Rapport de soutenance J3D

1. Review du TSD :

Par rapport aux objectifs fixés dans le cahier technique pour la première soutenance, certains points ont évolué différemment de ce qui était initialement prévu, principalement en raison de nos choix de priorisation.

Les valeurs initiales de la comparaison se trouvent en annexe 1.1.

Pixel art

Avancement attendu : 20 %

Avancement estimé : ~5 %

Les éléments réalisés sont uniquement des tuiles de test, qui ne seront pas conservées dans la version finale. Ce retard n'est pas problématique, car l'équipe a fait le choix de prioriser le développement technique plutôt que l'aspect visuel à ce stade du projet.

Musique

Avancement attendu : 0 %

Avancement estimé : ~20 %

Des musiques issues de packs ont déjà été sélectionnées et le travail sur les bandes-son originales a commencé, ce qui nous donne un avancement supérieur à l'estimation initiale.

Animation

Avancement attendu : 0 %

Avancement estimé : 0 %

Aucune animation n'a encore été réalisée, ce qui est cohérent avec l'état actuel du projet, puisqu'il n'y a pas encore de gameplay ni de pixel art définitif à animer. L'avancement sur cette partie concorde avec l'estimation.

Sound design

Avancement attendu : 0 %

Avancement estimé : 0 %

Le sound design n'a pas encore été abordé. Il sera intégré en fin de projet afin de peaufiner l'ambiance sonore une fois le gameplay développé correctement. Tout comme les animations l'avancement concorde avec l'estimation pour cette partie.

Multijoueur

Avancement attendu : 100 %

Avancement estimé : 90 %

Le multijoueur est entièrement fonctionnel pour la version actuelle du code. Il nous permet de faire communiquer les deux ordinateurs en P2P d'une manière fluide et sans latence. Il pourra néanmoins nécessiter des ajustements ultérieurs pour intégrer les énigmes et les ennemis.

Gameplay

Avancement attendu : 50 %

Avancement estimé : 0 %

Aucune interaction ni compétence n'est encore implémentée. Le gameplay sera développé pour la seconde soutenance, nous avons sous-estimé les priorités précédent l'implémentation des compétences des joueurs ainsi que des interactions avec l'environnement.

Map

Avancement attendu : 20 %

Avancement estimé : ~20 %

La structure générale de la carte est définie à travers des schéma et les idées sont écrites, il manque maintenant la matrice 3d qui représentera la map. Nous avons avancé à la vitesse attendue sur ce point. D'ici la seconde soutenance, nous prévoyons d'avoir une map complète du jeu sans inclure le combat final encore.

Déroulement du jeu

Avancement attendu : 80 %

Avancement estimé : 80 %

Le scénario global et les différentes étapes de progression sont entièrement écrits. Les détails sur le combat final sont encore manquants, nous y consacrerons du temps durant la dernière étape avant la troisième soutenance. L'avancement estimé est donc atteint.

Affichage du jeu

Avancement attendu : 80 %

Avancement estimé : ~60 %

L'affichage fonctionne correctement pour les tuiles de test qui pourront être remplacées facilement par les tuiles finales. L'affichage des deux joueurs est aussi géré. L'affichage est optimisé et permet l'utilisation de matrice conséquente (+ de 100 000 cases). Seulement il manque encore l'affichage des objets et différents ennemis avec qui les joueurs interagiront.

L’avancement attendu comprenait l’affichage des objets, en plus des tuiles et des joueurs. Nous attendons d’avoir la carte complète sous forme de matrice pour ajouter les objets. La sous-estimation du temps que prendrait le développement de cette matrice explique le retard.

Développement des joueurs

Avancement attendu : 70 %

Avancement estimé : ~30–40 %

Les classes des joueurs existent,et permettent l’affichage et le déplacement. Il manque donc les compétences ainsi que les différentes propriétés telles que les points de vie ou l’énergie. Nous avons sous estimé le temps que l’implémentation de ces différentes fonctionnalités demandaient et avons prioriser le développement du multijoueur et de l’affichage, ce qui explique le retard sur cette tâche.

Affichage du menu

Avancement attendu : 30 %

Avancement estimé : ~50 %

L’interface du menu est plus avancée que prévu, son implémentation s’étant révélée plus simple et plus rapide que ce qui avait été anticipé. Il nous restera les détails du design ainsi que certains bugs à réparer pour les prochaines soutenances.

Intelligence artificielle (IA)

Avancement attendu : 20–30 %

Avancement estimé : 0 %

L’IA n’a pas encore été développée. Ce choix est volontaire car nous avons décidé de prioriser les bases du jeu que sont l’affichage et le multijoueur, aussi nous allons débuter le développement de l’IA ennemi seulement une fois l’environnement des énigmes mis en place.

2. Rapports personnels

2.1. Antoine LAPP :

Mon travail a été principalement consacré à l’implémentation de la partie multijoueur du jeu.

1. **Choix de la technologie et de la librairie**

J’ai commencé par me renseigner sur les différents types de multijoueur qu’il était possible d’implémenter en python. Il en existe 3 principaux :

- P2P (Peer To Peer) : chaque joueur envoie ses informations à tous les autres joueurs
- LAN : Tous les joueurs sont sur le même réseau local (wifi ou filaire) et ensuite il y a deux possibilités, soit on fait un P2P local et tous les joueurs s’envoient leurs

informations entre eux, soit tout le monde envoie à un même joueur qui joue le rôle de serveur
- Online : tous les joueurs se connectent à un serveur externe qui donne les informations des autres joueurs et fait les calculs notamment pour la gestion d'ennemis.

Je suis arrivé à la conclusion que le P2P était le plus adapté à notre projet car tout d'abord, cela évite les frais lié à l'hébergement d'un serveur et en plus comme on fixe le nombre de joueur à 2, on a pas de risques de saturation qu'on peut avoir avec beaucoup de joueur en P2P.

J'ai par la suite recherché comment il était possible de faire du P2P en python. J'ai fait face à un problème : pour faire du P2P entre 2 machines qui ne sont pas sur le même réseau, il faut que la machine du joueur 1 contacte le routeur du réseau du joueur et qu'il lui demande de se connecter à la machine du joueur 2. Pour des questions de sécurité, ceci implique que le joueur 2 doit modifier certains paramètres de son réseau. Or ceci est très contraignant et souvent impossible. Par exemple, si un joueur est connecté au wifi de l'école ou un autre wifi public, il ne va pas pouvoir modifier les paramètres de ce réseau. Pour contourner ce problème, il faut faire ce qui s'appelle un NAT Transversal qui se fait grâce à différents protocoles réseaux (STUN,TURN,ICE). Ceux-ci sont à la base de la technologie WebRTC qui est utilisée dans de nombreux jeux-vidéos, mais aussi dans des applications telles que Zoom ou Discord. Pour utiliser cette technologie en python, on ne peut pas utiliser de librairies standards telles que *socket*, la seule librairie moderne qui gère ça est *aiortc*.

2. Fonctionnement du P2P avec *aiortc*

Ensuite, pour bien comprendre le fonctionnement de la librairie et comment implémenter du multijoueur dans un jeu Pygame, j'ai développé de mon côté un mini jeu très simple dans lequel chaque joueur a une boule qu'il peut déplacer sur un fond noir, et ses déplacements ont reproduits sur l'écran de l'autre joueur. Voici comment cela fonctionne concrètement :

- Le joueur 1 que l'on qualifie de "host" va créer une offre SDP (un code JSON). On peut retrouver un exemple de clé SDP dans l'image 2.1.1 de l'annexe.
  Cette clé est ensuite transmise à l'autre joueur par un moyen externe.
- Le joueur 2, appelé "client" va créer une réponse SDP (aussi au format JSON) à partir de l'offre du joueur 1. Ensuite il transmet cette réponse SDP au joueur 1.
- Maintenant les 2 machines savent avec qui elles doivent communiquer, un datachannel est créé entre elles, elles peuvent s'envoyer des messages textuels.

3. Implémentation d'une base de donnée pour les codes de partie

Ceci fonctionne, mais il n'est pas très pratique de devoir partager des clés SDP aussi longues et difficiles à retenir. Il serait plus simple d'avoir des codes de parties de quelques lettres comme dans le jeu *Among US*. Pour cela, on a besoin d'une base de données qui stock chaque code de partie et offre et réponse SDP qui lui sont associées. Il est possible d'héberger une base de données gratuitement via le service firebase de Google. Cette base

de données est accessible via une API REST. On peut donc lire ou modifier une ligne pour y ajouter la réponse ou offre SDP grâce à la librairie *requests* de python.
Le joueur host génère donc un code de partie qui n'existe pas encore dans la base de données ainsi que son offre SDP (et l'affiche à l'utilisateur). Il crée une nouvelle entrée dans la base de données dont l'identifiant est le code de partie. Ensuite, il vérifie tous les x temps si une réponse à été renseigné pour initialiser la connexion. De son côté, le client récupère l'offre SDP dans la base de données grâce au code fourni par le joueur, génère sa réponse et la met dans la base de données. Ensuite, il attend que la connexion soit établie pour que le jeu puisse commencer.

4. Gestion des boucles

Au moment d'implémenter la partie jeu avec ce code multijoueur, j'ai fait face à une difficulté supplémentaire : la gestion des boucles.

Notre jeu, pour fonctionner, a besoin de faire tourner plusieurs boucles :

- la boucle du jeu : elle sert à mettre à jour l'affichage du joueur. Cette boucle tourne à une fréquence de 60 FPS : on réaffiche le jeu 60 fois par seconde
- la boucle de connexion (boucle du datachannel) : pour que le datachannel entre les 2 joueurs reste ouvert, il faut qu'une boucle tourne en continu.
- la boucle d'envoie des données : c'est une boucle qui permet d'envoyer les données aux joueurs, lors de mes tests, il était possible de faire du 60 FPS aussi car je n'avais pas beaucoup d'infos à envoyer, mais si notre jeu se complique et qu'on doit partager plus d'infos, ça peut entrainer un ralentissement donc un moyen d'optimiser est d'envoyer les infos à une fréquence plus faible (30 FPS par exemple).

Donc pour fonctionner correctement, notre jeu a besoin que ces 3 boucles tournent en même temps. Or en python ce n'est pas possible : les boucles s'inter-bloquent entre elles.
Une première solution que j'ai trouvé est de combiner les boucles de jeu et d'envoi des données en une seule boucle : La boucle tourne à la vitesse de la boucle du jeu (60FPS) et met à jour l'affichage du jeu à chaque tour. Et on a une variable `network_interval` en milliseconde qui définit à quelle fréquence on envoie les données sur le réseau. Pour faciliter la compréhension, toutes les actions qui doivent être réalisées dans la boucle du jeu (en dehors de l'envoie des données) sont stockées dans une fonction `update_game()` de `game_logic.py`.
En revanche, la boucle qui permet de maintenir la connexion avec *aiortc* ne peut pas être combinée avec l'autre boucle. Pour que ces deux boucles puissent tourner en même temps, on va devoir faire du threading : on va placer chaque boucle dans une boite indépendante (appelée thread). Pour cela on utilise la librairie *threadings*. Une information à prendre en compte est que pygame doit obligatoirement tourner dans le thread principal, c'est donc la boucle de maintien de la connexion qui est placée dans un thread séparé. Si des données doivent être connues et pouvoir être modifiées par les 2 boucles, il faut les déclarer de façon globale dans le fichier python.

5. Partage des informations entre les fichiers

Afin d'augmenter la clarté et de faciliter le développement du jeu, on a séparé le code source dans plusieurs fichiers python : un fichier pour la partie réseau, un pour le menu, un pour le jeu en lui-même, un fichier de lancement… Mais ces fichiers ont besoin de se partager des informations entre eux et il faut que ces informations soient synchronisées et les mêmes dans tous les fichiers. Par exemple, il ne faut pas qu'un fichier croit qu'un joueur se trouve à certaines coordonnées et qu'un autre fichier croit qu'il se situe à un autre endroit. Pour centraliser tout cela, j'ai créé une classe `GameContext` dans `game_context.py`. Pour l'instant, cette classe contient ces attributs :

- La fenêtre et la clock pygame, qui sont nécessaires pour afficher des éléments et gérer la boucle principale du jeu.
- Les dimensions de la fenêtre
- Le joueur host et le joueur client représentés par des objets de la classe `player`.
- Le code de partie si une partie est en cours. Si aucune partie n'est en cours, c'est une chaîne vide. Si le joueur client a entré un code non valide, la valeur est à `wrong_code`
- Un booléen `is_host` qui indique si le joueur local (qui a lancé l'instance du jeu en cours) est host ou non.
- La map
- L'objet caméra qui est centré sur le joueur

Au lancement du jeu, le fichier principal (`launch.py`) crée un objet de cette classe et cet objet sera partagé aux autres fichiers en paramètre de fonction quand ceux-ci seront appelés.

6. Prochaines missions

Actuellement, la partie réseau du jeu est globalement finie, il sera nécessaire de faire quelques adaptations assez simples pour pouvoir également partager la position des ennemis.
Je vais maintenant me charger d'adapter la fenêtre du jeu à la taille de l'écran de l'utilisateur. Je vais également aider à implémenter les différentes énigmes au sein du jeu. Je vais également essayer d'implémenter un menu pause.

7. Ressources

Voici différentes ressources qui m'ont été utiles pour la réalisation de mon travail :

- https://code.tutsplus.com/building-a-peer-to-peer-multiplayer-networked-game--gamedev-10074t
- https://aiortc.readthedocs.io/en/latest/
- https://dev.to/whitphx/python-webrtc-basics-with-aiortc-48id

2.2. Guillaume Klein :

Je me suis chargé du développement du moteur isométrique au sein de `isometric_motor.py`, de la détection des collisions avec les murs dans `game_logic.py` ainsi que de la création des premières tuiles en pixel art pour tester le moteur.

1. **Objectif du moteur**

Mettre en place un moteur d'affichage isométrique fonctionnel, servant de base visuelle au jeu *Echoes Of Lights.* Il doit permettre :

- l'affichage d'une carte en vue isométrique
- la gestion d'une map en 3 dimensions
- le rendu de deux joueurs en coopération
- la mise en place d'une caméra dynamique
- une optimisation du rendu en n'affichant que les zones utiles

Ce moteur constitue une fondation technique sur lequel sera ajouté ultérieurement le gameplay, les interactions, les animations et les assets finaux.

2. **Représentation de la carte**

La carte du jeu est gardé en mémoire sous la forme d'une matrice 3D, structurée selon trois axes :

- X : hauteur de la carte
- Y : largeur de la carte
- Z : niveaux de hauteur (relief)

Chaque case de la matrice contient un entier correspondant à un type de tuile (il y aura bien plus de tuiles par la suite) :

- 0 : vide
- 1 : sol
- 2 : mur

Une carte de test est générée procéduralement afin de donner une base de lancement au jeu. Elle prend la forme d'un rectangle avec des murs sur les bords, ce qui permet de tester les collisions et d'empêcher les joueurs de sortir de la zone jouable.
Cette représentation cartésienne facilite la gestion logique du terrain, des collisions (plus tard dans le rapport) et des futures interactions.

3. Conversion cartésien vers isométrique

Le cœur du moteur repose sur la conversion des coordonnées cartésiennes vers des coordonnées isométriques, nécessaires à l'affichage.
La fonction `cart_to_iso` permet cette transformation :

- Les coordonnées de la matrice (x, y, z) sont convertis en coordonnées écran (screen_x, screen_ y)
- La largeur et la hauteur des tuiles (TILE_WIDTH, TILE_HEIGHT) sont utilisées pour respecter les proportions isométrique
- La coordonnée z influence l'axe vertical afin de simuler la hauteur
- Un offset est appliqué pour centrer la carte à l'écran

Les calculs à partir de `x`, `y` et `z` les coordonnées cartésiennes, `TILE_WIDTH` et `TILE_HEIGHT` les mesures des tiles sont les suivants:

```
screen_x = (x - y) * (TILE_WIDTH // 2) + 400
screen_y = (x + y) * (TILE_HEIGHT // 2) - z * TILE_HEIGHT
```

Cette méthode permet de conserver une logique simple du côté du stockage, tout en affichant correctement la carte en vue isométrique à partir des fonctions de transformation.

4. Conversion isométrique vers cartésien

Afin de pouvoir retrouver la position d'un joueur sur la carte à partir de sa position à l'écran, une fonction inverse a été implémentée: `iso_to_cart_tile` .

Cette fonction :

- retire l'offset de la caméra,
- inverse les calculs de projection isométrique,
- retourne les coordonnées cartésiennes correspondantes dans la matrice.

Les joueurs stockent seulement les coordonnées isométriques dans leur classe Joueur et cette fonction permet de retrouver leur position sur la matrice.
Les calculs à partir de `iso_x` et `iso_y` les coordonnées cartésiennes, ainsi que de `TILE_HEIGHT` et `TILE_WIDTH` les dimensions des tuiles:

Enlever l'offset du centrage caméra :

```
iso_x = screen_x - 400
iso_y = screen_y + z * TILE_HEIGHT
```

Puis inverser la matrice isométrique :

```
cart_x = (iso_y / (TILE_HEIGHT / 2) + iso_x / (TILE_WIDTH / 2)) /
2
cart_y = (iso_y / (TILE_HEIGHT / 2) - iso_x / (TILE_WIDTH / 2)) /
2
```

Elle permet alors de :

- déterminer la position du joueur,
- gérer les collisions,
- et préparer les futures interactions avec l'environnement (notamment des possibles plaques de pressions, leviers, et autres).

5. Gestion de la caméra

Une classe `Camera` permet de simuler une caméra dynamique qui suit la position du joueur en train de jouer. Son rôle est d'appliquer un offset global à toutes les coordonnées affichées, sans modifier les données internes de la carte.

Les différentes méthodes :

- `follow` permet donc de calculer l'offset pour les coordonnées ajustées à partir d'un facteur de lissage et de la position du joueur, le facteur de lissage permet de rendre le décalage fluide et plus agréable.
- `apply` renvoie des coordonnées auquel est ajouté l'offset

Cette approche rend le système modulaire et facilement ajustable.

6. Optimisation de l'affichage

Pour éviter beaucoup de calculs inutiles, un système d'affichage partiel de la carte est mis en place grâce à la fonction `display_ranges`.
Cette fonction détermine une zone rectangulaire de la matrice autour du joueur, correspondant aux tuiles à l'écran.

On peut voir sur l'annexe 2.2.6 que les cases aux extrémités ne sont pas affichées car j'ai réduit la portion de cases affichées manuellement pour que cela soit visuel.
Ainsi, seules les tuiles proches sont parcourues et affichées, cela améliore les performances et permet d'envisager une carte nettement plus grande grâce au coût du rendu qui est diminué.

7. Classe Map et rendu de la scène

La classe Map centralise l'affichage des tuiles de la carte ainsi que les deux joueurs. Par la suite, cet affichage sera complété par l'environnement pour les interactions et les énigmes.
Le rendu est effectué du fond vers l'avant en respectant un ordre isométrique, ce qui garantit un affichage correct des superpositions.
L'affichage de la matrice 3D est réalisé en respectant l'ordre propre à la vue isométrique afin de garantir une superposition correcte des éléments. La carte est parcourue colonne par colonne, en dessinant les tuiles du fond vers l'avant grâce à 3 boucles for qui itèrent sur les indices donnés par la fonction `display_ranges`. Pour chaque colonne, l'affichage se fait

ligne par ligne sur l'axe Y, puis sur l'axe X, ce qui permet de respecter la profondeur visuelle de la scène.
Cet ordre de rendu assure que les tuiles et les entités les plus proches du joueur sont affichées au-dessus des éléments plus éloignés, sans nécessiter de tri supplémentaire.

Voir annexe 2.2.7 pour un schéma

Les joueurs sont affichés sur un niveau spécifique (z = 1) avec un décalage vertical adapté à la hauteur de leur sprite, afin de donner l'impression qu'il reposent correctement sur le sol.

8. Les collisions

Au sein du fichier `game_logic.py`, dans la fonction `update_game`, j'ai développé la détection de collisions avec les murs de la carte.
La position du joueur avec ses coordonnées isométriques (x,y) est stockée dans la classe Joueur, seulement les coordonnées représente un point et le sprite possède une largeur qui implique des fonctions auxiliaire pour détecter le dépassement de sa position sur un mur pour le pied gauche et droit.
Les coordonnées stockées sont celles correspondantes au pied droit, il faut donc déduire la coordonnée du pied gauche pour vérifier la collision proprement.

La fonction `deduce_foots_from_iso_coords` défini dans isometric_motor.py me permet de déduire la position des pieds du personnage à partir de sa largeur en pixel et de la position stockée au sein de sa classe. La position correspond au pied droit du joueur, on retourne ainsi la coordonnée x en retirant la largeur du sprite pour avoir la position de l'autre pied.

Finalement dans la fonction `update_game` où les déplacement sont mis en place grâce à l'écoute sur les différentes touches de déplacement, qui sont actuellement les flèches, je vérifie la collision directement dans la matrice grâce aux deux paires de coordonnées de chacun des pieds calculés à partir de la fonction `deduce_foots_from_iso_coords`.
Pour les déplacements horizontaux, le code ne vérifie que les collisions dans la direction utile (Gauche pour les déplacements vers la gauche et vice versa).
Pour les déplacements verticaux, les deux pieds sont vérifiés.

9. État actuel et perspectives

A ce stade, le moteur permet :

- l'affichage d'une carte isométrique en tuile pixel art,
- le rendu multi-niveaux,
- la gestion d'une caméra fluide et agréable au déplacement
- l'optimisation de l'affichage,
- l'intégration de joueurs avec collisions basiques.

10. Prochaines missions

À partir de ce moteur qui constitue la base technique de l'affichage, nous allons développer les différentes tuiles nécessaires à la création des énigmes.
Je vais me charger des tâches suivantes pour la prochaine soutenance :

- ajouter la possibilité d'afficher les objets de l'environnement qui permettent des interactions,
- développer les fonctions permettant l'affichage des ennemis,
- créer la matrice 3d de la map intégrale comprenant les 4 branches ainsi que la matrice du combat final.

11. Ressources

Les différents références dont je me suis servi pour mon travail :

- https://www.reddit.com/r/gamedev/comments/v2duc7/how_isometric_coordinates_work_in_2d_games/?tl=fr
  https://www.reddit.com/r/gamemaker/comments/8k9fqq/intro_to_isometric_projection/?tl=fr
- https://www.youtube.com/watch?v=2Ucv9XM0pFI
- https://fr.wikipedia.org/wiki/Isom%C3%A9trie

2.3. Antoine Muh:

Je me suis occupé de programmer le menu principal du jeu *Echoes of Light* en utilisant la bibliothèque Pygame. J'ai mis en place toute la structure du menu, l'affichage des différents écrans, la gestion des boutons, de la musique et des options accessibles au joueur.

1. Rôle du menu

Le menu sert de point d'entrée au jeu. Il permet au joueur de lancer une nouvelle partie, de charger une partie, d'accéder aux options ou de quitter le jeu. J'ai conçu ce menu pour qu'il soit simple à utiliser et facile à faire évoluer plus tard, lorsque le gameplay sera ajouté.

2. Structure générale du programme

Le programme commence par initialiser Pygame, puis crée une fenêtre de 1280 par 720 pixels. Une image de fond est chargée et redimensionnée pour s'adapter à la fenêtre. Une musique de menu est également chargée avec `pygame.mixer.music`, jouée en boucle, et son volume est stocké dans une variable afin de pouvoir être modifié pendant l'exécution du programme.
Le fonctionnement du menu repose sur une boucle principale qui tourne à 60 images par seconde grâce à `pygame.time.Clock`. Cette boucle permet de mettre à jour l'affichage et de gérer les interactions du joueur en temps réel.

3. Gestion des écrans

J'ai utilisé une variable appelée `page` pour savoir quel écran doit être affiché. Selon sa valeur, le programme affiche le menu principal, l'écran de nouvelle partie, l'écran de chargement, ou le menu des options. Cette méthode permet de gérer plusieurs écrans sans avoir besoin de créer plusieurs programmes différents.
Chaque écran possède ses propres boutons et éléments graphiques, mais tous partagent la même structure de base.

4. Boutons et intéractions

Pour éviter de répéter du code, j'ai créé une classe `Button`. Cette classe gère l'affichage des boutons, leur position, leur effet de survol avec la souris et la détection des clics. Les boutons sont dessinés à l'aide de surfaces transparentes et de rectangles arrondis, ce qui permet d'avoir un rendu plus propre.
Un système de délai entre deux clics est mis en place avec un cooldown afin d'éviter que plusieurs actions se déclenchent en même temps lors d'un clic trop rapide.

5. Saisie clavier

Sur les écrans de nouvelle partie et de chargement, j'ai ajouté un champ de saisie pour entrer un code multijoueur. Cette saisie est gérée avec les événements clavier de Pygame. Le code est stocké dans une variable et limité à douze caractères. La touche retour arrière permet de supprimer des caractères.

6. Menu des options

Le menu des options permet de changer la langue et le volume de la musique. Les textes sont stockés dans un dictionnaire multilingue, ce qui permet de changer toute l'interface simplement en modifiant une seule variable.
Le volume est réglé avec une barre interactive. Lorsque le joueur clique sur la barre, la position de la souris est utilisée pour calculer une valeur comprise entre 0 et 1, qui est ensuite envoyée à `pygame.mixer.music.set volume` pour modifier le son en temps réel.

7. Gestion des événements

Le programme écoute en permanence les événements Pygame comme les clics de souris, les touches du clavier et la fermeture de la fenêtre. Lorsque l'utilisateur quitte le jeu, Pygame est correctement fermé avec `pygame.quit()` et le programme s'arrête proprement avec `sys.exit()`.

8. **État actuel**

À ce stade, le menu permet :

- d'afficher plusieurs écrans à partir d'un seul programme,
- de naviguer facilement grâce aux boutons,
- de gérer la musique et le volume,
- de changer la langue de l'interface,
- de récupérer les entrées clavier et souris.

9. **Prochains objectifs**

- Perfectionnement de l'interface utilisateur en la rendant plus agréable à regarder.
- L'implémentation des sons et musiques dans le jeu créé par Valentin.
- Ajout des animations notamment des joueurs avec les designs de pixel art.

10. **Ressources**

Les différents références dont je me suis servi pour mon travail :

- https://www.youtube.com/watch?v=iJ7DDqaGbG4
- https://people.montefiore.uliege.be/mathy/ppi/flappy.html
- https://www.reddit.com/r/pygame/comments/1grzi4g
- https://fr.musicful.ai/ai-music-generator/
- https://www.pixellab.ai/

2.4. Gustave Schwien :

Je travaille sur développement du gameplay, en particulier des différentes salles d'énigmes et de combat du jeu, ainsi que sur l'intelligence artificielle dont l'implémentation est obligatoire. J'aide également à la création de la musique et des effets audio.
Jusqu'à maintenant, j'ai principalement travaillé sur la structuration des différentes branches du sanctuaire et sur la recherche concernant le fonctionnement de l'intelligence artificielle que l'on retrouvera dans le jeu.

1. **Clarifications sur les personnages**

Nos deux personnages, Aeden et Lyra, sont liés. Aeden contrôle la lumière, Lyra l'ombre ; mais nous ne leur avions pas encore assigné de capacités précises. J'ai donc commencé par clarifier ces particularités avec le groupe. Nous avons décidé que chaque personnage allait posséder :

- Une capacité utilisable de loin.
- Une capacité utilisable de près.
- Une arme "corps-à-corps" pour infliger des dégâts aux ennemis.

Voici ce qui a été décidé :

Aeden (lumière) :
Capacité de loin : projection de faisceaux lumineux capables d'activer certains mécanismes
Capacité de près : flash de lumière capable de rendre visible des structures et objets invisibles, ainsi que d'éblouir certains ennemis
Attaque corps-à-corps : épée "Espadon des Lumières"

Lyra (ombre) :
Capacité de loin : déplacer et modifier la taille de structures particulières appelées "structures d'ombre" (qui peuvent être des cubes, des murs…)
Capacité de près : mode "ombre" : capacité lui permettant de traverser les murs non pleins. Dans ce mode, Lyra ne peut que se déplacer et ne peut rien faire d'autre
Attaque corps-à-corps : épée "Lame des Ombres"

Ce point nous a permis de démarrer la conception des énigmes.

2. Conception des énigmes

J'ai créé une liste des différents types d'énigmes, en détaillant pour chaque énigme :
- Son fonctionnement global, en quoi elle consiste, si les deux personnages sont dans la même salle ou pas (cela dépend des énigmes)
- Son design théorique pour les graphismes de la salle (il sera adapté en fonction des branches, leur thème étant toujours différent)
- L'objectif, les différentes contraintes et la solution de l'énigme si nécessaire

Pour créer ces énigmes, il a fallu prendre en compte plusieurs conditions :
- L'énigme nécessite l'utilisation d'au moins une capacité par personnage
- L'énigme n'est pas lassante : elle n'est ni trop courte, ni trop longue (résolvable en quelques minutes)
- L'énigme est modulable : il est possible de créer facilement différentes formes de l'énigme pour en implémenter d'autres versions dans d'autres branches ou pour faire varier le niveau de difficulté (voir 4.)
- L'énigme est raisonnable d'un point de vue développement : une énigme trop difficile à développer pourrait comporter de nombreux bugs difficiles à résoudre, ce qui impliquerait du temps de débogage inutile ainsi qu'un impact négatif sur l'expérience de jeu.

Un exemple d'énigme se trouve en annexe.

En plus des énigmes, les branches comporteront des salles de combat.

3. Conception des salles de combat

En parallèle à la liste des énigmes citée plus haut, j'ai également travaillé sur la conception des salles de combat : ces salles seront implémentées dans les branches, en plus des énigmes. En théorie, on devrait retrouver 4 (minimum) à 6 (grand maximum) salles de combat par branche, toutes les 2 ou 3 salles d'énigmes. En somme, une branche devrait retrouver une quinzaine de salles en moyenne. Pour certaines salles de combat les deux personnages seront séparés, pour d'autres ils combattront ensemble.
Dans une salle de combat se trouveront des ennemis. Selon la difficulté de la salle (voir 3.), pourront varier :

- Le nombre d'ennemis
- Leur vitesse d'attaque et de déplacement
- Leur nombre de points de vie
- Leur type : les ennemis les plus coriaces imposeront l'utilisation des capacités des deux personnages en même temps (notamment dans la dernière salle d'une branche)
- La taille de la salle

La dernière salle d'une branche sera toujours une salle de combat commune, après laquelle les deux personnages devront unir leurs capacités pour ouvrir l'accès à l'autel sur lequel repose l'artefact.
Pour augmenter le nombre d'ennemis, donc augmenter la difficulté et rendre les salles plus intéressantes, j'ai évoqué la possibilité de les faire apparaître progressivement (par vagues) dans la salle, de manière à ne pas être limité par sa taille. L'implémentation de cette méthode dépendra de notre ressenti sur la difficulté des salles lors de nos premiers tests.

4. Elaboration des niveaux de difficulté

En travaillant sur la conception des énigmes et des salles de combat, j'ai eu l'idée de diviser les branches en trois niveaux de difficulté :

- Niveau 1 : niveau facile. C'est le niveau des premières salles d'une branche.
- Niveau 2 : niveau moyen. Les énigmes et les combats se corsent : les personnages doivent utiliser leurs capacités pour vaincre les ennemis. C'est le milieu d'une branche.
- Niveau 3 : niveau difficile. Les énigmes sont plus longues et les personnages peuvent avoir besoin d'unir leurs capacités pour arriver à bout de certains ennemis. Cela correspond aux dernières salles d'une branche.

Comme les énigmes et les salles de combat ont été faites pour être modulables, il suffit de modifier certains paramètres pour chaque salle pour faire varier le niveau de difficulté. De cette manière, le joueur peut constater son évolution dans une branche, ce qui rend le jeu plus intéressant.

5. L’intelligence artificielle (IA)

J’ai commencé à faire des recherches sur les possibles implémentations de l’IA dans le jeu, et sur les manières de la développer en Python.
Nous avons décidé que l’IA serait implémentée dans les programmes des ennemis : je me suis donc renseigné sur le fonctionnement général d’un ennemi dans un jeu vidéo. Dans notre cas, son objectif principal est d’aller vers les joueurs pour les attaquer. Il faut donc qu’il soit capable de :

- Détecter un joueur.
- Se déplacer en autonomie en trouvant le meilleur chemin pour aller de sa position de départ au joueur.
- Attaquer le joueur.

Pour détecter un joueur, le programme est simple : toutes les X images du jeu (*frames*), ou toutes les X secondes, l’ennemi calcule sa distance entre lui et les 2 joueurs à partir de leurs coordonnées cartésiennes avec la formule :

$$d = \sqrt{(x_b - x_a)^2 + (y_b - y_a)^2 + (z_b - z_a)^2}$$

La variable d étant la valeur que l’on compare à une valeur de référence choisie. Si d <= référence alors l’ennemi détecte le joueur.
Il est possible de ne pas utiliser la fonction sqrt de Python si on compare d² à référence² : cela peut simplifier le programme.

Si un joueur est détecté, alors l’ennemi va devoir se déplacer vers lui. Pour ce faire, il va déterminer le meilleur chemin pour se rendre à la position du joueur en prenant en compte les obstacles (murs par exemple). On peut alors utiliser la méthode A* (dite “A star”) : elle permet de tester tous les chemins possibles (où il n’y a pas d’obstacles) et de garder le plus court tout en prenant en compte le chemin déjà parcouru, à la manière d’un GPS. C’est une méthode très répandue dans le domaine du jeu vidéo, spécifiquement pour les ennemis, et très faisable en Python.
Enfin, pour attaquer le joueur, l’ennemi pourra une nouvelle fois comparer sa distance avec le joueur : si elle passe en dessous d’une autre valeur de référence, alors l’ennemi pourra attaquer le joueur à l’aide d’une attaque spécifique à l’ennemi.

6. Prochaines missions

Maintenant que nous avons une idée plus claire des particularités des personnages, de la structure des branches et de la consistance des énigmes et des salles de combat, nous allons pouvoir commencer à esquisser les différentes salles à l’aide du moteur de Guillaume.

Concernant la partie IA, comme nous avons une idée assez claire de son implémentation, je dois pouvoir commencer le développement en Python. La difficulté principale sera de fusionner mon code avec celui du reste du jeu.

En plus des différents ennemis, le boss final aura également sa propre IA. Son comportement reste à définir car nous devons encore discuter de cette partie du jeu.

Je participerai également à la création de la musique et des effets audio du jeu. Maintenant que la structure des branches est plus claire (énigmes, salles de combat, autels des artefacts, boss final…), nous avons une meilleure idée de la musique qu'il va falloir produire.

7. **Ressources**

Sources de mes recherches sur l'algorithme A* :
- https://fr.wikipedia.org/wiki/Algorithme_A*
- https://www.datacamp.com/fr/tutorial/a-star-algorithm
- https://youtu.be/iMRJzA8BBIg?si=c0nFR3fhLia1vOxr

2.5. Valentin Goussot :

Je suis chargé de la composition des musiques et la création du Sound design et également de la conception du site web. Je dois créer des sons pour que les joueurs aient une immersion totale dans le jeu.

Pour la musique : je m'occupe de la composition des différents thèmes qui accompagnent les joueurs pendant le gameplay. Chaque morceau cherche à soutenir l'ambiance générale du jeu et à renforcer l'immersion des joueurs.

1. **Menu principal :**

   J'ai conçu la musique du menu principal pour que dès le lancement du jeu, il y ait une ambiance mystérieuse et immersive. Elle combine des sonorités douces à progression harmonique lente. Je l'ai structurée pour qu'elle soit facilement répétable ce qui permet de l'écoute sur une longue durée pour accompagner plus naturellement le joueur avant de lancer une partie.

2. **Musique de combat :**

   J'ai ensuite imaginé les musiques de combat de manière qu'elles marquent une rupture avec les phases d'énigmes, tout en restant cohérentes avec l'univers du jeu. Elles ont un rythme plus soutenu et des sonorités plus marquées pour accentuer les affrontements, surtout pendant le combat final. La musique accompagne les actions des joueurs tout au long des combats. Comme le reste de l'environnement sonore, j'ai écrit ces musiques pour qu'elles s'adaptent à la durée et à l'intensité des combats.

3. Énigmes :

Enfin j'ai commencé à écrire des morceaux pour les énigmes. Elles doivent accompagner la réflexion des joueurs pendant les salles d'énigmes. Elles reposeront sur des rythmes lents et répétitifs et des sonorités discrètes et atmosphériques, pour créer une ambiance qui permettra de bien se concentrer. La musique participe donc à l'immersion des joueurs dans les énigmes et renforce la cohérence du jeu.

Ensuite, le sound design fait également partie de mon rôle. Je crée et j'intègre les sons liés aux actions des joueurs, aux interactions avec l'environnement, aux énigmes et aux pouvoirs spécifiques. Ces sons servent à rendre le jeu plus vivant et à fournir des retours auditifs clairs et utiles au gameplay. Ils aident les joueurs à comprendre ce qu'il se passe et à anticiper certains événements. Chaque son est conçu pour être lisible, cohérent avec l'univers du jeu et adapté à son contexte d'utilisation.
En faisant des recherches, j'ai découvert la plateforme itch.io. Il y a un espace de diffusion pour les jeux indépendants et de nombreux packs de sons qui sont disponibles gratuitement et l'achat. Ces packs sont libres de droit ce qui permet de les utiliser légalement dans le jeu sans contrainte de licence. Ils offrent des effets sonores variés et de qualité qui sont adaptés pour ce projet.

J'ai créé le site internet Echoes of Lights comme un espace central qui rassemble toutes les informations sur le jeu. Le design est sombre et immersif pour représenter l'univers du jeu et ses thèmes autour de la lumière et de l'ombre. Dès la première page, on peut découvrir une présentation claire du jeu, de son concept et de son gameplay en coopération.
Le site contient aussi une section sur le lore, qui explique l'univers d'Equinoxa, son histoire, ses personnages et les enjeux du scénario. Les informations sont organisées simplement pour que ce soit fluide et compréhensible.
Une autre partie du site se concentre sur les aspects techniques du projet. Elle présente les outils utilisés, les ressources employées, les liens vers le dépôt GitHub et le rapport d'avancement pour suivre le développement du jeu.
Enfin, une page dédiée à l'équipe et présente les membres du projet et leurs rôles.

4. Prochaines missions :

- Rédaction des musiques avec les idées présentées précédemment
- Implémenter les musiques dans le jeu
- Quand le jeu sera fini : implémenter les sound designs

Ressources :

Voici les différentes ressources dont j'ai eu besoin pour mon travail :

- https://musescore.org/fr
- https://tommusic.itch.io/free-fantasy-200-sfx-pack
- https://leohpaz.itch.io/minifantasy-dungeon-sfx-pack

3. Récit de mise en œuvre du projet

3.1. Challenges

Pendant la réalisation de cette première étape, nous avons rencontré différents défis sur le plan de l'organisation ainsi que sur le plan technique.
Tout d'abord, le fait que les membres du groupe ne soient pas tous dans la même classe a rendu la communication et la coordination plus complexes. Cette contrainte a demandé la mise en place d'outils adaptés pour maintenir un suivi efficace du projet. Nous avons utilisé Discord pour les échanges plus réguliers et Notion comme espace de travail collaboratif, ce qui nous a permis de centraliser les informations, les tâches et l'avancement du projet. Ces outils nous ont permis de maintenir un projet structuré.
Voir annexe 3.1 pour des images de Notion et Discord.

Un autre défi majeur a été la prise en main de la bibliothèque Pygame. Pour la majorité d'entre nous, il s'agissait de la première expérience dans le développement d'un jeu vidéo, en particulier à l'aide de cet outil. Nous avons donc appris à maîtriser la librairie, surtout la gestion de l'affichage et des événements. Cette phase d'apprentissage a demandé un investissement important en temps et en recherches personnelles.

Le développement du jeu en lui-même a aussi soulevé des difficultés techniques, comme la gestion des différentes boucles du jeu qui sont : la boucle principale, la gestion des événements et la mise à jour de l'état du jeu, ce qui a nécessité une réflexion approfondie afin d'assurer un fonctionnement fluide et cohérent, en particulier dans le cadre d'un jeu multijoueur en réseau pair-à-pair (P2P). Il a fallu veiller à ce que les différentes parties du programme interagissent correctement sans provoquer de conflits ou de comportements non attendus.

Finalement, la fusion des différents codes développés par chaque membre du groupe a constitué une tâche particulièrement complexe. En début de projet, nous avons choisi de travailler de manière relativement indépendante sur nos tâches respectives, en fonction des rôles attribués. Cette méthode a permis d'avancer plus rapidement mais a rendu la l'intégration plus complexe. Nous avons dû adapter nos codes, harmoniser les structures et résoudre les incompatibilités afin de garantir un fonctionnement global cohérent du jeu.

3.2. Réussites / succès

Malgré les difficultés rencontrées, nous avons tout de même eu plusieurs réussites significatives.

Nous avons réussi à mettre en place une organisation efficace au sein du groupe. Grâce aux outils de communication (discord) et au tableau de gestion des tâches (notion), chacun a pu assurer son rôle et ses responsabilités. Cette organisation nous a permis d'assurer une progression continue du projet.

Ce projet nous a également permis d'apprendre certaines compétences sur des aspects techniques. En faisant face à de nouveaux problèmes, tels que le développement avec pygame ou la mise en place d'un multijoueur en réseau, nous avons acquis de nouvelles connaissances et renforcé nos capacités en programmation. Ces apprentissages nous seront très utiles pour nos futurs projets.

Par ailleurs, ce travail de groupe nous a appris à collaborer sur un projet de programmation de plus grande ampleur. Nous avons fait preuve de rigueur, en particulier dans la documentation du code et l'adaptation de notre travail aux participations des autres membres de l'équipe.

Enfin, à ce stade du projet, nous avons réussi à poser des bases solides pour un jeu vidéo multijoueur en 2D isométrique de type dark fantasy. Les fondations techniques et organisationnelles sont alors en place, l'objectif est maintenant de poursuivre le développement notamment des différentes branches du monde ouvert et des mécaniques de gameplay.

4. Annexes

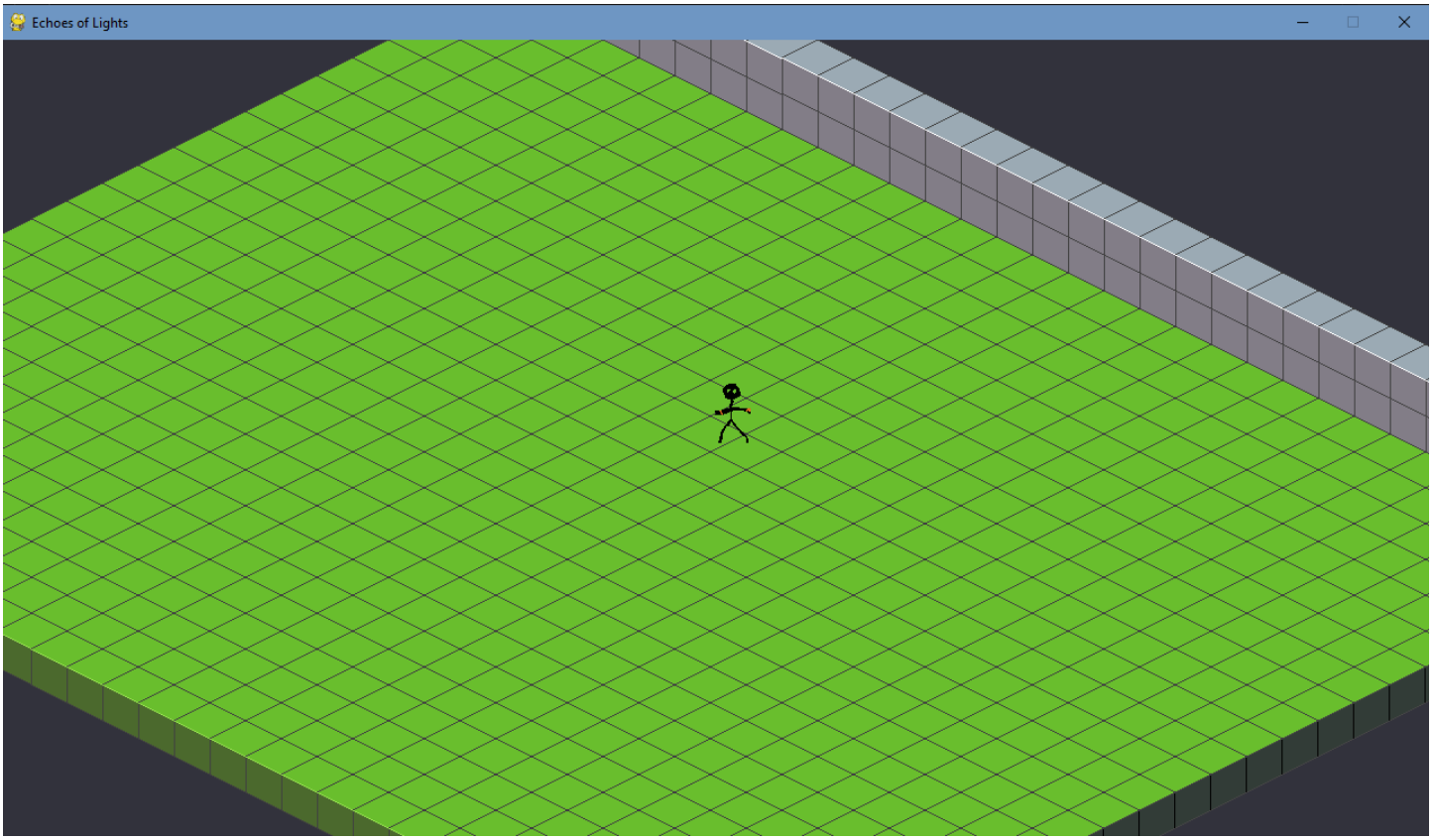
1.1. Tableau extrait du TSD avec estimations de l'avancée aux différentes soutenances

Task progress table			
Tâches	Soutenance 1	Soutenance 2	Sout. Finale
Pixel art	20%	50%	100%
Musique	0%	50%	100%
Animation	0%	50%	100%
sound design	0%	50%	100%
multijoueur	100%	100%	100%
gameplay	50%	90-100%	100%
map	20%	50%	100%
déroulement du jeu	80%	100%	100%
affichage jeu	80%	100%	100%
developement des joueurs	70%	90%	100%
affichage menu	30%	100%	100%
AI	20-30%	70%	100%
Veillez indiquer le pourcentage d'avancement prévu à la date de la soutenance			

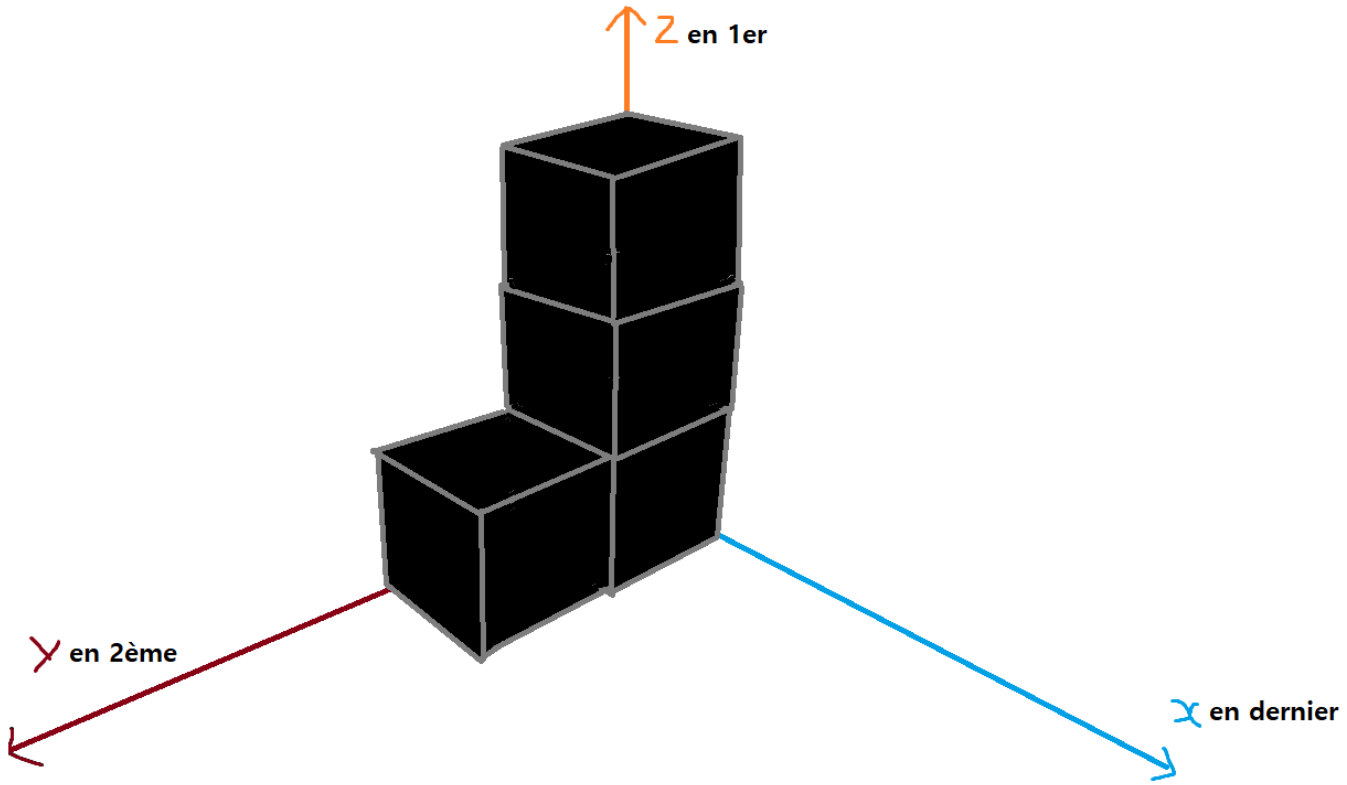
2.1.1. Une offre ou réponse SDP


```
{"sdp": "v=0\r\no=- 3970739909 3970739909 IN IP4 0.0.0.0\r\ns=-\r\nt=0 0\r\na=group:BUNDL
E 0\r\na=msid-semantic:WMS *\r\nm=application 59947 UDP/DTLS/SCTP webrtc-datachannel\r\nc
=IN IP4 10.2.0.2\r\na=mid:0\r\na=sctp-port:5000\r\na=max-message-size:65536\r\na=candidat
e:d6ee6c8481721c8f410c5e2bcc4334f8 1 udp 2130706431 10.2.0.2 59947 typ host\r\na=candidat
e:ae5c2359578edcf27f2d217b587052e6 1 udp 2130706431 192.168.56.1 59948 typ host\r\na=cand
idate:a7f0c3a094b0dc7d1343263db553d60c 1 udp 2130706431 192.168.1.6 59949 typ host\r\na=c
andidate:36780b3c8d48149a320c24d2c2fb881c 1 udp 1694498815 190.2.153.209 61566 typ srflx
raddr 10.2.0.2 rport 59947\r\na=end-of-candidates\r\na=ice-ufrag:s9co\r\na=ice-pwd:lDFpIF
c6F8YpLVNl3cb2KD\r\na=fingerprint:sha-256 AC:4E:E2:7C:3D:82:EB:2D:3F:40:12:AE:62:00:20:6
C:93:73:CB:1C:C8:69:64:3B:6F:11:C0:6D:95:4B:DF:96\r\na=fingerprint:sha-384 89:35:52:36:B
5:33:5D:68:3D:6C:28:F5:70:0C:50:15:6E:36:34:AA:C1:9F:37:CE:38:1F:03:A9:38:6B:C7:45:70:83:
10:92:0B:8F:77:41:B6:FD:89:CD:F3:80:38:26\r\na=fingerprint:sha-512 98:C2:AD:68:43:73:D5:C
0:AA:97:07:31:23:22:B2:E1:97:BB:3E:4E:07:A6:C7:66:85:C7:EE:1C:91:A8:AB:93:A7:48:BA:21:ED:
95:71:C4:2A:20:54:AB:FB:15:1B:41:99:DB:80:1C:A4:08:6C:3B:0E:5C:A1:79:2E:3F:C1:E7\r\na=set
up:active\r\n", "type": "answer"}
```

2.2.6. Capture d’écran de l’affichage du jeu



2.2.7. Schéma expliquant l’ordre d’affichage des tuiles isométriques



2.4.2. Exemple d’énigme (“La Séquence”) :

La Séquence :

Aeden et Lyra sont dans 2 salles séparées. La salle d’Aeden contient un message invisible sur le sol et 4 boutons sur un mur, celle de Lyra contient 4 plaques de pression et 1 cube d’ombre. Quand Aeden appuie sur un bouton, une plaque de pression s’illumine. Aeden fait apparaître le message au sol avec son flash de lumière : c’est une séquence pour les boutons. Avec son faisceau de lumière, il active les boutons dans le bon ordre pour illuminer les bonnes plaques de pression : ainsi, Lyra peut déplacer le cube et le poser sur la bonne plaque. Si une erreur est faite, il faut recommencer. Attention : 1 bouton ne correspond pas à 1 plaque ! Mais une plaque s’illumine seulement si le bon bouton est enclenché. Sinon, une lumière clignotante montre à Aeden qu’il s’est trompé dans la séquence.

3.1 Capture d’écran Discord et Notion respectivement :