Rapport final - Virus Gate

Ce jeu nous a demandé un certain temps, mais nous a permis d'apprendre sur les difficultés réelles pour la réalisation de certaines choses en Python, à savoir des choses qui semblaient complexes mais qui se sont avérées relativement simples à réaliser, ou à l'inverse des choses simples en apparence mais complexes à développer en pratique.

Outils utilisés

Python - https://www.python.org :

PyGame - https://www.pygame.org :

PyInstaller - https://pyinstaller.org :

Dans le chapitre 2, une fonction utilisée a également été tirée de StackOverflow -

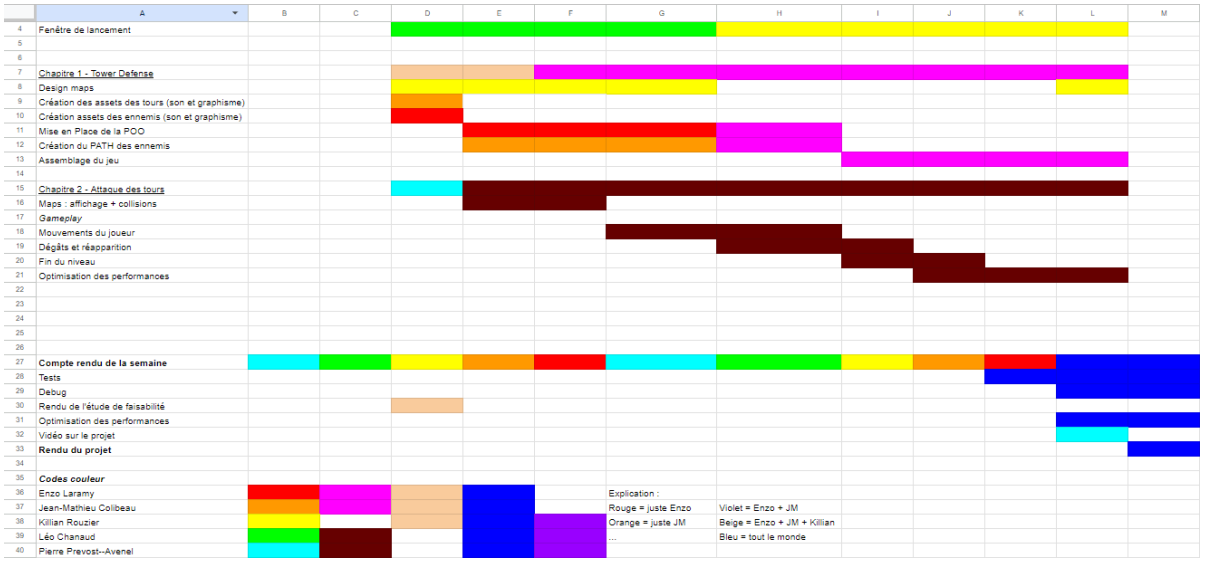
https://stackoverflow.com :

Organisation du projet

Pour réaliser les jeux et l'accueil en 12 semaines, nous nous sommes répartis par groupe de deux pour les deux jeux, et l'accueil sera codé par une seule personne à la fois.

<u>En résumé :</u>

- Pierre travaille sur le chapitre 2
- Jean-Matthieu et Enzo travaillent sur le chapitre 1
- Léo travaille dans un premier temps sur l'accueil pour initialiser les variables de base ainsi que les recherches pour l'utilisation de Pygame. Ensuite il rejoint Pierre sur le second chapitre 2
- Killian réalise dans un premier temps les assets concernant la carte pour le chapitre 1, puis en suivant il prendra le relai pour le code de l'accueil



Ci-dessus le diagramme de Gantt représentant la répartition initiale des tâches pour ce projet

Accueil - Léo (code) + Killian (code)

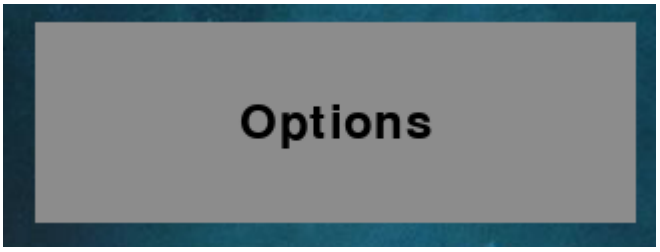
Introduction :
Dans le cadre d'un projet de développement pour un jeu vidéo, nous avons créé une page d'accueil interactive à l'aide de la bibliothèque Pygame (une librairie en langage Python). Cette page d'accueil comprend plusieurs éléments tels qu'un fond d'écran, une bande sonore, ainsi que des boutons pour accéder aux différents chapitres du jeu, ajuster les options de volume de la bande sonore et la luminosité, et quitter l'application.

Mise en place de l'interface utilisateur :
La page d'accueil a été conçue avec une interface utilisateur intuitive pour offrir une expérience agréable. Un fond d'écran attrayant a été choisi pour créer une ambiance immersive dès le début du jeu. La bande sonore a également été intégrée pour ajouter une meilleure expérience. Des boutons graphiques ont été ajoutés pour permettre aux joueurs d'interagir avec l'interface.

Voici les différents boutons disponibles :

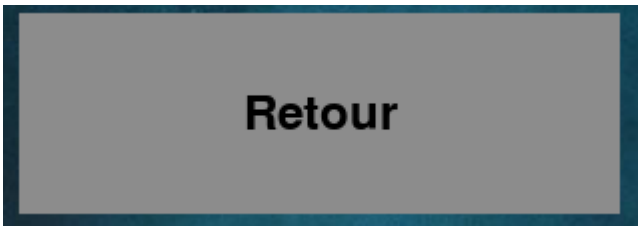
> <u>Boutons de chapitres</u> **:**
> Deux boutons, "Chapitre 1" et "Chapitre 2", ont été créés pour permettre aux joueurs d'accéder aux différents niveaux du jeu. Lorsque l'un des deux est pressé, le joueur est redirigé vers le niveau correspondant du jeu. Cela a été implémenté en utilisant les fonctionnalités d'interaction de Pygame pour détecter les clics de souris et effectuer les actions voulues. Ce bouton est disponible dans le menu principal.

Bouton d'options **:**
Un bouton "Options" a été ajouté pour permettre aux joueurs d'ajuster les paramètres de la bande sonore et de la luminosité. Lorsque ce bouton est appuyé, une fenêtre s'ouvre avec des barres avec un curseur pour régler le volume de la bande sonore et la luminosité du jeu. Les valeurs de ces options sont stockées et mises à jour en conséquence dans les chapitres. Ce bouton est disponible dans le menu principal.



Bouton de fermeture **:**
Un bouton "Quitter" a été ajouté pour permettre aux joueurs de fermer l'application. Lorsque ce bouton est cliqué, l'application se ferme proprement en utilisant les fonctionnalités de Pygame pour quitter l'application (“pygame.quit()”). Ce bouton est disponible dans les deux menus.

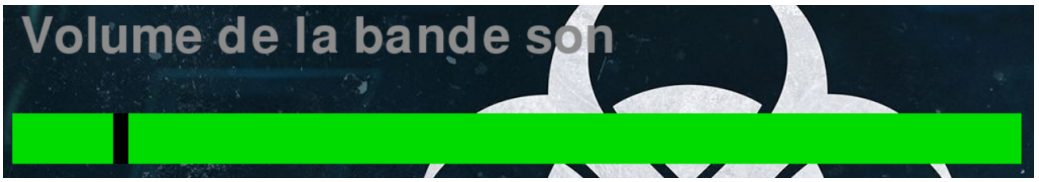
Bouton “retour” :
Enfin, un bouton nommé “retour” à été ajouté pour retourner au menu principal, il est donc disponible seulement depuis la page des options.



Voici les différentes barres de configuration (uniquement disponible dans la page des options) :

Barre “volume” :

Une barre avec son curseur amovible a été ajouté afin de permettre au joueur de régler le volume de la musique si le besoin se présente.



Barre “luminosité” :

Pour finir, une barre avec son curseur amovible a été ajouté afin de permettre au joueur de régler la luminosité de l’écran, si le besoin se présente.

Problèmes principaux rencontrés :

Ici nous allons simplement faire part des difficultés que le code a pu nous donner lors de sa réalisation,ainsi que les changements apportés au cahier des charges (les idées que nous avions à l’origine).

- Le premier et vrai problème tout au long a été la “propreté” du code, autrement dit son organisation. Nous n’avions pas eu l’idée de réaliser les boutons ainsi que les barres pour configurer le son et la luminosité avec des classes. Ainsi nous avons réalisé tardivement cette optimisation, après avoir compris que rédiger toutes les variables concernant chaque boutons à la main serait tâche pénible, et accessoirement bien trop coûteux en place : le nombre de variables à initier serait énorme. une fois les classes faites, il n’a fallu que les mettre dans un coin, et initier les boutons ainsi que les barres facilement juste en dessous. Le code après modification est bien plus réduit

et compréhensible, en plus de rendre bien plus simple la création d'un nouveau bouton ou nouvelle barre.

- En suivant ce nouveau système, le code a dû changer en suivant, et nous avons été confronté à un second problème, plus léger cependant : la détection au clic des boutons. Au lieu de se compliquer la tâche à chaque détecter de ressortir les clés concernant les coordonnées dans l'espace des boutons, etc, nous avons confectionné une fonction directement dans la classe ("is_clicked()") prenant simplement en argument la position de l'événement du clic ("event.pos", obtenable avec "pygame.event.get()").

- Ainsi avec tout ceci de réalisé, la création de boutons et barres (ainsi que leurs détections) sont choses aisées, nous avons pu nous attaquer au deuxième gros point concernant la page d'accueil : ces différents menus. C'est au niveau de l'architecture des boucles que nous nous sommes questionnés. Dans un premier lieu nous avons fait une grosse boucle principale, où tout pouvait être détecté, et il fallait simplement vérifier à l'aide des variables "in_menu" et "in_options" pour savoir s'il fallait considérer le/la bouton/barre ou non, et avec ceci en fonction des variables une fonction se déclenche pour actualiser la page d'un point de vue graphique.

- L'idée fonctionnait, cependant après avoir commencé à ajouter différents boutons, nous avons jugés que tout regrouper en un seul endroit n'était pas pratique pour se repérer après avec les yeux, en plus de devoir initier par exemple un bouton pour la page des options mais non dans la fonction "afficher_options()". Ainsi nous avons réfléchi à une seconde méthode pour architecturer tout ceci : garder une boucle principale bien plus légère, servant simplement de "direction" en fonction des deux variables, et ce serait dans les différentes fonctions que se retrouveront les boucles avec seulement les objets (boutons/barres) la concernant.

  Ainsi nous avons gardé cette architecture car en plus d'être bien plus agréable pour l'initiation de nouveaux objets, cela simplifie le debug du code, car avec la première méthode le code faisait des aller-retours de partout entre les fonctions et la boucle principale, alors que désormais il change de fonction, et reste dedans tant que l'utilisateur ne souhaite pas changer, au lieu de faire boucle puis fonction puis boucle puis fonction etc etc.

- Autre léger souci mais notable, a été le positionnement des objets en fonction de la taille de l'écran. Avant nous nous contentions de simplement définir les variables x et y sans réfléchir davantage, mais c'est une fois passé sur ordinateur portable que la page d'accueil est devenue trop grande. Alors nous avons initié les variables de la largeur et de la hauteur de l'écran, pour définir l'emplacement des différents objets, sans avoir le problème précédent.

Conclusion :
La page d'accueil interactive pour accéder aux jeux vidéo est fonctionnelle, offrant aux joueurs la possibilité d'accéder aux différents chapitres du jeu, d'ajuster les options de bande

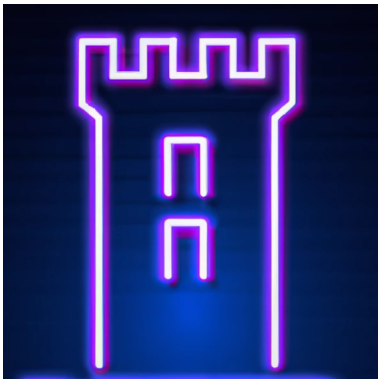
sonore et de luminosité, ainsi que de quitter l'application. Cette page d'accueil offre une expérience utilisateur immersive et engageante pour les joueurs, créant ainsi une base solide pour le développement du jeu vidéo.

Chapitre 1 - Enzo (code) + JM (code) + Killian (textures)

Introduction :

Note: Nous avons décidé de coder en respectant les normes internationales du codeur afin de se mettre dans la peau d'un codeur professionnel, nous avons donc mis notre code en 'anglais' c'est-à-dire le nom de variables, des fonction, des classes et des méthodes des classes. Certains de nos sprites sont récupérés sur internet et sont sous licence, merci de ne pas partager le dossier à un tiers ou d'essayer de le commercialiser.

Dans ce premier chapitre, on a voulu s'inspirer du style de jeu "Tower Defense". Le but, c'est que le joueur place des tours de manière stratégique pour contrer l'invasion des virus informatiques. Ces virus attaquent la carte réseau et leur objectif, c'est d'atteindre la carte mère.



Les virus ont des scripts qui leur donnent des caractéristiques spéciales comme la régénération de la vie, l'augmentation de la vitesse, la multiplication, ou encore la capacité à renforcer ou booster les autres virus à proximité. Ça rend le jeu plus intéressant et moins répétitif.

Les ennemis arrivent en vagues successives. Si le joueur parvient à empêcher suffisamment d'ennemis d'atteindre la carte mère après toutes les vagues, il gagne le jeu.

Heureusement, l'ordinateur a aussi son propre système de défense, divisé en quatre catégories principales : la défense de zone, le ralentissement, la défense ciblée et l'exécution. La réussite du joueur dépend beaucoup de sa capacité à comprendre et utiliser efficacement ce système de défense.

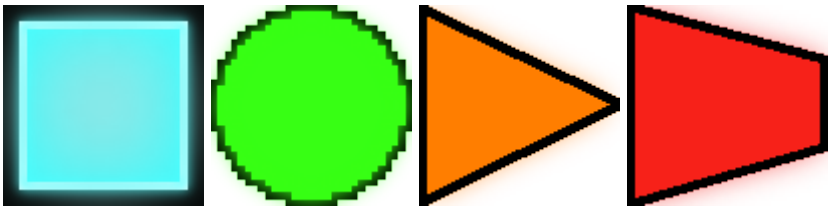
Pour faciliter la gestion des éléments en jeu, nous avons créé des classes telles que Tower et Enemy, cependant, nous avons rencontré quelques difficultés et problèmes quand on à voulu les implémenter.

Touche utile pour le jeu :

Pour vendre une Tower

Les Classes :

- **Towers** :



Comme le système de défense de l'ordinateur comprend 4 grands axes, nous avons implémenté 4 tours qui représentent chaque axe. La meilleure façon de faire avec python et pygame, c'est d'utiliser les classes, car grâce à celles-ci, on peut créer un 'squelette' général qu'on à appelé 'TowerDefense' qui comprend des méthodes communes à chaque tours comme 'draw' qui permet d'afficher la tour à l'écran, 'set_target' qui permet à la tour de sélectionner un ennemi particulier à toucher (à prendre en compte que cette méthode n'est pas utilisée par toutes les towers, car il y'en à qui attaquent en zone), 'in_range' qui sert à vérifier si un ennemi est assez proche pour être attaqué, 'sell' qui permet à l'utilisateur de vendre sa tour à tout moment, 'update' qui permet à la tour d'actualiser chacune de ses actions tous les ticks et enfin 'toggle_range' qui permet d'afficher ou non la range de la tour.

Grâce à ce squelette, nous avons ensuite créé des classes pour chaque tower car elles ont toutes des caractéristiques distinctes, c'est donc utile de le faire, c'est la que va nous servir la classe parente 'TowerDefense' car en la plaçant en argument de la classe enfant, (on en à 4 pour les 4 towers différentes) celle-ci héritera de toutes les méthodes mais aussi de tous les attributs de la classe parente donc on se retrouve avec une classe enfant qui est plus lisible et dans laquelle on doit juste se charger d'implémenter ses caractéristiques.

On à commencé par la ‘SquareTower’ car c’était selon nous la plus facile à implémenter et avec laquelle on fera les tests les plus faciles, car le style de jeu "Tower Defense" est une mécanique nouvelle et plus complexe que d’autres jeux donc il fallait se familiariser avec.
On à donné les attributs nécessaire à la classe ‘SquareTower’ comme ‘original_pos’ pour garder en mémoire sa position originale afin de placer une base à l’endroit de la tour si jamais le joueur décide de la vendre (une base sert à replacer une tower), ‘cost’ qui définit le coût de la tour, l’attribut ‘range’ qui définit la portée de la tour …

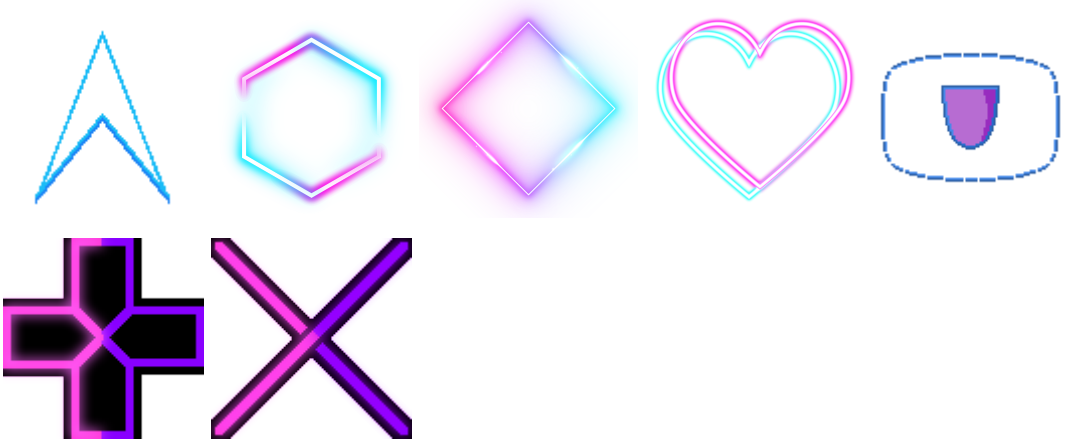
Ensuite, nous avons définit les méthodes de cette classe, la Square Tower représente l’axe de ralentissement du système de défense, donc on à créé la méthode ‘slow_enemies_in_range’ qui permet comme son nom l’indique, de ralentir tous les ennemis dans sa portée, et c’est là que la classe parent Tower Defense nous sert car on avait définit la méthode ‘in_range’, on à donc pu s’en servir. La méthode de ‘slow_enemies_in_range’ prend en paramètre un groupe d’ennemis et leur applique individuellement la méthode ‘slow’, qui est aussi dans la classe ‘Enemy’ et qui gère le ralentissement de celui-ci. On ajoute aussi les ennemis ralentis dans un liste faite pour remettre leur vitesse originale lorsqu’ils sortent de la portée de la tour.
En plus du ralentissement, la Square Tower peut aussi étourdir les ennemis pendant une certaine durée, il fallait simplement rendre l’ennemi immobile pendant une durée donnée (réduire sa vitesse à 0 pendant x millisecondes).
On à aussi rajouté une méthode ‘draw_range’ liée à la méthode de la classe parent ‘toggle_range’ qui permet d’afficher la range de la tour, on n’a pas mit cette méthode dans la classe parent car chaque tour à une range différente (carré, ronde…).
Enfin une méthode ‘update’ qui permet à la tour d’actualiser ses actions à chaque tick, en particulier la méthode ‘slow_enemies_in_range’.

On est ensuite passés à la classe ‘RoundTower’ qui représente l’axe de défense de zone du système antivirus, pour ce faire nous avons également créé les attributs nécessaires à cette classe comme pour la Square Tower ensuite on à fait une méthode qui permet à la Round Tower d’attaquer tous les ennemis dans sa range, on lui à rajouté une animation d’attaque comme pour toutes les autres Towers. Cette Tower était la plus simple à faire car elle ne modifie pas le comportement des ennemis, elle se contente juste de faire les dégâts.

La ou nous avons eu un peu plus de mal, c’est lors de la création des towers Triangle et Trapèze, en effet celle-ci jettent des projectiles sur les ennemis, donc déjà il fallait faire une classe projectile, sauf que ces deux towers tirent des projectiles différents : l’une tire des projectiles à cible et dégâts uniques tandis que l’autre tire des projectiles explosifs, nous avons donc dû créer deux classes de projectiles pour les gérer, un autre problème est apparu quand les towers tiraient des projectiles mais comme les ennemis se déplacent, le projectile suivait une ligne droite et ratait, nous avons dû faire en sorte que les projectiles ne ratent jamais leur cibles, en faisant en sorte qu’ils suivent les ennemis, en utilisant des formules de vecteurs et de distance relative au rect(carré) du projectile en fonction de celui de l’ennemi, on a aussi dû faire en sorte que le projectile se détruise au contact d’un ennemi. Pour le projectile

explosif, au a aussi fait en sorte que au contact d'un ennemi, on calcule une range à partir du contact et tous les ennemis dedans prennent les dégâts d'explosion.
Il fallait aussi faire en sorte que si l'ennemi mourrait pendant qu'un projectile se dirigeait sur lui (le projectile ne l'a pas encore touché), il fallait que le projectile disparaisse car sinon le projectile restait sur place et ne touchait personne.

- Enemy



Comme dit dans l'introduction, les ennemis ont développé des nouvelles capacités, ils se sont organisés pour attaquer intelligemment et donc, nous avons également dû créer une classe parent qui regroupe les méthodes communes à chaque ennemi et ensuite créer des classes enfant qui hériteront de ce squelette.
Dans la classe parent, on a donc définit les attributs suivant:

- l'image de l'ennemi (qui est transformé pour faire un certain nombre de pixels)
- son rectangle qui s'adapte automatiquement à la taille de l'image
- sa position
- sa position temporaire
- le chemin qu'il doit suivre (sous forme de liste de tuples)
- son checkpoint actuel
- sa destination
- un booléen qui reporte si oui ou non il à traversé tout le chemin
- la liste des towers qui le slow
- son état d'étourdissement

Tous ces attributs sont donc communs à tous les ennemis, ensuite les méthodes:

- 'is-dead' qui renvoie si l'ennemi est mort
- 'update' qui gère l'actualisation de la direction, de l'étourdissement, du ralentissement, du checkpoint actuel, de la mort de l'ennemi..
- 'take_damage' qui inflige des dégâts à l'ennemi
- 'slow' qui ralentit l'ennemi
- 'reset_speed' qui remet sa vitesse à zéro

- ‘draw’ qui dessine l’ennemi ainsi que sa barre de vie
- ‘get_pos’ qui renvoie la position de l’ennemi

Grâce à ces méthodes et attributs, nous avons enfin commencé à définir les classes enfant comme la Croix, qui est un ennemi basique sans particularités, la Grande Croix, qui fait apparaître des Croix toutes les X secondes, le Losange qui devient de plus en plus rapide au fil du temps, le Rectangle qui possède un bouclier qui le protège ainsi que les autres ennemis, l’Hexagone qui est le boss, et qui possède toutes les caractéristiques de tous les ennemis, et enfin la flèche et le coeur qui respectivement augmente la vitesse de déplacement des ennemis à proximité et soigne les ennemis)à proximité.

Les difficultés ont bien sur été rencontrées lors de l'implémentation des ennemis avec des capacités comme la grande croix (le problème est qu’elle faisait apparaître des ennemis, mais que la position de départ des ennemis est au début du chemin alors qu’on voulait qu’elle les fasse apparaître à côté de son rect. Le rectangle à demandé beaucoup de réflexion pour implémenter son bouclier, car en effet il faut que celui ci intercepte les projectiles qui sont potentiellement dirigés vers d’autres ennemis mais tout en étant dans la range du bouclier, la flèche et le coeur modifient le comportement des ennemis donc nous avons aussi rencontrés des problèmes tels que l'augmentation de la vitesse infinie, le soin qui va au delà des points de vie max d’un ennemi, le conflit entre la Square Tower et la flèche à rapporté également beaucoup de bugs. Enfin les deux ennemis simples à implémenter étaient la Croix et le Losange car le premier n’a pas de capacité et le deuxième à juste sa vitesse qui augmente au fil du temps.

- <u>Button</u>

Nous avons simplement créé un classe Button qui permet rapidement de créer un bouton à l’écran car notre jeu en possède quelques-uns, donc afin d’éviter les répétitions de code pour créer les boutons, il fallait à présent juste instancier un objet de type Button.
Les attributs sont basiques, et lorsqu’on doit instancier un Bouton, il faut mettre 6 paramètres, sa position x et y, sa longueur, sa largeur, le texte qui sera présent sur le bouton, et enfin sa couleur. Enfin, avec quelques méthodes comme ‘draw’ pour le dessiner à l’écran, ‘is_clicked’ qui vérifie si l’utilisateur clique dessus, et enfin ‘verify_click’ qui permet au bouton de rester activé, la classe Button est déjà finie et prête à être utilisée.

- <u>Base</u>

La Base représente l’endroit ou une tour doit être placée, donc nous avons décidé de créer une classe pour les mêmes raisons que nous avons créé la classe Button, car notre jeu possède énormément de Bases, mais aussi car les classes sont un moyen de rendre le code beaucoup plus lisible et optimisé, car pygame demande beaucoup de ressources.

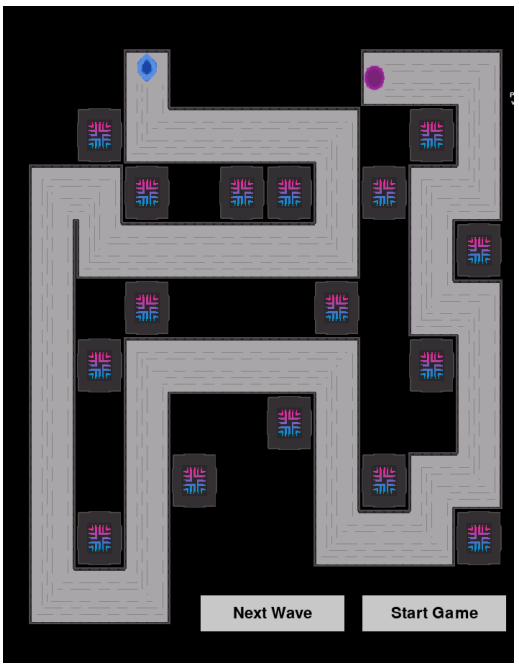
Avec des méthodes simples comme ‘add_tower’, ‘remove_tower’ lié avec les boutons de sélection des towers, la classe est rapidement créée. On utilise “generate_bases()”, qui à partir d’une matrice symbolisant la carte crée la liste des objets Base.

- La Map :

Pour enregistrer la map, on a opté pour l’utilisation d’un fichier txt nommé map.txt. Une fois lu on le convertit en matrice à l’aide de la fonction “generate_map” qui itère à travers le fichier avec la fonction ”next” , chaque liste de cette matrice correspond à une ligne de la carte dont pour chaque tuile est donné un encodage particulier. Afin d’afficher la carte, à chaque répétition de la boucle de jeu puisque on doit supprimer tout ce qui est affiché afin d’enlever des éléments, on appelle la fonction ”print_map” qui parcourt la matrice et en utilisant un dictionnaire “tiles” qui dont les clés sont les encodages et les valeurs les images affiche la carte tuile par tuile.

Comme on ne se déplace pas sur la carte, Il s’est alors posé le problème que la carte n’était pas forcément visible dans la fenêtre de jeu. On a alors le choix entre afficher la carte entièrement à l'écran, créer un système de déplacement, ou choisir arbitrairement une résolution et la faire fonctionner avec notre script. Ainsi notre première idée a été de faire apparaître la carte en entier, on calcule alors la taille que doivent faire chaque tuile afin de remplir tout l'écran, on divise les dimensions de l'écran par le nombre maximum de tuiles en colonne et en ligne retourné par la fonction” max_line”. Il fallait aussi actualiser les coordonnées de chaque objet. On avait alors réussi à avoir une carte qui remplissait même les dimensions d’écran les plus absurdes. Mais on a eu un problème lorsqu' on a voulu rajouter les ennemies c’est derniers devant aller à des couples de coordonnées précises ne collait pas avec des coordonnées et des tailles toujours changeante. Il aurait été possible de trouver une solution pour pallier au problème inopiné mais
on a décidé qu’afin de garder une cohérence avec le chapitre 2, de fixer la taille d’écran à du 1920*1080 cela avait aussi un intérêt esthétique car la taille des tuiles modifié dégradait leur qualité. Alors on affiche la carte en 800*800 au milieu de la carte sachant qu' à tout moment on peut en changer les dimensions.

Exemple de Map :



Le main :

Avec toutes ces classes, il fallait enfin commencer par les assembler entre elles et réaliser la gestion des événements:

Premièrement, il faut importer tous les scripts de classes et autres scripts complémentaires au main pour ça il faut simplement écrire ‘from script import*’

Ensuite vient le définition des variables, constantes et fonctions utiles au bon fonctionnement du jeu mais aussi à une bonne expérience comme la fonction ‘draw_win’ qui affiche l’écran de victoire, la fonction ‘draw_player_money’ qui affiche l’argent actuel du joueur…

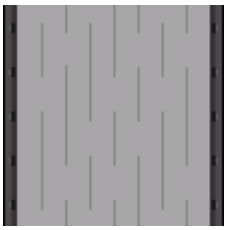
Pour finir, la gestion des événement, l’appel des classes et des fonctions du main se font dans une boucle ‘While’ par exemple la gestion des boutons, la gestion des bases, la gestion du comportement des ennemis, des towers…

Création de la Map :

Les routes

Pour réaliser la carte du jeu, ne sachant pas dans un premier lieu comment elle sera constituée, les assets pour chaque direction ont été réalisés. Par exemple voici différents assets utilisés pour les routes (en respectant l’échelle réelle) :

La route “*East-West*” pour faire un chemin d' Est en Ouest:

La route “*North-South*” pour faire un chemin du Nord au Sud: 

La route angulée “*East-South*” pour faire un chemin de l’Est vers le Sud:

La route angulée “*North-West*” pour faire un chemin du Nord vers l’Ouest:

Difficultés rencontrés :

Pour éviter une redondance entre les différents “assets”, nous avons jugé intéressant de ajouter des traits sur la route, avec un paterne en diagonal. Le résultat a été celui voulu, cependant le trajet pour un parvenir a été semé d'embûches :

- Il a fallu faire en sorte que entre chaque routes les lignes se rejoignent dans le cas où elles ne sont pas terminées (le but étant de faire Grande/Petite/Grande/Petite etc)
- Il a fallu comprendre comment réaliser les angles sans avoir des problèmes liés aux assets reliés

Après 3 séances en présentielle, les 12 assets pour chaque angle et chaque route toute droite ont été réalisées sans erreur.

Les socles

Pour permettre au joueur de savoir les endroits autorisés afin de placer les différentes tours, un asset réalisant un socle a dû être créé. Pour son design, afin de respecter l'idée générale du jeu, un logo représentant un micro-processeur se trouve en son centre, de différentes couleurs, avec une base carrée de 100x100 pixels (le but étant qu'il soit caché lorsque la tour “square” soit posée, mais visible pour les autres).

Remarque

Pour tous les assets dont nous avons parlé dans cette partie, chaque bordure est composée de la couleur noir. le but étant de compléter le reste des cases inutiles de la carte par simplement un carré noir. Ansi chaque asset se mélange parfaitement avec le fond noir qui est souhaité être représenté.
En regardant au pixel près, chaque asset des routes possède une bordure noire de 1 pixel pour chaque mur sur un côté.

Chapitre 2 - Pierre (code) + Léo (map)

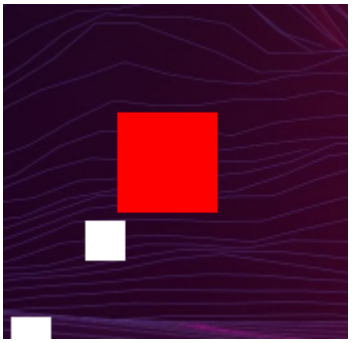
Le chapitre 2, qui consiste en l'attaque du serveur diffusant les virus, est un jeu de plateforme se jouant avec les touches du clavier. L'objectif est d'éviter les tirs et de parcourir le chapitre afin d'arriver au checkpoint final, le serveur.
Pour le chapitre 2, nous avons d'abord commencé par développer les obstacles. Cette tâche, relativement simple, consistait simplement à avoir une liste de rectangles adaptés à Pygame, puis à les afficher un par un à l'écran.

Map

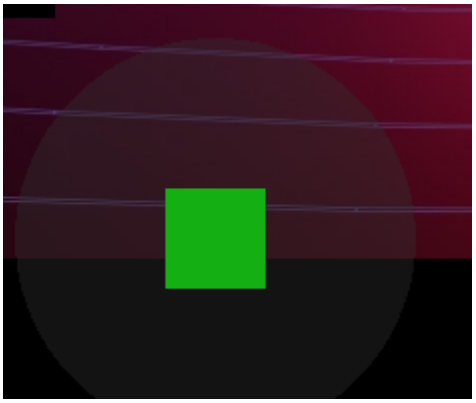
Pour la map, elle est constituée de plusieurs objets qui ont été créés avec des classes et qui constituent le corps du niveau :

- Les obstacles : ce sont de simples rectangles noirs créés avec la commande pygame.Rect ainsi que les coordonnées que l'on souhaite donner à l'objet, le joueur devra les utiliser à bon escient afin de parcourir le niveau. Ils permettent au joueur une avancée moins complexe tout de même.

- Les towers : elles sont les pièges à distances les plus difficiles à se défaire, elles sont placées difficilement et ont été initialisées avec une classe, voir la section “Tours ennemis”.



- Les traps sont des objets qui permettent à l’aide d’une zone de portée, de ralentir le joueur dans sa course et ainsi l’obliger à attendre les bon timings pour esquiver les différents pièges mis en place par le niveau. Ses dégâts sont variables et sa vitesse de tir aussi en fonction de la tour, ce qui constitue un excellent élément pour la structure, le gameplay et la difficulté.



- Les spikes sont des triangles aussi provenant d’une classe mais ayant un but spécifique : éviter au joueur l'accès à des zones auquel il ne devrait pas pouvoir accéder et ainsi le forcer à trouver un nouveau chemin. Ceux-ci infligent des dégâts immédiats et One Shot le joueur, ce qui signifie qu’il perd instantanément le niveau au contact de celui-ci, les hitboxes restent très difficiles à coder à cause du launcher (fenêtre pygame).

Bien évidemment, pour créer le jeu, il a fallu passer un temps monstre sur l'adaptation des coordonnées et la gestion des sauts, des hitboxes et du gameplay. Le plus long à été d'effectuer un test minimum pour absolument chaque modification. Il faut accorder tout ce qui a été créé pour faire une bonne harmonie singulière tout en cherchant l'inspiration pour pas que le joueur sente la répétition des parties du niveau avec des mécaniques bien distinctes qui demandent au joueur de se la donner.

Mouvement

Les premières difficultés sont apparues avec le joueur. Les mouvements sont relativement simples, à savoir Gauche/Droite/Saut, c'est-à-dire Q/D/Espace ou Flèche Gauche/Flèche droite/Espace, selon les préférences. De même pour la gravité : il a suffi de ressortir les cours de spécialité physique-chimie de terminale, d'avoir un vecteur vitesse verticale (stockée comme une variable entière) et de lui ajouter la valeur de la gravité à chaque unité de temps. Nous avions au départ pris les valeurs terrestres (environ 10m/s² pour la gravité), mais cela s'est vite révélé peu fun et trop lourd. Nous avons également modifié la hauteur des sauts, en sacrifiant certes du réalisme, mais il faut reconnaître qu'un humain ne saute pas assez haut pour être bon dans un jeu de plateforme. Il a aussi fallu limiter la vitesse verticale, pour simuler les frottements de l'air. Bien sûr, cette vitesse maximale a été écrite dans le code, et il est possible de modifier la valeur pour changer l'impression de poids du joueur (plus il tombe vite, plus il est ressenti comme lourd).

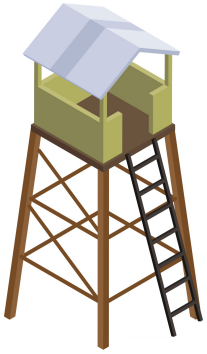
Illustration de l'air interagissant avec un objet. C'est cet effet qui, simulé, limite la vitesse de chute du joueur

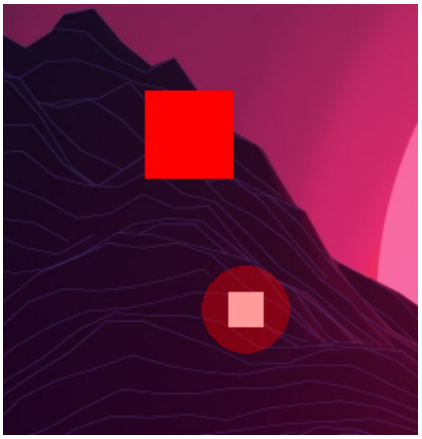
Lors des collisions avec les obstacles, nous avons rencontré des problèmes flagrants, comme le fait que le joueur traverse les obstacles, ce qui correspond mal avec la définition d'un obstacle. Ce problème apparaissait à la fois en déplacement horizontal et en déplacement vertical, mais le choix a été fait de traiter différemment ces deux cas.
Lors d'un déplacement horizontal, le joueur a une vitesse constante, à savoir celle définie en dur dans le code. En revanche, les chutes pouvant augmenter en vitesse de manière

significative, le problème a été plus complexe. Lors d'une chute, une fonction sur Obstacle s'exécute à chaque pixel de la chute pour savoir si le joueur est posé sur un obstacle ou non. Si oui, la chute s'arrête là, mais si non, la chute continue. Pour l'anecdote, lors des tests, avant que les collisions ne soient bien implémentées, le joueur passait fréquemment à travers les obstacles et se retrouvait donc sous la carte. Pour pallier ce problème, il a fallu ajouter une téléportation qui envoyait le joueur au-dessus de la carte s'il était en dessous. Une simple fonctionnalité de debug, s'est pourtant retrouvée dans le jeu final en devenant une mécanique à part entière du jeu.
Afin de permettre à deux personnes de travailler sur le code en même temps, le choix a été fait de déplacer les listes d'obstacles, de tours et de checkpoints dans un fichier séparé, afin d'avoir la map contenue dans des fichiers séparés du code en dur.

Tours ennemies

Une fois les déplacements et les obstacles effectués, il a fallu créer les tours ennemies. Pour cela, il suffisait pour chaque tour de regarder si le joueur était à sa portée, et si oui de tirer un projectile dans sa direction, via un calcul d'angle permettant de déduire sa vitesse horizontale et sa vitesse verticale. Chaque tour possède sa liste de projectiles, ainsi il est possible dans le code de déterminer quel projectile a été tiré par quelle tour, offrant la possibilité d'ajouter ultérieurement des fonctionnalités utilisant ce principe. Pour chaque objet Projectile, à chaque actualisation, il se déplace, se supprime éventuellement s'il est entré en collision avec un obstacle, et met éventuellement des dégâts s'il touche le joueur. Afin d'accorder au joueur un temps de répit après s'être pris un projectile, on lui donne un temps de pause, inscrit dans les attributs de l'objet Player. Pendant cette pause, tous les projectiles sont supprimés, et les tours sont temporairement désactivées. Une fois la pause terminée, le jeu reprend son cours et les tours se remettent à tirer des projectiles. A noter que les Traps (ralentisseurs) et les Spikes (pics qui tuent instantanément) ne sont pas concernés par cette période de pause.





Dégâts et mort

Le joueur peut mourir de quatre façons différentes : avec un Spike mortel au contact, ou par l'une des trois tours. La tour "classique" tire des projectiles à une vitesse normale, qui ont une vitesse normale. La tour "sniper" tire peu de projectiles, mais à une vitesse élevée, et la tour "dégâts de zone" tire des bombes à un rythme normal mais à une vitesse réduite. Pour cette dernière tour, seul le carré du projectile inflige des dégâts, sauf quand le projectile explose en rencontrant un obstacle, auquel cas le joueur prendra des dégâts s'il se situe dans le cercle autour du projectile. Concernant les dégâts, le Spike enlève toutes ses vies au joueur en cas de contact, les tours "classique" et "sniper" font perdre une vie au joueur, tandis que la tour "dégâts de zone" lui en enlève deux, pour compenser la lenteur des projectiles.
Les vies ont été relativement simples à développer : le joueur arrive avec cinq vies, et quand il n'en a plus, il meurt. L'affichage des vies se fait en bas à droite de l'écran, avec une barre rouge statique, recouverte d'une barre verte dont la taille dépend du nombre de vies du joueur.
Pour les checkpoints, la difficulté a été de sauvegarder le nombre de pièces et de vies du joueur au dernier moment où il a passé le checkpoint, afin d'éviter le cas où un joueur aurait 1 pièce, validerait un checkpoint, récupérerait une seconde pièce, mourrait, reviendrait au checkpoint et aurait quand même validé cette seconde pièce. Sur la version finale du jeu, le joueur peut revenir au dernier checkpoint, même s'il l'a déjà validé, afin de sauvegarder les pièces récupérées depuis la dernière fois où il est allé sur ledit checkpoint.

A tout instant, que le joueur soit mort ou non, il peut appuyer sur la touche Entrée pour revenir au dernier checkpoint validé, avec donc les pièces qu'il avait validées au dernier moment où il est allé dessus. S'il se retrouve à 0 vie, un texte lui affiche sa progression dans le chapitre (checkpoints validés / checkpoints totaux (en incluant celui marquant la fin du chapitre)), ainsi que ses possibilités (Echap pour retourner à l'accueil, Entrée pour retourner au dernier checkpoint). Si le joueur n'a validé aucun checkpoint, la touche Entrée lui fera recommencer le chapitre.
Lorsque le joueur a terminé le chapitre, c'est-à-dire qu'il a atteint le checkpoint final, le jeu s'arrête et lui indique le nombre de pièces qu'il a récoltées, avec une couleur différente selon le nombre de pièces. Le texte va de rouge, si le joueur n'a récupéré aucune pièce, à vert, s'il les a toutes récupérées, avec des mélanges entre les deux couleurs s'il a récupéré ni 0 ni 100% des pièces.

Lien avec l'accueil

Pendant le développement du chapitre, il était indépendant de l'accueil : il fallait directement exécuter Chapitre_2/Code/main.py pour commencer le chapitre. Afin de finaliser le jeu, il a fallu connecter l'accueil au Chapitre 2. Pour ce faire, le Chapitre 2 se lance avec une fonction start(), et le code est directement importé depuis le répertoire du Chapitre. Cependant, comme tout le code est exécuté depuis l'accueil, il faut parcourir l'arborescence dans les import, ce qui rend le code un peu plus fastidieux à lire, mais permet de lancer le Chapitre 2 directement depuis le bouton de l'accueil, ce qui améliore l'ergonomie.
La fonction leave() a également été mise dans un fichier séparé nommé quit.py afin de pouvoir être appelée séparément du reste du code. Ainsi, il a simplement fallu modifier cette fonction dans ce fichier séparé pour faire en sorte que quand le joueur quitte le niveau (ALT+F4 / Echap / Clic sur la croix en haut à gauche), on passe d'une fenêtre qui se ferme à un accueil qui se lance à la place.

```
import pygame
import sys

sys.path.insert(0, '..')
import Chapitre_1.Code.main as cp1
import Chapitre_2.Code.main as cp2
```

Exécution

A la fin du projet le choix a été fait de créer un fichier exécutable main.exe situé dans le répertoire d'accueil. Ce fichier, généré à partir de PyInstaller et nécessitant le dossier build, permet de lancer le jeu sans nécessiter Python. Cette fonctionnalité étant expérimentale, il faudra lancer le code avec Python en cas de problèmes.

Performances

Ce jeu étant réalisé en Python, un langage de haut niveau, les performances sont moins bonnes que s'il avait été réalisé dans un langage de plus bas niveau, comme du C. Nous avons néanmoins pu programmer plus vite et ainsi avoir un meilleur rendu que si nous avions passé un temps égal sur du C. Les performances sont malgré tout correctes, surtout sur des machines récentes, sans nécessairement être haut de gamme. Les lenteurs constatées sont lors du démarrage du jeu, pour charger toutes les ressources, lors du retour à l'accueil, notamment pour charger la musique, et lors du spawn des ennemis dans le chapitre 1, en raison du chargement de la texture des ennemis.

Problèmes d'exécution du code

L'exécutable ne fonctionne pas : exécuter le fichier main.py à la place du fichier exécutable.

Erreur Python - ressources non trouvées : il faut se trouver dans le répertoire Accueil et lancer le fichier main.py pour que les ressources soient chargées.

Problème d'affichage au chapitre 2 : pour des raisons techniques, ce chapitre fonctionne correctement lorsque l'affichage fait 1920x1080 pixels (FHD).

Double retour : La fenêtre pygame se ferme automatiquement lorsqu'on lance le jeu, puis on joue au chapitre 2, puis on quitte pour rejouer et enfin re-quitter. C'est dû à un problème dans l'accueil qui a une fonction retour mal codée.

Problème sonore : Le volume ne conserve pas toujours les réglages mis au départ.