QUORIDOR

PROJET 1-PROJ

Documentation technique

Auteurs : Tommy BRISSET, Orlann FERREIRA, Yann GELLNER et Thibault CHOUBRY

Campus : Tours

Année scolaire : 2022-2023

Date : 18-06-2023

SOMMAIRE

Justification du choix du langage et de la librairie graphique

Dans le cadre de notre projet, nous avons pris la décision de développer en utilisant le langage de programmation Python et la bibliothèque graphique Pygame. Ce choix s'est basé sur plusieurs considérations qui ont guidé notre décision.

Tout d'abord, parmi les langages de programmation disponibles, Python était celui que notre équipe maîtrisait le mieux. En raison de notre expérience préalable avec Python, notamment via le module réalisé durant notre année, nous avons estimé qu'il serait plus efficace et productif d'utiliser ce langage pour développer notre application. La syntaxe claire et concise de Python ainsi que sa large communauté de développeurs ont également été des facteurs déterminants dans notre choix.

En ce qui concerne la bibliothèque graphique, Pygame s'est révélé être la solution la plus intuitive et pratique pour notre projet. Sa simplicité d'utilisation nous a permis de créer rapidement une interface graphique interactive et attrayante. De plus, les nombreuses fonctionnalités pré-implémentées offertes par Pygame nous ont donné un avantage considérable dans la réalisation de notre projet. La gestion des éléments affichés à l'écran, tels que les sprites, les animations et les interactions utilisateur, a été grandement simplifiée grâce à la richesse de la bibliothèque.

De plus, l'un des avantages majeurs de choisir Pygame était la possibilité d'explorer une nouvelle librairie graphique. Cela nous a permis de découvrir une autre approche de la programmation graphique et d'élargir nos compétences en matière de développement d'interfaces utilisateur interactives. La flexibilité de Pygame nous a donné l'opportunité d'expérimenter et d'innover, tout en offrant des performances solides pour notre application.

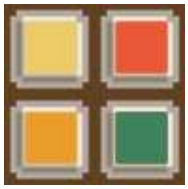
Il convient de noter que l'utilisation de Pygame représentait également un défi pour notre équipe, car nous avions une expérience limitée avec cette bibliothèque graphique. Cependant, nous avons vu cela comme une opportunité d'apprentissage et de croissance. Grâce à des ressources en ligne, des tutoriels et la documentation complète de Pygame, nous avons pu surmonter les obstacles et développer avec succès notre projet.

En résumé, notre choix de développer en Python avec la bibliothèque graphique Pygame était fondé sur notre familiarité avec le langage, la simplicité et la richesse des fonctionnalités de Pygame, ainsi que l'opportunité d'explorer de nouvelles possibilités de développement graphique. Malgré les défis auxquels nous avons été confrontés, nous sommes fiers d'avoir choisi cette combinaison et d'avoir réussi à créer une application à la fois fonctionnelle et esthétiquement agréable.

Organisation et contenu des fichiers

Dans notre rendu final, le dossier contenant tous les fichiers et autres documents est rangé selon les modalités suivantes :

- Un dossier **Assets :** contenant toutes les images et tous les éléments graphiques de chaque page du jeu. Ce dossier contient toutes les ressources graphiques utilisées dans le jeu. Les images peuvent être des éléments d'arrière-plan, des boutons, des icônes, des éléments de l'interface utilisateur ou tout autre composant visuel. Les sous-dossiers sont organisés par scènes pour faciliter la gestion des ressources.



Logo du jeu

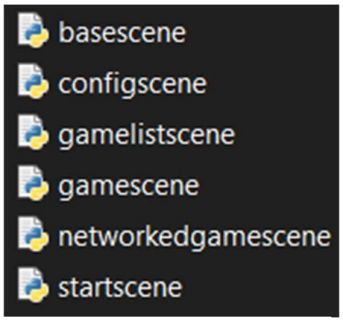
Bouton start de la première page du jeu

- Un dossier **Configs :** contenant les fichiers de configuration de l'interface utilisateur du jeu. Ces fichiers, au format JSON, définissent les attributs de différents éléments d'interface utilisateur, comme leur position, leur taille, leur image de fond, etc. Cela permet une certaine flexibilité dans la conception de l'interface utilisateur, car les changements peuvent être apportés en modifiant simplement ces fichiers de configuration.
- Un dossier **Game** qui contient les fichiers définissants la logique du jeu et les fonctions permettant de jouer à notre jeu.
- Un dossier **Libs** qui contient des fichiers contenant les différentes librairies externes utilisées tels que stockings.
- Un dossier **Network** contenant le code responsable de la gestion du jeu en réseau. Cela inclut la gestion des connexions réseau, la synchronisation des états de jeu entre différents clients, la gestion des retards réseau, etc. Ce code est crucial pour permettre le jeu en ligne ou en réseau local.
- Un dossier **Render** qui contient des fichiers pour la gestion graphique du jeu et un sous-dossier **Scene** avec les scènes de chaque page du jeu. Les scènes correspondent aux différents affichages de notre interface.

Cette organisation nous a permis de séparer clairement les différentes préoccupations du développement du jeu, de rendre le code plus lisible et plus facile à gérer, et de faciliter la collaboration entre tous les membres de notre équipes.

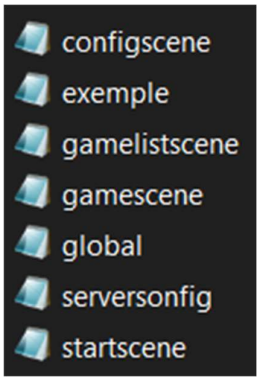
Description des structures de données

Afin d'avoir un code bien optimisé, nous avons créé un système de scènes pour chaque page du jeu qui héritent de la classe « basescene ». Cela permet une meilleure organisation du code et ça améliore la modularité du code. De plus cela nous permet de basculer d'une scène à une autre facilement afin de bien gérer la progression du jeu et la transmission de certaines données du jeu.



Listes des scènes du jeu

Nous avons aussi réparti les structures de données dans plusieurs fichiers textes ce qui nous a permis de personnaliser et de configurer les données facilement. Ces fichiers contiennent différentes informations sur les images de notre jeu tels que la « path » qui est le chemin vers notre image, le « size » qui représente la taille de l'image, « pos » qui est l'emplacement de l'image et « action » qui permet de savoir a quoi correspond l'image.



Listes des fichiers contenant les structures de données

Exemple du contenu des fichiers Json

Nous avons également utilisé des bibliothèques comme « rich.traceback » pour améliorer le traçage des erreurs en affichant les valeurs des variables locales dans la trace des erreurs. C'est utile pour le débogage en fournissant des informations supplémentaires sur l'état du programme lorsqu'une exception se produit.

Présentation des algorithmes de gestion du déplacement des pions.

La méthode « ProcessMove » est utilisée pour traiter le déplacement d'un joueur à une position donnée (pos). Elle vérifie si la position donnée fait partie des mouvements possibles (__possibles_moves) et met à jour la position du joueur. Ensuite, elle vérifie si le joueur a gagné « CheckWin » et passe au joueur suivant en appelant la méthode « SwitchPlayer ».

La méthode « SwitchPlayer » est appelée pour passer au joueur suivant. Elle met à jour la variable (__cplayer) pour définir le nouvel index du joueur courant. Ensuite, elle appelle « ProcessPossiblesMoves » pour calculer les mouvements possibles pour le nouveau joueur courant.

La méthode « ProcessPossiblesMoves » est utilisée pour calculer les mouvements possibles pour un joueur donné. Elle prend en compte la position actuelle du joueur (ppos) et les types de mouvement possibles (__MOVE_TYPE_UP, __MOVE_TYPE_DOWN, __MOVE_TYPE_LEFT, __MOVE_TYPE_RIGHT). Elle vérifie si chaque mouvement est valide en utilisant la méthode « CanMove ». Si un mouvement est valide, il est ajouté à la liste (possibles_moves). La liste des mouvements possibles est ensuite stockée dans (__possibles_moves) si l'appel à cette méthode n'est pas récursif, sinon elle est retournée.

La méthode « CanMove » est utilisée pour vérifier si un mouvement est valide. Elle prend en compte la position actuelle du joueur, le type de mouvement et la position de destination. Elle utilise la méthode « IsPosed » pour vérifier si la position de destination est occupée par une barrière.

Le code gère le déplacement des pions en vérifiant la validité des mouvements possibles et en mettant à jour les positions des joueurs en conséquence.

Présentation des algorithmes de gestion de la pose des barrières.

La Gestion et la pose des barrières se trouve dans la classe « BarrerPart ».

La classe « BarrerPart » a plusieurs méthodes et attributs :

Le constructeur « __init__ » initialise la classe de base « Rect » en lui passant les coordonnées de position, la largeur et la hauteur du rectangle. Il initialise également trois attributs supplémentaires : __id, __vertical, et __posed, qui sont initialisés à None, False, et False respectivement. Ces attributs sont utilisés pour stocker l'identifiant de la partie de la barrière, si elle est verticale ou non, et si elle est posée ou non.

La méthode « SetVertical » est utilisée pour définir si la partie de la barrière est verticale ou non. Elle prend un paramètre booléen vertical et affecte sa valeur à l'attribut (__vertical).

La méthode « IsVertical » retourne la valeur de l'attribut __vertical, indiquant si la partie de la barrière est verticale ou non.

Les méthodes « GetId » et « SetId » sont utilisées pour obtenir et définir l'identifiant de la partie de la barrière.

Les méthodes « IsPosed » et « SetPosed » sont utilisées pour obtenir et définir si la partie de la barrière est posée ou non.

Les méthodes « SetPos » et « GetPos » sont utilisées pour définir et obtenir la position de la partie de la barrière.

Cette classe hérite des fonctionnalités de la classe « Rect » de Pygame et ajoute des attributs supplémentaires et des méthodes pour représenter et manipuler les barrières dans le jeu.

Procédé utilisé pour faire communiquer plusieurs ordinateurs lors du jeu en réseau.

Nous avons 5 fichiers permettant de contrôler toutes la partie réseaux de notre jeu, plus précisément, nous avons :

- Un fichier « **clients.py** », qui est utilisé pour mettre en œuvre la communication client-serveur lors d'un jeu en réseau. Plus précisément, il est utilisé pour créer un client capable d'envoyer et de recevoir des informations d'un serveur de jeu.

Initialisation (NetClient.init) : Lorsqu'un client est créé, une instance de la classe NetClient est initialisée. L'initialisation comprend l'ouverture d'un thread pour la lecture des données à partir du serveur (ReadHandler), l'initialisation d'un objet Stocking qui gère la connexion de socket sous-jacente, et la création d'un dictionnaire actions pour contenir les actions que le client peut effectuer.

Gestion des actions (AddAction, DeleteAction) : Le client peut ajouter ou supprimer des actions, qui sont des fonctions associées à un nom d'action. Ces actions peuvent être déclenchées par des messages reçus du serveur.

Gestion des erreurs (IsStockError, Kick) : Si une erreur se produit lors de la lecture des données du serveur, le drapeau stock_error est défini. De plus, une méthode Kick est fournie pour gérer le cas où le client est expulsé du serveur.

Lecture des données (ReadHandler) : Un thread est utilisé pour lire en continu les données du serveur. Si une donnée est reçue, elle est interprétée comme une chaîne JSON. Si cette chaîne contient une "action", la fonction associée à cette action dans le dictionnaire actions est appelée.

Connexion et déconnexion (Connect, Disconnect) : Le client peut se connecter à un serveur en utilisant la méthode Connect, qui crée un socket, se connecte à l'adresse et au port du serveur et initialise le Stocking avec ce socket. De même, le client peut se déconnecter du serveur en utilisant la méthode Disconnect.

- Un fichier « **gamelistserver.py** », qui met en œuvre un serveur de liste de jeux, qui est une sorte de serveur centralisé que les clients peuvent consulter pour trouver des serveurs de jeu auxquels se connecter.

Initialisation (GameListServer.init) : Lorsqu'un serveur de liste de jeux est créé, une instance de la classe GameListServer est initialisée. L'initialisation comprend la configuration de l'adresse et du port du serveur (ici, '127.0.0.1' et 50000, respectivement), la création d'une liste pour stocker les serveurs connectés (__servers), et l'ajout de deux actions que le serveur peut effectuer en réponse à des messages des clients : retreive_servers et register.

Récupération des serveurs (RetreiveServers) : Lorsqu'un client envoie un message avec l'action "retreive_servers", le serveur de la liste de jeux répond avec un message contenant une liste des adresses de tous les serveurs de jeu actuellement connectés.

Enregistrement des serveurs (Register) : Lorsqu'un serveur de jeu envoie un message avec l'action "register", le serveur de la liste de jeux ajoute le serveur de jeu à sa liste de serveurs et répond avec un message confirmant l'enregistrement.

Suppression des serveurs (RemoveStocking) : Si une connexion avec un serveur de jeu est interrompue, le serveur de la liste de jeux le retire de sa liste de serveurs.

- Un fichier « **gameserver.py** », qui contient la logique pour l'instance de serveur de jeu qui est responsable d'une session de jeu spécifique. Ce fichier contient le processus pour établir une communication entre plusieurs ordinateurs lors d'un jeu en réseau.

Initialisation (GameServer.init) : Lors de l'initialisation d'un serveur de jeu, une instance de la classe GameServer est créée avec une adresse et un port spécifiques (ici, '127.0.0.1' et port 50001, par défaut). Le serveur initialise un jeu (__game), crée une liste de joueurs (__players) et ajoute des actions spécifiques au jeu (telles que "register", "place_barrer", "player_move", "ping") qui définissent la façon dont le serveur doit réagir à des messages spécifiques des clients.

Connexion au GameListServer : Après l'initialisation, le serveur de jeu tente de se connecter au serveur de liste de jeux (ici, '127.0.0.1' et port 50000) et envoie un message pour s'enregistrer.

Gestion des Clients : Le serveur de jeu peut gérer plusieurs clients (joueurs) en même temps. Les fonctions AddClient et RegisterClient sont utilisées pour ajouter un nouveau client à la liste des joueurs et pour envoyer les données initiales du jeu au client, respectivement. Les actions "place_barrer" et "player_move" sont utilisées pour gérer les mouvements des joueurs.

Gestion des erreurs et des déconnexions : Si une erreur se produit dans la connexion d'un client, la fonction StockError est utilisée pour supprimer le client de la liste des joueurs. De même, la fonction Kick est utilisée pour déconnecter un client du serveur.

Répondre aux Pings : Le serveur de jeu utilise également une fonction Ping pour répondre aux requêtes de ping en envoyant un message contenant les détails actuels du jeu, tels que le nombre de joueurs, le nombre maximum de joueurs, le nombre de barrières, la taille du plateau de jeu et l'adresse du serveur.

- Un fichier « **quoridorstocking.py** », qui contient la classe QuoridorStocking qui hérite de la classe Stocking. Elle est utilisée pour encapsuler et gérer les détails spécifiques de la connexion pour notre jeu de quoridor.

init(self, server, conn, id, fatal=False) : Cette méthode initialise une nouvelle instance de QuoridorStocking. Elle prend le serveur, la connexion, un identifiant unique pour la connexion et un indicateur pour savoir si une déconnexion doit être considérée comme fatale.

close(self) : Cette méthode est utilisée pour fermer la connexion. Avant de fermer la connexion, elle informe le serveur de supprimer cette connexion de sa liste de connexions actives.

GetId(self) : Cette méthode retourne l'ID unique de cette connexion.

IsFatal(self) : Cette méthode retourne un booléen indiquant si une déconnexion doit être considérée comme fatale.

IsDisconnected(self) : Cette méthode retourne un booléen indiquant si la connexion est déconnectée.

Disconnect(self) : Cette méthode est utilisée pour déconnecter la connexion.

- Un fichier « **server.py** », qui définit une classe « Server », celle-ci est utilisée pour gérer et administrer les connexions pour le jeu Quoridor.

init(self, console, host, port) : Initialise une nouvelle instance de Server, crée un socket d'écoute et démarre un nouveau thread pour gérer les nouvelles connexions. Elle enregistre également certaines commandes sur la console.

GetStocks(self): Renvoie la liste de toutes les connexions (stockings).

Write(self, stock, data) : Écrit des données sur une connexion donnée. Gère les erreurs de pipe cassé et les erreurs de réinitialisation de connexion en supprimant la connexion cassée.

AddAction(self, name, func) : Ajoute une action que le serveur peut effectuer. Cette action peut être invoquée en envoyant des données avec le nom de l'action comme champ "action".

StockError(self, stock, err) : Cette méthode n'est pas implémentée dans le code fourni, mais elle est vraisemblablement destinée à gérer les erreurs qui se produisent sur une connexion.

ReadHandler(self): Lit les données de tous les clients connectés et invoque l'action appropriée pour chaque élément de données.

RemoveStocking(self, stock): Supprime une connexion de la liste des connexions actives.

GetStockings(self): Renvoie la liste de toutes les connexions actives.