Rapport sur le projet "Jeu de quilles" de programmation

Edgar Bizel et Neil Dahoun

1 Description

1.1 Graphismes

Le jeu de quilles que nous avons créé se passe dans l'espace. En arrière plan, des astres défilent. Des étoiles, mais aussi des constellations, des galaxies et même un astre non identifié, arbitrairement appelé "Supernova".
L'ensemble de ces astres est généré aléatoirement. Chacun commence avec une position de départ et une taille de départ, déterminées par la taille de l'écran.
Au premier plan, des astéroïdes sont présents. Ils représentent les quilles, et leur forme est également aléatoire.

1.2 Comment jouer

Pour jouer, on utilise les flèches directionnelles et la barre d'espace. Les flèches permettent de sélectionner le(s) astéroïdes qui seront détruits. Les astéroïdes détruits sont représentés de la même manière, avec un noyau "laser" d'où émanent des rayons traversant l'astéroïde. Ceux ci sont rouges pour le joueur, noirs pour l'ordinateur, afin de pouvoir les différencier.
Le but est de détruire le dernier astéroïde.

2 Structure du projet

2.1 Modules

Pour éviter de se perdre, nous avons limité la taille des modules et nous avons préféré en créer un plus grand nombre.
Les modules sont les suivants :

- **action** : contient des fonctions relatives aux actions. Les actions sont des chaînes de caractères sous la forme `"ligne:côté"`
- **arriere_plan** : est responsable de l'arrière plan. C'est ce module qui lance la génération et maintient à jour l'arrière plan
- **asteroide, constellation, etoile, galaxie, supernova** : ces modules contiennent les informations sur les différents objets affichés. Ils exposent tous une fonction "generer" qui permet de générer un nombre fixé de ces objets, et "avancer_de" qui permet de les déplacer.
- **constantes** : contient uniquement des constantes, comme les couleurs des étoiles ou le nombre d'étoiles générées
- **couleur** : contient des fonctions concernant les couleurs, permettant par exemple de générer un dégradé entre deux couleurs
- **curseur** : contient le code permettant de contrôler le curseur
- **jeu** : contient les fonctions principales du jeu
- **main** : point d'entrée du programme
- **taille_ecran** : contient la taille de l'écran. Celle-ci est adaptée à l'écran du joueur (responsive)
- **texte** : contient les textes affichés par le jeu

- **turtle_util** : contient des fonctions utilitaires concernant turtle, comme "deplacer", une fonction permettant de déplacer la tortue et de la préparer à écrire.
- **util** : contient des fonctions utilitaires générales, comme "position_autour", une fonction utilisant les coordonnées polaires pour déterminer un point à la distance souhaité d'un autre. Ou "depasse_ecran", une fonction indiquant si l'objet a dépassé la limite de l'écran

2.2 Principe de l'animation

Pour réaliser une animation, nous avons du étudier la documentation de turtle.
Nous avons découvert les fonctions "ontimer" et "mainloop", permettant d'exécuter une fonction à intervalle régulier.
Malheureusement, celles-ci ne fonctionnaient pas de la manière prévue. Une fois le temps écoulé, même si l'exécution précédente n'était pas terminée, celle-ci était arrêtée et la fonction recommençait depuis le début. Cela a causé plusieurs bugs difficiles à diagnostiquer
Nous avons donc utilisé une simple boucle while
Le principe est donc d'effectuer les actions suivantes de 60 à 144 fois par seconde :

1. avancer la position de tous les objets
2. si l'objet est sorti de l'écran, on le repositionne à gauche
3. effacer l'écran
4. redessiner tous les astres dont la position a changé

Cependant, nous nous sommes rendus compte qu'effacer intégralement l'écran causait des problèmes visuels. Nous avons décidé d'utiliser une instance de turtle pour chaque objet affiché à l'écran.

2.3 Structures de données

Après nous être lancé dans ce projet, nous avons rapidement réalisé que stocker des objets sous forme de listes polymorphiques ne serait pas pratiques. En effet, en accédant à des propriétés grâce à un index, il est très facile de commettre une erreur. Nous nous sommes renseignés et avons découvert les "namedtuple", et, plus tard, les "dataclasses". Elles permettent de regrouper ensemble des données, et de donner un nom à chacune de ces données. Ainsi, au lieu de la liste suivante :

```
# position, tortue, taille, couleur
etoile = [( -5, 5 ), turtle.Turtle(), 5, "white"]

couleur = etoile[3]
```

Nous avons pu utiliser le code suivant :

```
Etoile = namedtuple('Etoile', ['position', 'turtle', 'taille'
etoile = Etoile( position = ( -5,5 ), turtle = turtle.Turtle(
couleur="white" )

couleur = etoile.couleur
```

Mais nous nous sommes rendus compte que les "namedtuple" sont partiellement "immutable", c'est-à-dire que certaines de leurs propriétés ne peuvent être modifiés. C'est ainsi que nous avons trouvé les dataclasses, une addition récente de Python.
Elles s'utilisent de la manière suivante :

```
@dataclass
class Etoile:
        position: turtle.Vec2D
        turtle: turtle.Turtle
        taille: float
```

```
    couleur : str

    # L'utilisation est ensuite identique aux namedtuple
```

2.4 Génération aléatoire

2.4.1 Arrière plan

Générer aléatoirement une étoile est facile. Il suffit d'une taille, d'une position, et il n'y a plus qu'à la dessiner.
Mais comment générer une galaxie ou une constellation ?
Nous avons eu recours aux coordonnées polaires. Grâce à celles-ci, nous n'avons plus besoin de coordonnées absolues. Seuls un point, une distance et un angle sont nécessaires. A partir de là, il est donc facile de s'assurer que la distance entre les étoiles d'une constellation reste constante.

2.4.2 Astéroïdes

En revanche, générer un astéroïde s'est avéré bien plus complexe.
Notre première idée était assez simple. On se place dans le coin supérieur gauche. On avance vers la droite avec une distance aléatoire, et un angle légèrement aléatoire. Une fois la limite dépassée, on procède de la même manière vers le bas, vers la gauche, vers le haut, et on relie le dernier point au premier point.
Cette approche présentait cependant deux problèmes :

- La taille était aléatoire. Il était impossible de s'assurer que l'astéroïde allait avoir la taille souhaitée.
- Le rendu final était parfois étrange.

Nous avons tenté de corriger ces problèmes, mais le code devenait de plus en plus complexe et nous avons fini par comprendre qu'il fallait trouver une autre méthode.
Nous avons finalement eu l'idée de revenir aux coordonnées polaires.
On se place au centre de l'astéroïde. On prélève un point à l'angle 0, avec un rayon aléatoire (compris entre un minimum et un maximum). On augmente l'angle d'un montant aléatoire, et on recommence. Une fois que l'angle dépasse 360 degrés, on arrête.
Le code résultant était beaucoup plus simple.

2.4.3 Destruction des astéroïdes

Après avoir créé un astéroïde, il fallait maintenant être capable de le détruire. Il fallait trouver un moyen qui fonctionnerait à chaque fois, pour tout astéroïde. Pour donner l'illusion d'une arme futuriste, nous avons créé un point dégradé au centre de l'astéroïde. Le code permettant de créer le dégradé a été trouvé sur StackOverflow. Pour l'afficher, il suffisait de faire des points de plus en plus petits, de couleur différente à chaque fois.
Mais il manquait quelque chose. Nous avons donc cherché à créer des "fissures". Leur réalisation a été légèrement compliquée, mais une fois encore, les coordonnées polaires nous ont permis de réussir. Nous avons aussi découvert la fonction atan2.

3 Problèmes rencontrés

Au delà des problèmes déjà cités dans les autres sections, nous avons rencontrés d'autres problèmes.

3.1 turtle.undobuffer

Un problème qui nous a fait perdre beaucoup de temps est le undobuffer de turtle.

En effet, lorsque l'animation se jouait, elle devenait de plus en plus lente. Nous avons vu que la mémoire utilisée par le projet augmentait de plus en plus, et nous en avons déduit que cela venait du undobuffer de turtle. Celui-ci, en théorie, permet de revenir en arrière. Mais lorsque des centaines de tortues sont présentes, la mémoire utilisée devient très vite bien trop grande.
Nous avons donc essayé de désactiver ce undobuffer avec `turtle.setundobuffersize(0)`. Sans succès. La tortue continuait à utiliser autant de mémoire.
Après beaucoup de recherches, et lecture du code source de turtle, nous avons fini par comprendre que la taille du undobuffer était réinitialisée en appelant `turtle.clear()`. Pour le corriger, il fallait définir undobuffersize lors de l instantiation de la tortue. Cela n'est pas précisé dans la documentation, et nous avons d'ailleurs signalé cette erreur.

4 Répartition du travail

Neil Dahoun s'est chargé des formes de base, des fonctions principales du jeu et de la recherche du style graphique.
Edgar Bizel, étant plus expérimenté, s'est chargé de l'animation et de la génération des objets complexes.

5 Travail restant

Le travail effectué est déjà très complet, mais l'on pourrait imaginer des fonctionnalités supplémentaires, comme la possibilité de faire une partie à deux joueurs (en local voire même en réseau), ou la possibilité de rejouer directement après avoir terminé la partie.