui marcherait tout le temps. On importe le fichier `adaptative_average_angle.py` et la fonction `adapt_angle_average`. On suppose ici que l'ensemble des angles sera compris dans un **intervalle de $\frac{\pi}{2}$ rad d'écart modulo 2π rad**.

On prend l'angle le plus petit, si un angle n'a pas un écart de moins de $\frac{\pi}{2}$ rad, alors on lui retire 2π rad jusqu'à que ça soit vrai (car on suppose que l'écart est vrai modulo 2π rad). Ensuite nous pouvons calculer la moyenne et l'angle est cette fois-ci correct (on prendra la valeur modulo 2π rad pour éviter les angles négatifs).



Il existe cependant une légère subtilité pour ces angles selon le mode de jeu. Tout d'abord pour le mode *Reverse*, il faut **rajouter un angle de π rad** pour les objets Flèches qui sont censées indiquer la direction du tracé. Ensuite pour le mode *Miroir*, la liste L devient `L = ["^", "<", "v", ">"]` et la coordonnée x devient $-x$ (donc elle redevient la “vraie” composante de la coordonnée x de la matrice de la carte).



Flèche retournée en Mode Reverse

Si c'est un objet de type *Cube* ou un **objet 3D custom avec collision (3DCC)** (un Bloc “B”, un Cactus “C”, un Palmier “P”), on ajoute **le type de l'objet dans une liste triée par catégorie** dans une liste `self.cube_list` de longueur n (par exemple `self.cube_list = ["B", "B", "B", "C", "C", "P"]` est de longueur $n = 6$).

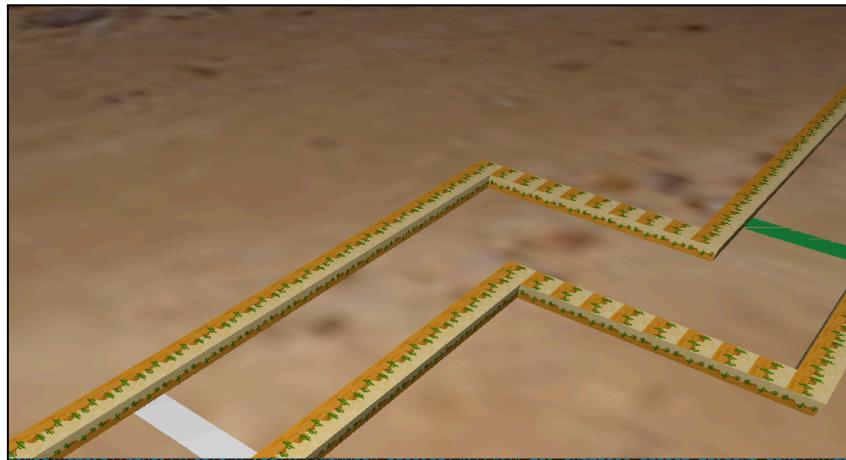
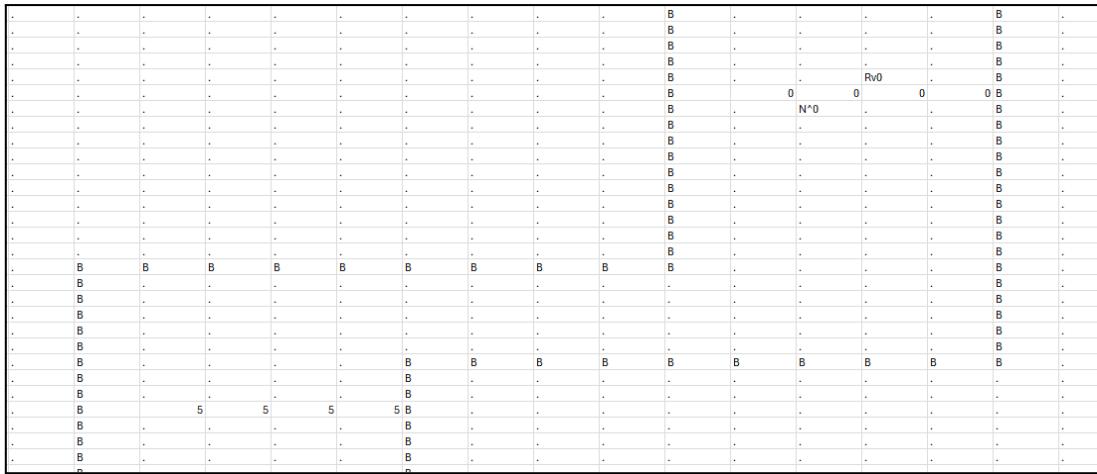
Cette liste est importante car on va la parcourir lors de tests de collision avec les objets, vu que chaque type d'objet *Cube* ou *3DCC* a **des dimensions de hitbox différentes**, il faudra s'adapter en fonction de l'objet qu'on est en train de traiter. On le fait en cherchant les données de hitbox dans un dictionnaire `self.cube_hitbox` en rentrant comme clé *Cube*, *Cactus* ou encore *Palmier*. On traite les collisions entre le vaisseau et les objets à partir de l'indice 2 jusqu'à l'indice $n+2$ de la liste `self objs` (les 2 premiers étant pour le vaisseau et le sol, cf fin de la partie).

```
def collision_translation(self, p):    TristanP, 2 weeks ago • Speed_Real_Time+Checkpoint_Working
    valren = True
    offset = 1
    vship_hitbox_sizex_0 = (self.ship_hitbox_x * self.ship_size_x / 3)
    vship_hitbox_sizez_0 = (self.ship_hitbox_z * self.ship_size_z / 3)
    angle = self.objs[0].transformation.rotation_euler[prr.euler.index()].yaw

    vship_hitbox_sizex, vship_hitbox_sizez = size_x_and_zDepending_angle(vship_hitbox_sizex_0, vship_hitbox_sizez_0, angle)

    for nb_obj in range(len(self.cube_list) + offset):
        if nb_obj >= offset:
            cube_hitbox_x = self.cube_hitbox[self.cube_list[nb_obj - offset]][0]
            cube_hitbox_z = self.cube_hitbox[self.cube_list[nb_obj - offset]][2]
        else:
            cube_hitbox_x = self.cube_size_x
            cube_hitbox_z = self.cube_size_z
        vsizemax_x = vship_hitbox_sizex + cube_hitbox_x
        vsizemax_z = vship_hitbox_sizez + cube_hitbox_z
        if abs(self.objs[nb_obj + offset].transformation.translation.x - p.x) <= vsizemax_x and abs(self.objs[nb_obj + offset].transfo
            valren = False
    return valren
```

Pour les **checkpoints** on distingue les repères des checkpoints dans la liste `self.checkpoint_list` des objets checkpoints représentés par des cubes dans `self.checkpointobj_list`. La première liste est composée d'un triplet par checkpoint avec ses coordonnées x et z ainsi que le numéro du checkpoint c (on pensera à mettre $-c\%(c_{\text{dernier}}+1)$ dans le cas où on joue en mode *Reverse*). Cette liste sera utilisée pour **la collision des checkpoints**. La deuxième liste ne sert qu'à stocker **les objets 3D qui sont des cubes** suffisamment encrés dans le sol pour croire qu'il s'agisse de ligne épaisse de couleur verte (sauf pour la ligne de départ de checkpoint 0, elle est blanche).



Enfin on créera **une géométrie** qui sera un objet 3D mais qui correspond en réalité au sol. On applique dessus une texture qui dépend du circuit qu'on a choisi, en faisant dépendre l'argument du chemin du fichier dans la fonction `load_texture` de `glutils.py`. Ce sol sera proportionnel à la taille des cubes de notre circuit (quand on augmente la taille des cubes, le sol prendra lui aussi plus de place).

On ajoutera tous les objets dans une **liste `self.objs`** dans cet ordre: Vaisseau, Sol, Objets Cube, les Flèches et les Checkpoints (attention les objets 3D ici!). Cela permettra de faciliter la gestion des collisions comme vu précédemment.

VII) Interfaces

Système d'affichage de données à l'écran

Le système d'affichage de données se fait avec des **objets Textes** stockés dans le dictionnaire `self.text_dict` avec la signification du texte en clé et l'objet en valeur. Ce qu'affiche l'objet texte est stocké dans la **caractéristique .value** tandis que les emplacements sont gérés par l'emplacement du coin inférieur gauche `.bottomLeft`, et le coin supérieur droit `.topRight`. À part les cas cités dans le menu Pause, leurs emplacements se font à la main.

À chaque boucle d'affichage, on appelle la **méthode `updating_text`** qui vient mettre à jour la caractéristique `value` de chaque texte qui doit se mettre à jour (les timers, le compteur de vitesse, le nombre de tour et de section et les composantes de *transformation* visibles dans le *View Mode*). Dans la plupart des cas, il suffit de reprendre directement la valeur qui nous intéresse (comme le nombre de tours et le numéro du dernier tour), mais pour **la vitesse**, on prend la vitesse d'origine du vaisseau (et non celle compensée par la synchronisation en temps réel) et on la multiplie par une valeur `self.speed_text_convert` pour la “convertir” en km/h (Cette valeur est paramétrable dans *Data.txt* et vaut 205 km/h par unité normale dans le jeu).



Pour les timers, on calcule la plupart d'entre eux directement dans la fonction d'affichage. On stock le temps du départ de la course dans la valeur `self.time_start` et celles des différents tours dans la liste `self.time_lap`. On initialise les valeurs à -1 lors de l'exécution du fichier et lors du premier appel de la fonction, on les ré-initialise à nouveau (cela permet d'initialiser quand la course a commencé plus tard dans le développement si on avait plus de temps). Les valeurs de réinitialisation sont 0 pour les temps de tours non courants (typiquement en condition normal, tous les tours 2,3,4...) et de `start = glfw.get_time()` pour le début de la course et le tour courant. Quand on finit un tour, on met le temps du tour qu'on vient de finir à `glfw.get_time() - start` et celui du tour qu'on commence à `glfw.get_time()` (si la course n'est pas finie). Si la course est finie, on fait aussi `glfw.get_time() - start` (son start à lui a été fait au début de la course).

Une fois qu'on obtient le temps qu'on souhaite afficher, on fait la division euclidienne par 60 pour obtenir **les minutes**, puis on soustrait le temps par le nombre de minutes fois 60 et on fait la division euclidienne par 1 pour obtenir **le nombre de secondes** dans la minute et enfin on enlève le nombre de secondes et on fait la division euclidienne par 0.01 pour avoir **le nombre de centième de seconde**.

TRACK: 1' 36" 63 LAP: 0' 06" 66 LAST: 1' 29" 95

Système de cinématique de fin de course

Lorsque la course prend fin, on passe le paramètre `self.race_finished` à `True`, ce qui va bloquer le fait de pouvoir accélérer (le vaisseau se déplacera toujours jusqu'à atteindre une vitesse nulle) et la caméra ne va plus suivre les instructions qui force à suivre le vaisseau lors du déplacement (translation et angles d'euler).

Dans ce cas, on augmente à chaque rafraîchissement l'angle `yaw` de $\frac{\pi}{100}$ rad, pour que la caméra tourne autour du vaisseau (quand nous sommes pas dans le menu pause). De plus, on joue sur l'angle de rotation `roll` pour avoir un angle de vue aplati. Plutôt de prendre l'angle `roll` présente dans l'angle offset de la caméra (qui est de $\frac{\pi}{10}$ rad) on va le translater vers un angle presque plat. Pour cela on mémorise le temps à laquelle la course se finit avec `self.time_finished`. On peut alors avoir $\Delta t = \text{glfw.get_time}() - \text{self.time_finished}$. On souhaite au bout de $T = 3\text{s}$ arriver à un angle de $\frac{\pi}{10} - \frac{\pi}{12}$ rad en partant d'un angle de $\frac{\pi}{10}$ rad (par rapport au vaisseau). Alors on déduit que l'angle `roll_rotation` est

$$r = \frac{\pi}{10} - \min\left(\frac{\pi}{12}, \frac{\pi}{12} \times \frac{\Delta t}{T}\right).$$

Cette méthode permettrait également de faire des cinématiques au début du circuit, ce que nous aurions fait si nous avions eu plus de temps.

On copie la position du vaisseau pour que le centre de rotation de la caméra prenne ces coordonnées et que la position de la caméra soit celle du vaisseau ajouté de l'offset habituel.

On décide aussi de faire apparaître les temps de la course et des tours et de stocker les textes dans une liste `self.race_finished_race_text` dont chaque texte de la liste deviendra invisible quand on mettra en pause le jeu.



VIII) Conclusion:

Ce projet nous a permis de prendre en main l'interface graphique OpenGL et de le manipuler avec Python. Il nous a permis de voir les différentes contraintes et difficultés de la programmation d'un moteur de jeu comme la création de la scène, le chargement des objets, la gestion des interactions et des collisions. Le projet nous a aussi permis de développer une physique pour le jeu et d'interpréter une base de données ordonnée et générique. Enfin il nous a permis de développer un jeu avec une intention de jeu et donc des règles qui permettent le bon fonctionnement de jeu avec cette intention de jeu (ne pas abuser du système de checkpoint, des menus pause, des timers) tout en implémentant des éléments artistiques tels que des objets ou du texte personnalisé.

Ce projet de jeu de course en OpenGL a permis d'explorer divers aspects du développement de jeux vidéo, y compris la gestion des graphismes et la programmation des interactions utilisateur. Les concepts abordés, tels que les transformations géométriques et la gestion des collisions, sont cruciaux pour tout développeur de jeux vidéo.

Si vous vous ennuyez, vous pouvez aussi mettre *Ship 0* comme nom de vaisseau dans le *Data.txt*, c'est un vieux tas de ferraille mais il est tellement puissant qu'il a été banni du Grand Prix du jeu.

