Rapport TD LOG

github : https://github.com/Bilal59170/TDLOG_Space_Wars

Mathis Wauquiez, Bilal Ben Moussa, Enzo Azzoug,
Bronislav Abadie, Alex Crozat (malade)

January 2024

1 Ojectifs initiaux

1.1 Le Jeu Diep.io

Notre objectif en travaillant sur ce projet a été de recréer le jeu "diep.io". Le jeu consiste à piloter avec les touches haut, bas, droite, gauche un vaisseau sur une carte en 2 dimensions. Lorque le joueur effectue un clic gauche, le vaisseau tire dans la direction de la souris. La carte contient d'autres joueurs et leurs vaisseaux, et des polygones représentant des astéroïdes que l'on peut détruire avec l'arme du vaisseau afin d'amasser de l'experience (XP). L'objectif est alors d'obtenir le maximum d'XP au fur et à mesure de la partie sans détruire son vaisseau.

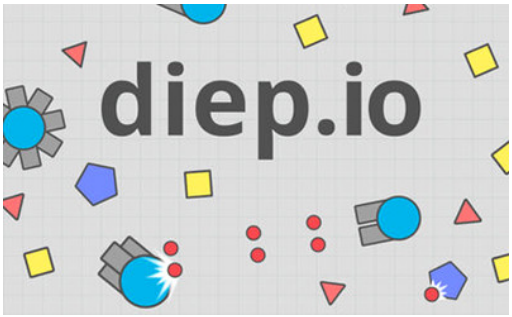

Figure 1: Image de *Diep.io*

Le vaisseau a des attributs qu'on peut améliorer en gagnant de l'XP dans une partie donnée. En gagnant de l'XP le vaisseau monte en niveau. A chaque niveau passé le joueur peut améliorer ces attributs. Des exemples de ceux-ci sont la vitesse de projectile, les dégâts, ou la santé parmi d'autres. En plus de ces attributs, en passant le niveau 15, 30 et 45 le vaisseau peut effectuer une amélioration de classe. Ces améliorations sont plus conséquentes et permettent

de changer la nature du vaisseau : cela permet une diversité des stratégies dans le jeu. En effet certains vaisseaux peuvent avoir une grande mobilité avec peu de santé et d'autres peuvent avoir une vitesse de tir et des dégâts élevées en étant très peu mobiles.

1.2 Notre jeu: Pied.io

1.2.1 La description du jeu final Pied.io

On décrit le fonctionnement du jeu dans cette section en passant rapidement sur les éléments similaires à Diep.io expliqués en première partie. On l'explique en se mettant à la place d'un joueur qui découvre le jeu.
Tout d'abord, au lancement du jeu, un menu d'entrée simpliste s'affiche. Il est composé des boutons usuels "Jouer" et "Quitter", mais aussi d'une encoche pour permettre au joueur de mettre son pseudo. Enfin, un tableau des scores des joueurs précédents classés dans l'ordre décroissant de haut en bas est affiché. Naturellement, lorsque le bouton "quitter" est appuyé, la fenêtre se ferme et le programme s'arrête. À l'opposé, écrire son pseudo et appuyer sur "jouer" permet de commencer une partie.
Le joueur apparaît sur une carte avec des entités similaires à Diep.io (astéroïdes natures différentes et ennemis de niveaux différents). Il contrôle un petit vaisseau triangulaire rouge qui possède de la vie (barre rouge en dessous du vaisseau), de l'expérience, et des ressources. Les ennemis sont aussi triangulaires et sont de niveau variés.

Figure 2: Image d'une partie de *Pied.io*

La carte est plus grande que la fenêtre d'affichage (la caméra virtuelle du jeu) et elle est représentée en haut à droite de l'écran par un rectangle aux arrêtes noires. Le joueur, lui, est représenté par un point vert tandis que la caméra, est représentée par un rectangle blanc. La caméra est toujours centrée sur le joueur sauf lorsque celui ci est sur les bords de la carte. Cependant, ces bords sont justes visuels car il est en fait possible pour le joueur de passer d'un

bord à l'autre. Par exemple, si le joueur traverse le bord gauche, il se retrouvera sur le côté droit de la carte. La caméra se déplace alors directement pour se recentrer sur le joueur lorsque celui-ci se "téléporte" d'un bout à l'autre de la carte.

Il est possible pour le vaisseau du joueur et pour les ennemis de tirer des projectiles. Le joueur gagne de l'expérience en détruisant les astéroïdes et en tuant les ennemis. Il perd de la vie en se faisant toucher par un astéroïde ou par un projectile émit par un vaisseau ennemi. En gagnant suffisament d'expérience, le joueur monte en niveau et ses caractéristiques (vie, dégâts de tir par exemple) sont améliorées. Il y a au total quatre niveaux mais pas de limite de d'expérience après le quatrième niveau. Nous avons fixé le but du joueur comme celui de Diep.io c'est à dire amasser le plus d'expérience possible.

Les astéroïdes et les ennemis peuvent, comme le joueur, traverser les bords de la carte. Cependant, nous avons choisi de faire disparaître les projectiles lorsqu'ils sortaient de la carte visible pour éviter une surcharge de projectiles. Nous détaillons la programmation des ennemis dans la partie technique du projet.
Lorsque le joueur meurt, il s'affiche le redouté écran de mort avec deux boutons, "Quitter" ou "Rejouer". Le pseudonyme et le score du joueur s'affichent, ce qui est assez classique dans les jeux vidéos. Le bouton quitter ne fait pas revenir à l'écran d'accueil, mais quitte directement le jeu.

Figure 3: Menu de mort de *Pied.io*.

1.3 Idées non implémentées

Nous présentons les idées qu'on a eu sans avoir eu le temps de les implémenter, la principale étant le mode multi joueurs.

1.3.1 Mode multi online

Une fois le jeu avec un seul joueur contre des ordinateurs fait, l'idée était d'implémenter un mode multijoueur qui ressemble à celui de *Diep.io*. Même si nous n'avons pas eu le temps d'implémenter ce mode, on explique brièvement les idées auquelles on a pensé en commençant d'abord par l'aspect réseau.

Le changement majeur par rapport au jeu à un seul joueur est qu'il faut prendre en compte plusieurs ordinateurs et les mettre en lien par **réseau local**. Ainsi, le jeu ne s'exécute pas sur les ordinateurs, mais sur un serveur qu'on voulait créer. Comme le jeu à un seul joueur est déjà implémenté, les seules difficultés à surmonter auraient été les interactions ordinateur-serveur avec la création de sockets appropriés et la gestion de décalage réseau (choix de fonctionnement par flux ou datagramme par exemple) dû à la qualité du socket.

Pour ce qui est du déroulement du jeu, nous avons pensé à plusieurs modes de jeu multijoueur. Tout d'abord, un mode collaboration entre joueurs contre les ordinateurs (mode coopératif) puis des modes où les joueurs s'opposent en équipe ou chacun pour soi (modes versus). Nous n'avions cependant pas fixé le but des joueurs dans le cas coopératif mais les deux choix naturels sont soit de tuer tous les ennemis (si le spawn des ennemis s'arrête à un moment) soit de tuer le plus d'ennemis possibles (en un temps infini comme dans le jeu en solitaire ou en un temps fini). La remarque est la même dans les modes versus mais le rôle des ennemis est pris par les joueurs adverses (avoir le meilleur score que les concurrents ou bien tous les vaincre).

Les modes versus amènent la gestion des apparitions (dit *spawn* en anglais) des joueurs. En effet, faire apparaître deux joueurs adverses l'un à côté de l'autre n'est pas ce qu'on souhaite car le combat commence directement sans que les joueurs aient pu gagner en niveau. Pour le cas du mode versus en équipe, il y aurait alors une zone de "spawn" éloignée pour chaque équipe, où les astéroïdes sont absents. Cette zone peut être aussi une zone de protection, un bastion pour l'équipe où les autres joueurs ne peuvent ni rentrer, ni tirer dedans. Pour le mode versus en chacun pour soi, les appartitions des joueurs seraient plutôt uniformément réparties sur la carte, tout en s'assurant de ne pas faire apparaître un joueur dans un astéroïde.

1.3.2 Site interne et IA ennemi

Nous avions aussi comme idée de mettre le jeu sur un site internet accessible à tous où le tableau des scores serait mis à jour directement sur internet. Cette idée cependant a été rapidement éloignée du fait de l'interface graphique que nous avons choisi de réaliser en python.

Une dernière idée a été de faire des IA neuroévolutives. Le concept aurait été de créer des IA jouant au jeu, s'affrontant les un les autres, et apprenant

des stratégies au cours du temps. Cette IA aurait été basée sur l'algorithme NEAT, pour NeuroEvolution of Augmenting Topologies, qui reproduit les règles de l'évolution génétique à l'entraînement d'un réseau de neurones très simple, à topologie "augmentative" (qui est amenée à évoluer). C'est un algorithme qui est très adapté pour les simulations simples, et qui convient en général bien pour les jeux. Le réseau de neurones prend en entrée des caractéristiques que nous avons décidées, qui aurait par exemple pu être la distance avec les ennemis les plus proches, ou avec les astéroïdes, et qui en sortie donne la direction à suivre, la cible sur laquelle tirer.

Cette IA aurait été entraînée en temps réel au cours d'une simulation dans laquelle les IAs qui aurait le meilleur score survivraient le plus longtemps, et auraient le plus de chances de se "reproduire" (comprendre par là faire une réplique du réseau de neurones, avec une "mutation", une petite variation de la topologie ou des poids du réseau). Au cours des générations, les IAs restantes seraient celles qui ont le meilleur score.

Une fois l'apprentissage terminé, nous aurions pu les faire jouer contre des joueurs réels pour voir qui était le meilleur.

2 Choix techniques

Nous avons fait le choix de travailler selon une méthode Agile, et en intégrant différents niveaux de prototypage. Au début, nous avons fait le choix d'avoir une équipe sur le premier prototype, et une seconde sur l'architecture, afin que la première équipe (Bilal, Enzo et Bronislav) se familiarise avec Pyglet pendant que la deuxième (Alex et Mathis) décide de l'architecture à employer.

Comme pyglet présentait assez peu de fonctionnalités de haut niveau (comprendre par-là les collisions entre objets par exemple, ainsi que certains événements), et dans un souci de pouvoir rapidement adapter notre code à de nouvelles fonctionnalités, telles que le mode multijoueur par exemple, nous avons choisi d'implémenter nous même nos propres classes de Sprites (un sprite est globalement une petite image représentant un objet, un personnage ou une entité dans un jeu vidéo), sous-classant une classe "Entity" qui permet la gestion des coordonnées des objets. Toutes ces classes sont implémentées dans une partie séparée du code, nommée "game_engine", utilisée comme un module, comprenant les fichiers suivants : "entity.py", où l'on gère les entités, "sprites.py", qui sert à implémenter différents types de sprites, "collisions.py", qui sert à implémenter des méthodes de détection de collisions entre les différents types de sprites, utilisé directement dans l'implémentation des sprites, "profiling.py", qui sert à profiler le temps d'exécution de différentes fonctions et à l'afficher au vol, "config.py", qui définit les variables du jeu (peut-être aurait-il été judicieux de faire une configuration pour le jeu ainsi qu'une pour la partie "game engine", et enfin on trouve le fichier "utils.py" qui contient diverses fonctions utilisées.

Ainsi, pour être plus précis, la classe "Entity" permet de connaître pour chaque entité sa position sur la carte ainsi que sa position sur l'écran, et de l'actualiser à chaque boucle de jeu pour chaque entité, selon sa vitesse. Initialement, on avait prévu deux classes "Entity" différentes : en plus de celle existant, il y aurait eu une classe "OnlineEntity" calculant la position sur la carte et à l'écran en utilisant une connexion à un serveur de jeu et en la déduisant de la dernière position et dernière vitesse connues (interpolation linéaire).

Cette gestion des entités simplifie l'écriture des différentes classes de sprites. Les différentes classes de sprites implémentent toujours trois méthodes, qui parlent d'elles-mêmes : "draw" pour l'affichage du sprite, "collides" pour voir s'il intersecte un autre sprite, et "is_on_screen", utilisée pour optimiser la vitesse d'affichage.
Il en existe trois types : les images / animations, les polygones, et les cercles. Initialement, nous avions prévu d'implémenter des lasers, ainsi que des vaisseaux spatiaux s'affichant avec des images. C'est l'utilité prévue de la classe " Image ". Elle aurait été utile notamment pour gérer les collisions entre les vaisseaux spatiaux et les lasers. Dans la version finale du code, cette fonctionnalité de collision image-image n'est pas utilisée. En revanche, la classe Image est utilisée pour afficher les animations d'explosion.
La classe Polygon, elle, est utilisée partout : elle permet l'affichage des astéroïdes, l'affichage du vaisseau, des ennemis, et la gestion des collisions de ces différentes entités.
La classe Circle, elle, est utilisée pour gérer la collision entre les projectiles (qui ont finalement l'apparence d'une étoile mais sous-classent quand même une classe de sprite cercle) et les différentes autres entités, via les collisions polygones-cercle principalement.

Pour fonctionner, ces différentes entités ont besoin de certaines informations sur le jeu, par exemple : quelle est la taille de la fenêtre ? Quelle est la taille de la carte ? Quelle est la position du joueur principal ? etc. Il est donc naturel de rattacher toutes ces entités à une instance de classe spécifiant toutes ces informations. Cette classe correspond à la classe Game, que nous avons choisi d'implémenter en dehors de l'engin de jeu car il n'est pas générique, et qui implémente bon nombre de méthodes.
C'est le fichier le plus long du projet, avec plus de 530 lignes de code. Dans un souci de clarté, au vu de la complexité de ce système, il contient deux classes : la classe GameEvents, chargée de gérer les différents événements du jeu, comme les collisions par exemple, et la classe Game, sous classant GameEvents. Pour plus d'informations, se référencer à la documentation de la classe GameEvents.
La classe Game contient tout ce qui est nécessaire pour faire tourner le jeu, sans pour autant rajouter la logique propre au jeu : il contient la gestion de la fenêtre et de l'affichage, les différentes listes d'entités, avec les listes d'astéroïdes et d'ennemis notamment, les informations sur l'état de la partie comme le score du joueur ou s'il est mort ou non, ... Tout comme la classe Ship, il sous

classe " pyglet.event.EventDispatcher ", ce qui permet de créer des méthodes " on_[event_name] " appelées automatiquement par pyglet à chaque événement se passant sur la fenêtre. Par exemple, " on_close " est appelé à la fermeture de la fenêtre et permet de terminer le programme.
Cette classe implémente aussi la boucle de jeu, qui consiste à d'abord gérer les événements, puis appeler une fonction " update ", qui réalise un peu de logique de jeu, appelle la méthode " tick " des entités, appelle la méthode d'affichage, et attend si nécessaire afin de respecter le taux de rafraîchissement souhaité.
Le choix de réaliser de la logique de jeu dans la méthode update plutôt qu'en utilisant l'événement @game.each(1) qui appelle la fonction décorée à chaque boucle de jeu est questionnable.
Ainsi, à chaque instance de jeu est rattachée plusieurs entités. Ces entités sont implémentées dans le dossier " game_objects ", qui référence les astéroïdes " asteroids.py ", les ennemis (" enemies.py "), les projectiles (" projectiles.py "), le vaisseau principal (" ship.py ") et des animations d'explosion. Ces classes implémentent une méthode " tick ", appelée à chaque boucle de jeu, qui sert à la gestion de la position (en appelant la méthode tick de la classe entity dont on hérite), et à la logique de jeu propre à l'objet considéré.
Sans rentrer trop dans les détails, l'affichage est effectué en affichant chaque entité séparément via leur méthode " draw " et en appelant aussi une fonction " static_display " s'occupant d'afficher les éléments fixes du jeu et le menu de mort une fois le joueur mort. Cette fonction " static_display " est définie dans le fichier " UI.py ".
Ce fichier implémente aussi le menu de démarrage, StartMenu, appelé avant le lancement de la boucle de jeu principale. Cet appel est effectué dans le fichier " main.py ", où l'on instancie le menu de démarrage, en spécifiant que le bouton de démarrage appelle une fonction instanciant la partie, avec le nom du joueur choisi sur le menu de démarrage, où l'on active la gestion de la logique de la partie, et où l'on démarre la partie.
Ces éléments de logique sont implémentés dans le fichier " game_logic.py ", où l'on s'occupe de la gestion de l'apparition des ennemis et des astéroïdes, des collisions astéroïde/astéroïde, projectile/astéroïde, projectile/vaisseau, projectile/ennemi, astéroïde/vaisseau, astéroïde/ennemi, et où l'on s'occupe (ou non) de l'affichage du taux de rafraîchissement réel.
Ainsi, ce choix d'architecture, fondée sur des classes de sprites communes aux méthodes abstraites, nous as permit de travailler en parallèle de façon très efficace, et a été récompensée par un très faible nombre de conflits lors des merges github ainsi qu'une clarté de code améliorée.

3 Réalisation

Dans cette partie nous allons détailler par étapes comment nous avons procédé pour la réalisation du projet. Nous sommes partis de rien et il y a eu deux grandes parties dans cette réalisation : avant et après nos premiers affichages de notre vaisseau.

3.1 Avant le premier affichage de notre vaisseau

Pour commencer il fallait tout d'abord choisir une bibliothèque en python pour coder notre jeu. Après quelques recherches notre choix s'est porté sur Pyglet (et non Pygame par exemple) choisie pour sa simplicité d'utilisation. Ensuite il a donc fallu se renseigner sur la documentation Pyglet pour savoir quelles fonctions utiliser pour nos différentes fonctionnalités.

Dans le même temps nous avons réfléchi à la structure du code avec différentes classes qui héritent les unes des autres avec par exemple une classe pour les vaisseaux, les astéroïdes qui héritent de la classe mère Entité. Nous nous sommes ensuite rapidement réparti le travail entre nous. Il y avait un sous groupe qui travaillait sur l'aspect graphique (affichage du vaisseau et des projectiles) et un autre qui travaillait sur les mécaniques du jeu (visée avec la souris, déplacement du vaisseau...). Cette partie de la réalisation fut longue et difficile car c'était un peu comme si on avançait dans le noir.

3.2 Avancée avec affichage du vaisseau

Lorsqu'on a enfin réussi à afficher notre vaisseau à l'écran on a pu vraiment implémenter beaucoup plus de fonctionnalités car on pouvais directement les tester. On a d'abord testé les fonctionnalités qu'on avait déjà implémenté pour les améliorer (déplacements et visée notamment). Ensuite on a continué à travailler avec les mêmes sous groupes qu'avant. Au niveau de la méthode de travail on se concentrait chacun sur une fonctionnalité voire une sous-fonctionnalité puis on l'implémentait. Cela nous a permis surtout pendant ces dernières séances de vraiment pouvoir développer beaucoup de fonctionnalités à chaque séances.

Par exemple pour les ennemis : on a commencé à créer des instances en se basant sur ce que l'on avait codé pour notre vaisseau, puis on a géré le spawn d'ennemis, leurs déplacements, le fait qu'ils tirent, puis qu'ils nous infligent des dégâts et qu'on puisse leur en infliger et enfin on a implémenter différents niveaux d'ennemis avec des statistiques propres à chaque niveau.

Sur les dernières séances nous avons eu le besoin de jouer au jeu pour tester quelques cas limites et cela nous a permis de détecter plusieurs bugs que l'on a pu rapidement corriger.

Et bien-sûr l'usage de Git nous a aidé a partager les fonctionnalités ajoutées et travailler sur des branches différentes pour d'un côté les fonctionnalités graphiques et de l'autre les fonctionnalités du jeu. Notre cordination au sein de l'équipe nous a permis d'obtenir un jeu satisfaisant bien qu'avec plus de temps nous aurions pu davantage le développer.

4 Difficultés rencontrées

4.1 Compréhension des librairies et du travail de chacun

Certaines personnes du groupe avaient déjà une expérience dans le développement de jeu-vidéo (Mathis, Alex). Ainsi, Mathis a proposé des idées d'architecture de jeu (arborescence de classes, structure de dossier, etc.) qu'il a développé de son côté pendant que les autres membres du groupe se familiarisaient avec Pyglet. En effet, la programmation objet étant assez récente pour certains, il a fallu un certain temps à certains pour acquérir des automatismes/intuitions sur la création de classes qui peuvent servir dans un jeu vidéo. Par exemple, il a fallu beaucoup de temps avant que l'on arrive à afficher des formes basiques sur une fenêtre, puis arriver à effectuer des actions qui détectent les clics et la position de la souris (pour faire tourner le vaisseau en fonction de la position du curseur).
Après que Pyglet ait bien été pris en main par tous, il a donc fallu comprendre la structure de jeu présentée par Mathis pour pouvoir ensuite implémenter les fonctionnalités du jeu. Par exemple, comprendre comment les animations sont implémentées pour pouvoir afficher une animation d'explosion pour la mort du vaisseau du joueur. Ainsi, la difficulté était double. Il a fallu dans un premier temps apprendre les mécaniques de base d'un jeu vidéo et se familiariser avec Pyglet, puis arriver à remettre tout le monde à niveau sur la compréhension de l'architecture choisie.

4.2 Difficultés diverses

4.2.1 Les fenêtres de début et de fin

Une des étapes de notre projet a été celle de l'implémentation de la fenêtre de départ et de mort du vaisseau. Nous avons eu des problèmes dans l'intégration de ceux ci au sein de notre jeu. En particulier nous avons passé un certain temps sur les bouttons "quitter" et "rejouer" et sur l'affichage du score qui ne fonctionnaient pas de la même manière selon chez tout le monde. Il a été nécéssaire de bien lier le lancement et le relancement du jeu à l'implémentation de ces fenêtres.

4.2.2 L'équilibrage du jeu

Les vaisseaux et les ennemis comportent de nombreux paramètres qu'il a été nécéssaire de bien calibrer afin de rendre le jeu jouable. Il fallait faire attention aux paramètres à chaque nouveau niveau franchi par le joueur. Par exemple, nous avons implémenté différents types d'ennemis qui sont plus ou moins forts et qui apparaissent plus ou moins selon le niveau du joueur. La fréquence d'apparition de ces ennemis est donc importante a bien choisir. Un autre exemple est celui du déplacement des ennemis. Ceux-ci se déplacent de la manière suivante. Lorsque leur distance au joueur est supérieur à un certain seuil (le "engage radius"), ils s'avancent vers le joueur. Lorsqu'ils sont trop proches et

que leur distance au joueur est inférieure à un certain seuil (le "caution radius") ils se déplacent dans la direction opposée et s'éloignent du joueur. Si l'engage radius est trop grand alors les ennemis restent très loin du joueur. A l'inverse si le caution radius est trop petit les ennemis vont avoir tendance à submerger le joueur et à rendre le jeu trop difficile.

4.2.3 Version de pyglet

Au cours du projet, nous avons également rencontré des difficultés sur la version de pyglet utilisée. En effet, courant novembre, pyglet a silencieusement sorti sa version 2.0. Cette mise à jour est venue avec bon nombre de changements, et la documentation n'était pas à jour. Nous avons donc passé beaucoup de temps à essayer de comprendre pourquoi les exemples de la documentation ne fonctionnaient pas de notre côté, avant de nous apercevoir du problème et de passer à la version antérieure. Certaines personnes ont eu du mal à mettre à jour la version de pyglet sur l'environnement python de travail, le mettant à jour sur un environnement différent et pensant utiliser la version antérieure et utilisant en réalité la version 2.0.

4.2.4 Problème de lag

Au cours du projet, il a été difficile d'opitimiser la vitesse de l'affichage des objets pour minimiser le lag.
L'implémentation d'une fonction de détection de collisions entre polygones efficace a également été très difficile : après une première approche naïve avec des détections de chevauchement ligne-ligne, une approche plus complète en utilisant le théorème de séparation des axes a été utilisée, en utilisant aussi la vectorisation via numpy ainsi que la compilation du code python grâce à numba (décorateur njit pour no just in time), qui se charge d'inférer le type des variables pour compiler les fonctions, ce qui améliore grandement la rapidité des boucles.