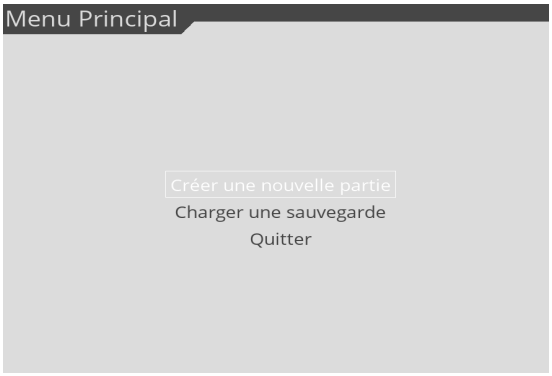
Introduction

Ce rapport détaille le développement d'une application informatique pour jouer au jeu d'Hex, un jeu de stratégie. L'objectif principal du projet est de créer une interface permettant aux utilisateurs de jouer à ce jeu. L'application offre plusieurs fonctionnalités telles que la création de nouvelles parties, la possibilité de jouer contre un adversaire humain ou une intelligence artificielle.

Présentation Globale



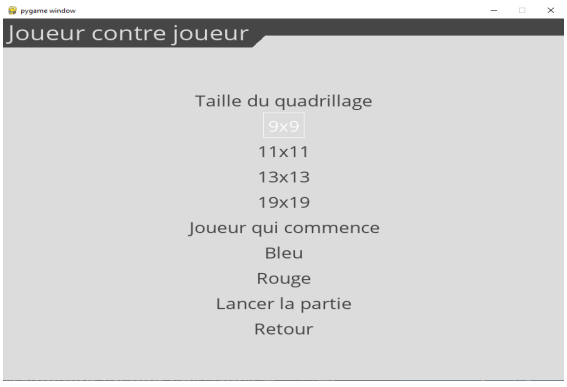
Menu Principal
L'application démarre avec un menu principal proposant trois options :

- Créer une Nouvelle Partie
- Charger une Sauvegarde
- Quitter

Créer une Nouvelle Partie
L'utilisateur est dirigé vers un sous-menu offrant trois options :

- Joueur contre Joueur
- Joueur contre IA
- Retour

Avant de lancer la partie, on peut choisir la taille du quadrillage et quel joueur commence.

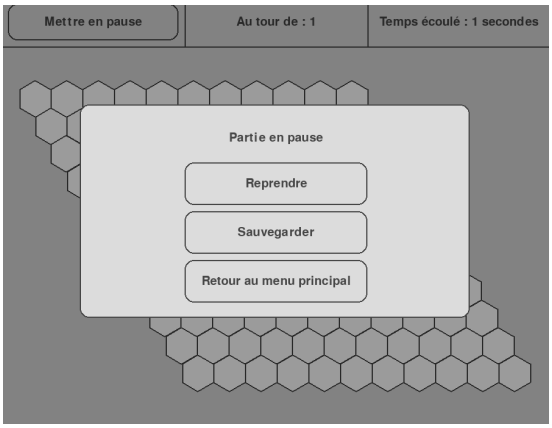


Charger une Sauvegarde : L'utilisateur peut accéder à une sauvegarde précédemment enregistrée. Cette sauvegarde permet de reprendre la partie.

Quitter
Quitter l'application.

Mettre en pause
Le joueur dispose du bouton "Mettre en pause" une fois la partie lancée, offrant trois options :

- Reprendre
- Sauvegarder
- Retour au menu principal



Présentation Approfondie

La Logique et l'IA

La partie logique dans le fichier `logic.py` a été faite en deux phases que nous allons présenter.

1) Joueur contre joueur

La première phase a eu pour but de pouvoir jouer en joueur contre joueur sans IA.
Pour cela nous avons modélisé le plateau puis nous avons créé le système de pose de pions et la détection d'un gagnant.

Pour cela nous avons créé une classe `logic` ayant comme attributs:

- la taille du plateau (`self.size`)
- le plateau sous forme de matrice carrée (`self.board`)
- un dictionnaire contenant les ensembles de pions connectés de chaque joueur sous forme de sets (`self.connected_pieces`)
- un dictionnaire contenant les coordonnées des bordures à atteindre pour chaque joueur, cela nous servira pour détecter un gagnant (`self.borders`)
- le nombre de tours joués (`self.cpt_tours`)

Le plateau est représenté sous forme d'une matrice carrée (liste de listes) de la taille du plateau souhaité.

Chaque élément est soit:

- un 0 (case libre, aucun pion n'est posé)
- un 1 (case contenant un pion du joueur 1)
- un 2 (case contenant un pion du joueur 2)

Pour faciliter et réduire la complexité de la détection d'un gagnant nous avons créé plusieurs méthodes.
La fonction `update_connected_pieces` est utilisée lors de la pose d'un pion pour mettre à jour le dictionnaire de sets de pions connectés.
Lorsqu'un joueur pose un pion à côté d'un autre de ses pions, ce nouveau pion est ajouté au set déjà existant. Si le pion posé n'est voisin d'aucun autre pion du même joueur, alors on crée un set contenant seulement le nouveau pion.

Pour détecter si le pion est voisin d'un autre pion, on utilise la méthode `get_neighbors` qui prend en paramètres les coordonnées du pion et qui renvoie tous les pions qui lui sont voisins.

Ainsi pour détecter s'il y a un gagnant, il suffit de parcourir les sets de pions connectés de chaque joueur et on vérifie si au moins un des sets du joueur contient un pion dans chacune des extrémités du plateau (haut et bas pour le joueur 1, gauche et droite pour le joueur 2).
C'est ce qui est fait dans la méthode `check_winner`.

2) Codage de l'IA

Pour créer une IA, nous avons choisi d'utiliser l'algorithme minimax dans lequel nous avons inclus l'élagage alpha-beta pour tenter de réduire le temps d'exécution.

Le joueur maximisant qui sera joué par l'IA lors des parties est le joueur 2.
Les futures situations du jeu sont créées en faisant une copie de l'objet de la classe `logic` en utilisant `deepcopy`.
La condition d'arrêt de la recherche de l'algorithme est atteinte soit quand la profondeur de recherche souhaitée est atteinte ou soit quand on détecte un gagnant.

L'évaluation du plateau est obtenue en soustrayant l'évaluation du joueur 1 à l'évaluation du joueur 2.

Evaluation des joueurs :

Après de nombreux tests, nous avons décidé d'évaluer le joueur 1 différemment du joueur 2 puisque cela rendait l'IA plus compétente.

Le joueur 1 est évalué sur 2 aspects:

- La longueur (horizontale) des chaînes de ses pions
- Ajout d'un bonus si les chaînes sont reliés à un des bords

Le joueur 2 est évalué de la même façon que le joueur 1 mais avec d'autres aspects en plus:

- Un bonus est ajouté pour inciter l'IA à poser un pion d'une certaine manière qui favorise le blocage du joueur opposé.
- Un bonus est ajouté si le pion posé est voisin de exactement 1 pion du joueur opposé

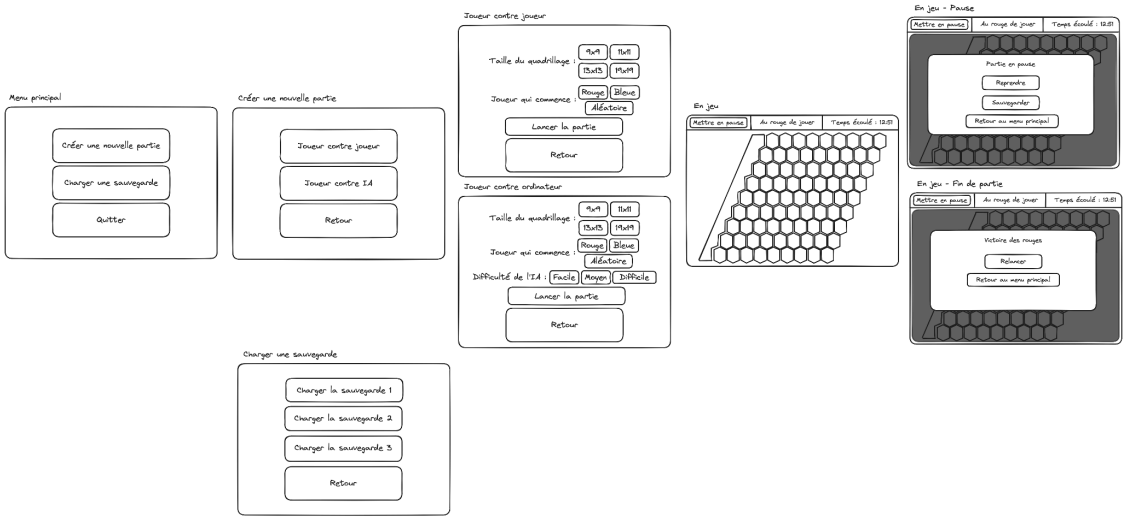
Difficultés rencontrées:

Malheureusement nous n'avons pas pu compter sur la puissance de calcul pour avoir une IA très compétente puisque le nombre de cases du plateau est élevé et donc les possibilités de déroulement de la partie sont élevées aussi, ce qui rend l'exécution très longue lorsqu'on choisit une grande profondeur de recherche.
Nous avons essayé d'appliquer des filtres pour réduire le nombre de possibilités mais cela demandait aussi de la puissance de calcul.
Ainsi pour jouer en mode 'facile' la profondeur de recherche sera de 2, pour le mode 'moyen' elle sera de 3 et elle sera de 4 pour le mode 'difficile'.

La deuxième difficulté rencontrée concerne l'évaluation du plateau car il n'y a pas de façon évidente d'évaluer les situations du jeu. Dès le départ nous avons dû chercher et tester plusieurs méthodes d'évaluations simples dont aucune ne donnait une IA assez forte. Nous avons donc tenté d'améliorer l'IA avec des méthodes d'évaluation plus complexes mais même après plusieurs tentatives d'amélioration, l'IA fait parfois de mauvais choix. Malgré tout, la fonction convient pour jouer à un mode de difficulté facile ou intermédiaire.

L'Interface Utilisateur

Pour *designer* l'interface utilisateur, nous avons utilisé trois outils/notions ; Excalidraw, Pygame (et Pygame Menu) et la programmation orientée objet. Nous avons commencé par « dessiner » notre interface avec Excalidraw, pour nous permettre d'y voir plus clair sur les couleurs, les formes, les widgets et le nombre de menus que l'on trouvait adéquat.

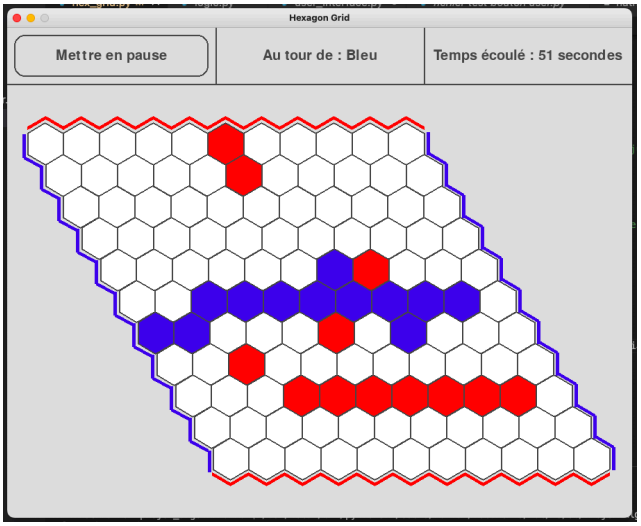


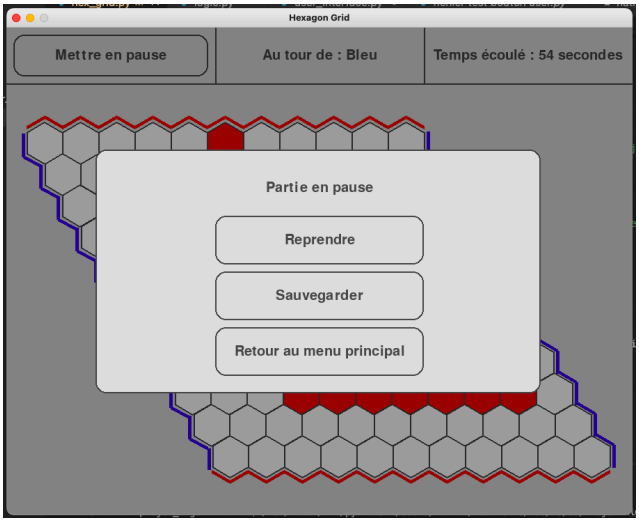
Ceci nous a donné une base commune pour commencer à programmer l'interface. Thushanth et Ania ont programmé les hexagones et la grille et Nathan les menus. La structure logique qu'on a tout de suite envisagée était une classe pour les objets hexagones de la grille et une classe pour la grille, qui appellerait les objets hexagones pour « peupler » l'espace de jeu.

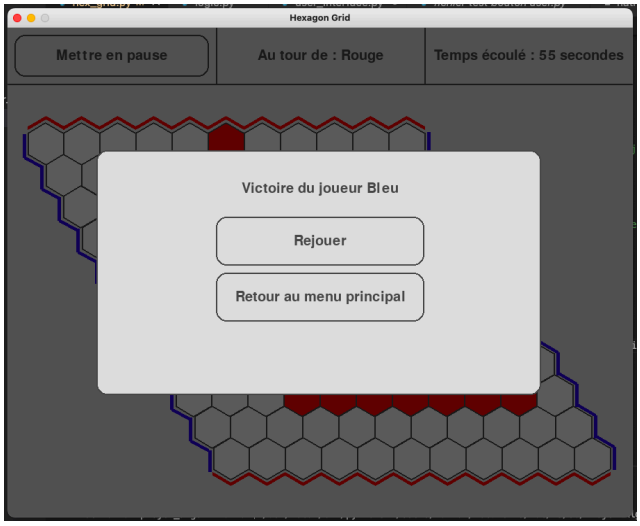
Pour notre classe hexagone (`Hexagon()`) nous avons notamment défini les attributs `x` et `y` les coordonnées de chaque hexagone dans le canevas (utilisé pour dessiner les hexagones et les bordures), `numero` (utilisé notamment pour communiquer avec le tableau logique), `size` (pour régler la taille des hexagones, en fonction de la taille de la grille), `color` (en blanc, bleu ou rouge) et `grid_size` la taille de la grille (qui permet de régler automatiquemet la taille des hexagones ; par exemple si on est sur une grille 19x19, les hexagones sont plus petits que sur une 11x11). Nous avons également les méthodes `draw()` qui dessine les hexagones et les bordures, et `contains_point()` qui crée un rectangle de collision sur l'hexagone qui appelle cette méthode, et qui retourne si l'utilisateur a cliqué dessus ou non.

Pour la classe de la grille (`HexagonGrid()`), nous avons les attributs usuels `width` et `height` qui définissent la largeur et la hauteur du canevas, que nous avons fixé à 900 par 700, l'attribut `main_menu` qui permet d'appeler le menu Pygame Menu du menu principal, un attribut `grid_size` qui permet de gérer le nombre d'hexagones créés et la taille de la matrice logique, `hexagons` qui est une liste pour stocker les objets hexagones créés, `is_paused` qui est un booléen qui gère si le jeu est en pause ou non (car si il est en pause par exemple, on veut qu'on ne puisse pas cliquer sur des hexagones), `turn` qui est le témoin du joueur qui doit jouer à chaque tour (et qui peut être paramétré avant le début d'une partie, pour choisir quel joueur joue le premier), `first_turn` qui a été rajouté plus tard et qui copie le premier état de `turn` (ça permet lorsque l'on rejoue de faire commencer le joueur qui avait commencé la première partie, sinon parfois lors d'une nouvelle partie le paramètre de premier joueur n'était pas respecté), `victory` qui est un booléen et gère l'état de victoire (car si victoire il y a, on veut afficher un menu par exemple), et un attribut `logic` qui est une instance de la classe `Logic`, et qui permet de communiquer avec la logique pour vérifier si il y a une condition de victoire, ou pour jouer avec l'IA. Sur cette classe on a également les méthodes `setup_hexagons()` qui en fonction de la taille de la grille va avec deux boucles créer tous les objets hexagones avec les paramètres appropriés, `calculate_offset()` qui est utilisée dans `setup_hexagons()` et qui permet de calculer le décalage entre chaque hexagone (et qui s'adapte aussi du coup en fonction de la taille de la grille), `draw_pause_menu()` qui affiche le menu de pause si on appuie sur le bouton de pause et qui permet de sauvegarder ou de revenir au menu principal (sans utiliser un menu Pygame Menu, mais un canevas !), `draw_victory_menu()` qui utilise la même structure que `draw_pause_menu()` mais en cas de victoire (affiche un menu qui propose de rejouer ou de revenir au menu principal), et la fonction `handle_click()` qui est appelée dans la boucle principale de *run* et qui permet de gérer les cas de clique sur hexagone (pour les cliques sur le bouton de pause, nous l'avons géré directement dans la boucle principale de *run*). Enfin, nous avons la fonction `run()` dans laquelle nous assemblons tous les éléments définis précédemment pour créer la fenêtre de jeu, c'est ici également que l'on ajoute des lignes, le bouton pause, un label indicateur de tour et un label indicateur de temps écoulé, et où l'on gère tous les évènements de clique (ou alors on appelle les fonctions les gérant comme pour les cliques sur hexagones avec `handle_click()`), les conditions de victoire et de pause, etc.

C'est donc dans ce fichier que nous gérons les hexagones, la grille et l'intégration de la logique (détaillée dans la partie sur l'IA).







Pour ce qui est des menus, nous avons plutôt utilisé Pygame Menu qui nous a permis de rapidement et facilement établir une arborescence de menus. Pygame Menu et plus simple à utiliser que de créer des canevas avec Pygame, mais permet moins facilement de dévier de sa structure et de faire ce que l'on veut, donc on pense que si c'était à refaire, on aurait peut-être choisi de faire ces menus avec les canevas de Pygame plutôt.

Ici nous avons une seule classe `GameMenu`, qui possède les attributs usuels `surface`, `largeur` et `longueur` qui permettent de définir la taille de la fenêtre Pygame, `main_menu` qui est donc le menu principal et le premier menu hiérarchiquement parlant, `game_mode` qui est une *string* qui conserve le type de partie (`pvp` pour du joueur contre joueur ou `pve` pour du joueur contre IA), `selected_starting_player` qui stocke le joueur qui commencera choisi dans

le menu, et `selected_grid_size` qui stocke la taille choisie (ces deux arguments seront passés à l'instance `HexagonGrid` au lancement d'une partie).

Pour ce qui est des méthodes de classe, nous avons `set_main_menu()` qui permet simplement de créer le menu principal (avec ses labels et boutons correspondant), `mainloop()` qui permet de lancer la boucle principale en tant qu'attribut d'objet (vu qu'on a tout programmé en POO), `create_new_game()`, `load_saved_game()`, `pvp()` et `pve()` qui sont tous des menus pour créer ou charger une partie, pour créer une partie joueur contre joueur ou une partie joueur contre IA (avec pour chaque menu encore une fois des labels et des boutons correspondant). On a aussi défini les petites méthodes `set_grid_size()` et `set_starting_player()` qui sont appelées lorsque l'on appuie sur les boutons pour définir la taille de la grille ou le joueur qui commence et qui mettent à jour les attributs correspondant (pour indiquer à l'objet `HexagonGrid` quels paramètres utiliser). Enfin, nous avons la méthode `start_game_2()` (la première version a fini par être supprimée, nous ne l'oublierons jamais 😢) qui crée un objet `HexagonGrid` avec les paramètres choisis par le joueur, et qui est appelée lorsque que l'on clique sur le bouton `Lancer la partie` des menus de jeu.