Document de la phase d'analyse du projet tutoré

Sommaire:

Marcelin Ray - Ilyès Lounici - Cyprian Chailan - Geoffrey Gros
Vincent Thomas
BUT 3 | RA-IL 1

Liste de fonctionnalités

Moteur graphique:

- Raycasting
- BSP
- gestion de texture

Gestion du jeu :

- Collision
- création du labyrinthe
- mise à jour des graphismes

Ennemie:

- behavior gestion de comportement (fuite, poursuite, patrouille)
- prise de dégâts
- déplacement via Pathfinding
- perception de l'environnement

PNJ:

- gestion de quête
- discussion intelligent avec les joueurs

Système multijoueur:

- peer to peer
- transmission des coordonnées des joueurs et des monstres
- récupération des coordonnées
- connexion et déconnexion de joueurs

Joueur:

- avancer
- attaquer

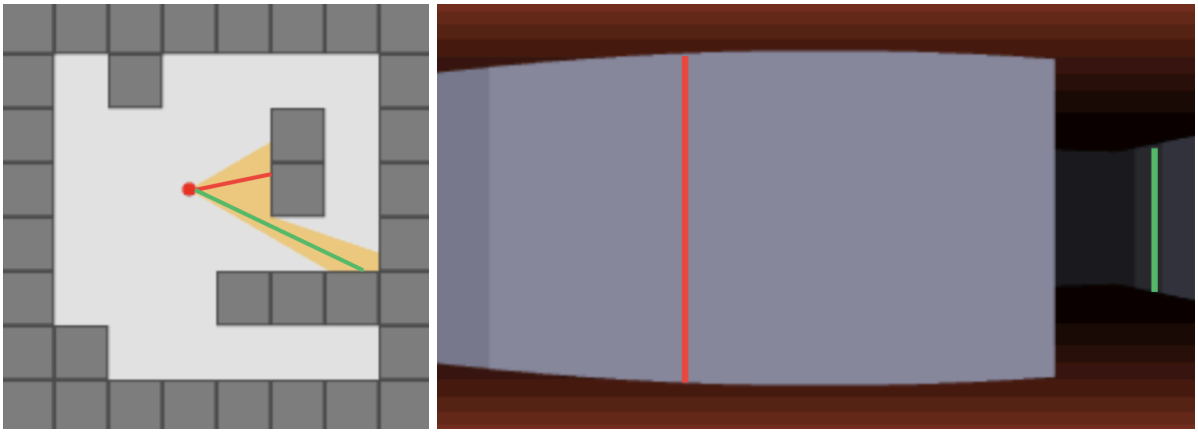
Moteurs

Nous allons réaliser 2 moteurs “3D” différents durant le projet.

Raycasting

Le principe du raycasting est de projeter un environnement 2D en pseudo 3D.

Exemple:



https://en.wikipedia.org/wiki/Ray_casting

Le point rouge représente notre joueur, et les carrés gris nos murs.

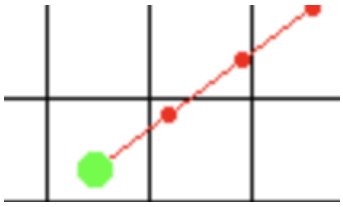
Pour ce faire, on doit dans un premier temps définir un champ de vision pour notre personnage, ici représenté en jaune pour une meilleur compréhension.
Ensuite, on va balayer chaque angle de ce champ de vision, afin de récupérer la distance entre le joueur et le mur le plus proche.

Plus la distance est courte, plus le trait sera représenté de manière haute et claire, et plus la distance sera longue, plus le trait sera dessiné de manière petite et sombre, créant ainsi un rendu pseudo 3D. 2 rayons sont représentés en couleur afin de mieux comprendre le passage de la 2D en 3D.

Afin de mesurer la distance (entre le joueur et le mur le plus proche, à un angle donné), il y a plusieurs manières de se faire:

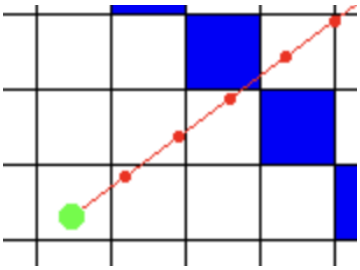
- la solution “évidente” serait de tracer la droite et d’avancer d’une distance fixe, puis comparer si l’on touche un des murs

Exemple imagé:



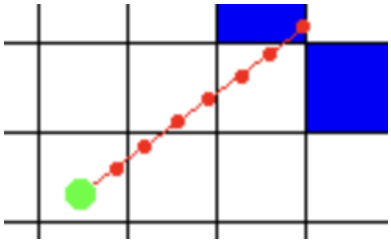
https://lodev.org/cgtutor/raycasting.html

Problème ? -> On peut très vite dépasser un mur, et ne pas détecter la collision …



https://lodev.org/cgtutor/raycasting.html

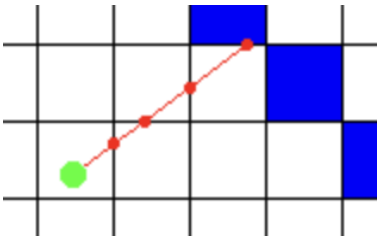
Pour pallier à cela, on pourrait imaginer de prendre une distance encore plus courte:



https://lodev.org/cgtutor/raycasting.html

Mais avec cela, l’optimisation devient catastrophique, les temps de calculs s’allongent énormément et le problème n’est même pas toujours réglé, on peut encore traverser un mur en passant sur un coin.

- la deuxième solution prend en compte le fait que notre environnement est normé par des “bloc”, en utilisant un algorithme nommé DDA (Digital Differential Analysis), on peut très rapidement balayer la diagonale sans rater de détail:



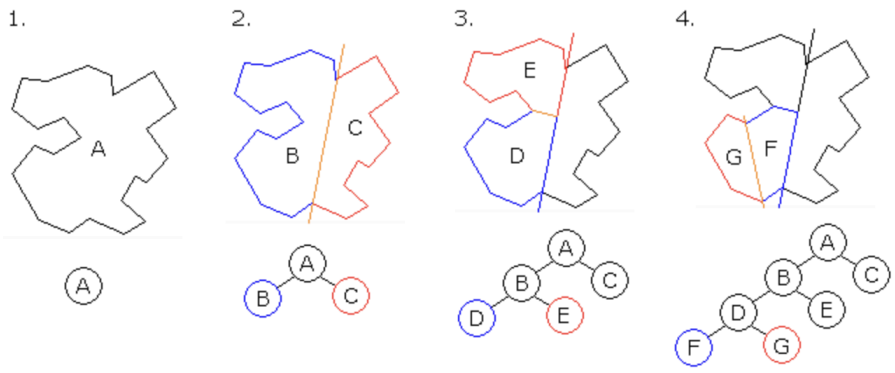
https://lodev.org/cgtutor/raycasting.html

Notre but sera ensuite d'ajouter des textures à ces murs.

Pour ajouter les textures sur les murs, si le rayon qui touche un mur proche, la colonne est grande et la texture apparaît agrandie ; s’il touche un mur lointain, la colonne est petite et la texture est réduite. La position exacte du choc sur le mur nous dit quelle bande verticale de l’image de texture afficher. On peut enfin assombrir avec la distance pour donner un peu de relief.

BSP - Binary Space Partitioning

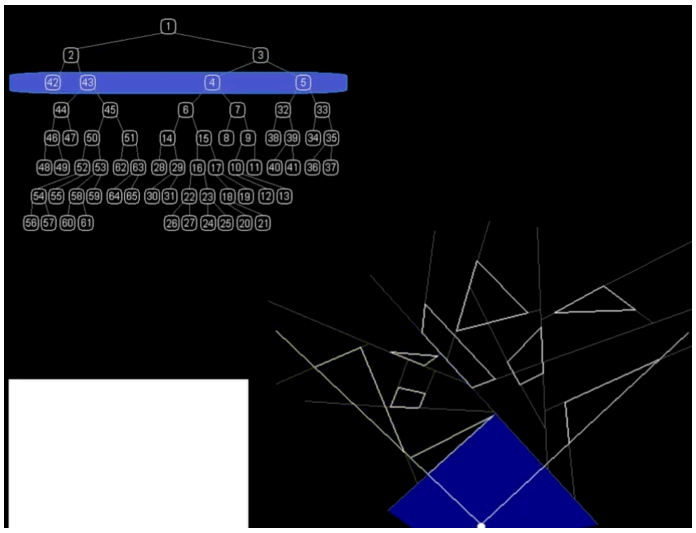
L'objectif du BSP est d'avoir un arbre, constitué de zone convexe de notre map trié.
On va découper notre espace en plusieurs espaces convexe, en disant lesquels sont devant et derrière.



https://en.wikipedia.org/wiki/Binary_space_partitioning

Ensuite, quand l'on veut rendre une image, on n'utilise plus la carte, on utilise seulement l'arbre:

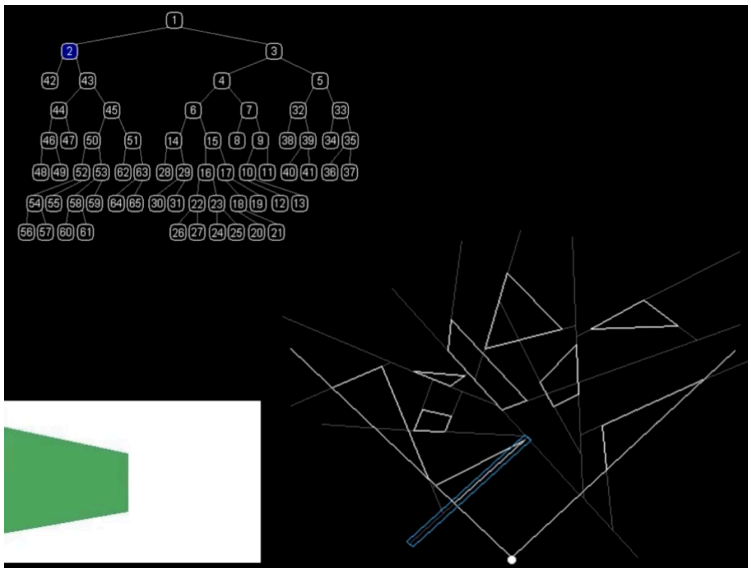
1er étape:
On parcourt l'arbre pour trouver où est le joueur. Pour ce faire, on regarde le premier nœud, si le joueur est à gauche du nœud, on va dans le sous arbre gauche, si il est à droite on va dans le sous arbre droit, et on continue récursivement jusqu'à atteindre la fin de l'arbre.



https://www.youtube.com/watch?v=yTRzfKh4Tg0

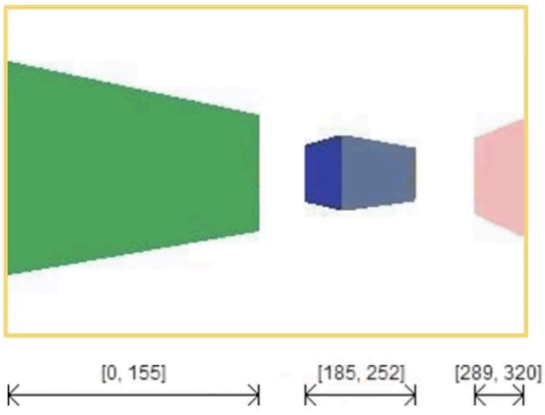
Dans ce schéma, on voit l'arbre BSP en haut à gauche, ainsi que Ici, on voit que le joueur se trouve dans l'espace 42, qui n'a plus de feuilles.

On aura donc une position dans l'arbre, et à partir de là, on va remonter l'arbre et afficher les murs visibles, un par un.



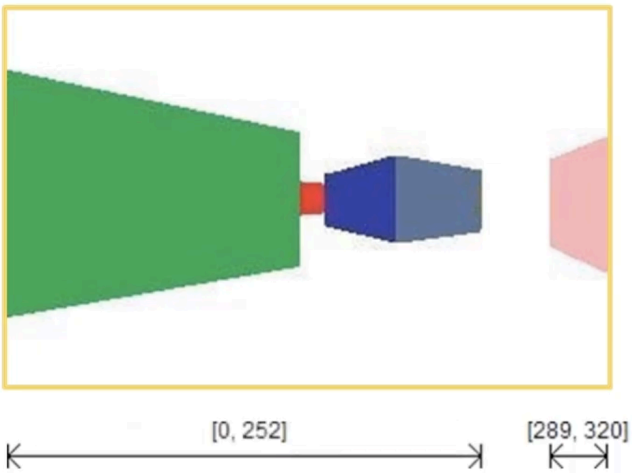
https://www.youtube.com/watch?v=yTRzfKh4Tg0

Le premier nœud ici est le 2, puis on remonte et parcourt récursivement jusqu'à la fin de l'arbre. Nous ne connaissons pas à cette étape l'algorithme précis pour le parcours efficace de l'arbre.



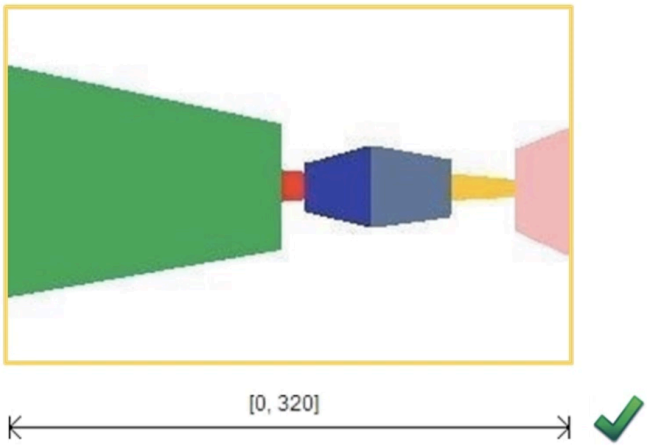
https://www.youtube.com/watch?v=yTRzfKh4Tg0

A chaque fois que l'on dessine un mur, on stock la taille de l'écran dessiné. Ainsi, si on dessine un mur derrière, on va ignorer les parties non visibles de ce mur.



https://www.youtube.com/watch?v=yTRzfKh4Tg0

Et quand on rajoute un mur, on écrase/fusionne les tailles stockées.



https://www.youtube.com/watch?v=yTRzfKh4Tg0

Ainsi quand l'écran est fini, on peut arrêter le parcours de l'arbre.

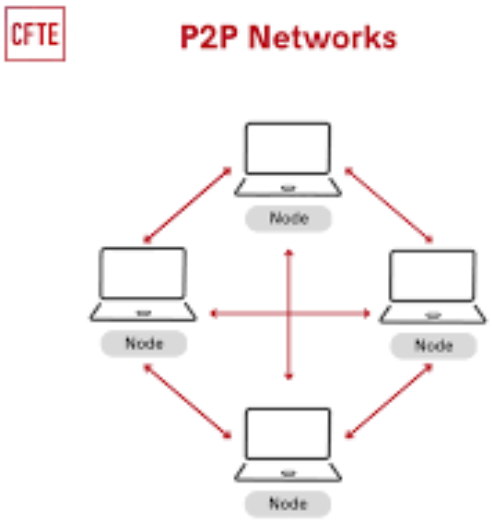
Système multijoueur

Pour une raison de temps et de recentrage du projet nous avons décidé de réaliser seulement le P2P. Il sera plus adapté à nos ressources car nous ne disposons pas de serveurs dédiés pour héberger le mode en ligne.

Le système pair-à-pair, aussi appelé peer-to-peer (P2P), est un mode de connexion où les joueurs communiquent directement entre eux, sans passer par un serveur central. Dans ce modèle, chaque joueur est à la fois client et serveur : il envoie et reçoit des informations aux autres participants afin de synchroniser la partie.

Lorsqu'un joueur effectue une action, comme se déplacer ou tirer, son jeu transmet directement l'information aux autres machines connectées. Chacun des ordinateurs reçoit alors ces données et met à jour sa propre version du jeu pour refléter ce qui vient de se passer. Ainsi, la cohérence du monde virtuel repose sur la bonne communication entre tous les joueurs.

Pour réaliser ces deux solutions, nous utiliserons les sockets de java afin d'assurer la transmission des données entre les clients pour P2P et les clients et le serveur. Pour entrer dans le réseau, nous utiliserons une API qui s'occupera de stocker les IP et port des différents host. Cela permettra de simplifier la connexion pour les différents tests à réaliser.

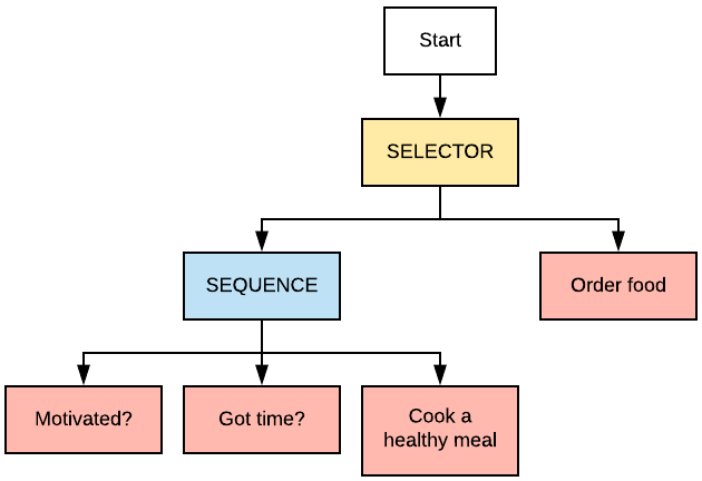


https://www.google.com/imgres?q=p2p%20image&imgurl=https%3A%2F%2Fblog.cfte.education%2Fwp-content%2Fuploads%2F2023%2F03%2FDiagram-of-P2P-Network-1024x1024.png

Monstre

Les différents monstres que nous souhaitons implémenter auront des comportements différents afin de varier. Ils pourront attaquer le joueur, patrouiller ou bien fuir le voyant. Un axe d'amélioration possible serait d'intégrer un arbre de comportement ou un automate afin de rendre l'ennemi plus autonome afin de les rendre plus durs à tuer.

Un arbre de comportement (ou behavior tree en anglais) est un modèle hiérarchique utilisé pour décrire, organiser et contrôler les comportements d'un agent. Ici, cet agent sera le monstre. Par exemple, si le joueur tire sur le monstre, le monstre pourrait fuir pour se cacher. Au contraire, s'il le montre ne fait pas tirer dessus et qu'il se trouve à portée d'attaquer, il pourrait attaquer le joueur.



https://www.google.com/imgres?q=behavior%20tree&imgurl=https%3A%2F%2Fblog.zhaytam.com%2Fimg%2FBehaviorTreeExample.png

Avec cet exemple, start est le point de départ. Le SELECTOR représente un ou. Soit on commande, soit on fait Sequence. Sequence est le et logique. Donc si on est motivé et qu'on a le temps alors on cuisine un repas simple sinon on commande.

Un système de perception pourrait permettre d'affiner cette arbre, si le monstre se rend compte que le joueur le voit, il pourrait fuir directement. A l'inverse si le monstre voit qu'il n'est pas vu alors, il pourrait l'attaque par surprise.

Pour la patrouille, le monstre pourrait se déplacer en utilisant l'algorithme de A* afin de l'implémenter rapidement. L'implémentation de RRT* pourrait être ensuite réalisée afin d'être plus efficace et explorer une solution plus complexe.

RRT*

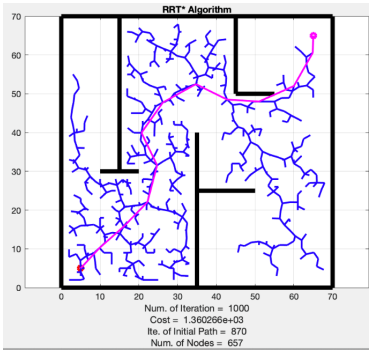
L'algorithme RRT* (Rapidly-exploring Random Tree) est un planificateur à échantillonnage qui construit un arbre de positions sans heurter d'obstacles à partir d'un point de départ, puis améliore localement cet arbre pour réduire le coût (longueur/temps/énergie) du chemin. Contrairement à RRT (qui rend vite un chemin), RRT* tend vers le chemin optimal à mesure que l'on ajoute des échantillons (optimalité asymptotique)

Deux ajouts clés par rapport à RRT :

- chaque nœud stocke un coût depuis le départ et choisit, parmi les voisins, le parent le moins coûteux ;
- le rewiring : si passer par le nouveau nœud diminue le coût d'un voisin, on recâble ce voisin.
  → chemins plus droits et courts que RRT, au prix de calculs supplémentaires.

Comment ça marche (cycle)

1. Init : l'arbre contient seulement la position de départ de l'ennemi.
2. Échantillon x_rand : on tire un point aléatoire (parfois biaisé vers la cible).
3. Extension steer : on va du nœud existant le plus proche vers x_rand d'un pas max η → x_new.
4. Collision : on ignore x_new si le segment coupe un obstacle.
5. Voisinage : on récupère les nœuds proches de x_new dans un rayon r_n (il décroit avec la taille de l'arbre).
6. Meilleur parent : on relie x_new au voisin qui minimise coût(parent) + distance(parent, x_new).
7. Rewiring : pour chaque voisin, si passer par x_new réduit son coût, on recâble ce voisin (et on met à jour les coûts de ses descendants).
8. But : si x_new tombe dans la zone objectif, on mémorise le meilleur chemin courant.
9. Répéter jusqu'au budget d'itérations/temps. Plus on itère, plus le chemin devient court et lisse (après lissage).



https://www.researchgate.net/publication/366866319_An_Integrated_RRTSMART-A_Algorithm_for_solving_the_Global_Path_Planning_Problem_in_a_Static_Environment

A*

L'algorithme A* sert à trouver le chemin le plus court entre un point de départ et un point d'arrivée dans un graphe ou une grille. Ici cette grille sera notre labyrinthe. L'heuristique s'est l' estimation du coût total pour atteindre l'objectif.

Algorithme :

```
Structure nœud = {
    x, y: Entier
    cout, heuristique: Entier
}
```

```
Fonction compareParHeuristique(n1:Nœud, n2:Nœud)
    si n1.heuristique < n2.heuristique
        retourner 1
    ou si n1.heuristique == n2.heuristique
        retourner 0
    sinon
        retourner -1
```

```
Fonction cheminPlusCourt(g:Graphe, objectif:Nœud, depart:Nœud)
    closedList = File()   // file des noeuds parcours
    openList = FilePrioritaire(comparateur = compareParHeuristique) // file des
noeuds à explorer
    openList.ajouter(depart) // on ajoute le noeuds de départ

    tant que openList n'est pas vide
        u = openList.defiler()   // prend le nœud avec le plus petit f(n)

        si u.x == objectif.x et u.y == objectif.y
            reconstituerChemin(u) // noeud d'objectif atteint
            terminer le programme

        pour chaque voisin v de u dans g
            si v non dans closedList ou openList avec coût inférieur
                v.cout = u.cout + 1
      // coût réel + estimation jusqu'à l'objectif
                v.heuristique = v.cout + distance([v.x, v.y], [objectif.x,
objectif.y])
                openList.ajouter(v)

        closedList.ajouter(u)
```

PNJ

Certains PNJ auront des rôles narratifs, servant à transmettre des informations sur le contexte ou les mystères du labyrinthe. D'autres proposeront des petites quêtes secondaires permettant au joueur de découvrir des récompenses inédites. Ces quêtes peuvent aller de la recherche d'objets rares à l'accomplissement de tâches simples, mais elles participent toutes à rendre l'expérience plus vivante et variée.

Nous utiliserons une API comme Gemini ou Mistral afin de faire des dialogues plus poussés et réalistes en utilisant des prompts afin qu'elle se prenne pour le personne qu'elle est censée être.

Risques du projet

Catégorie	Tâche	Niveau 1-10
Raycasting	réalisation basique avec swing et jpanel	5
	collision avec DDA	2
	affichage texture	4
		à peu près 4,5
BSP	Afficher un mur (gestion clipping, etc ..)	10
BSP	Afficher un mur texturé	6
BSP	Découper l'espace	10
BSP	Painter's algorithm	7
		à peu près 9
Comportement ennemi	behaviors tree	7
	ajout de perception visuel	4
deplacement ennemi	A*	1
	RRT*	5
multijoueur	synchronisation P2P	8
PNJ	discussion avec prompt	9

Planning

Itération 1

Impératif

- Base commune : repo, branches, squelette Java, boucle de jeu minimale.
- Prototype raycasting (labyrinthe pseudo-3D, colonnes sans textures).
- Prototype P2P minimal : connexion 2 clients, échange positions via sockets.

Optionnel

- Assemblage prototypes : contrôler le joueur localement + positions reçues du P2P.
- Premiers tests A* sur grille simple.

Livrables

- Repo initial
- Exécutable minimal affichant raycast + échange de positions entre 2 clients.

Itération 2

Objectifs impératifs

1. A* : Implémentation complète A* sur la grille du labyrinthe (chemin pour ennemis).
    - Tâches : structure Node, open/closed lists, heuristique Manhattan/Euclidienne testable, reconstruction de chemin.
    - Tests : cas simples (obstacle direct), cas maze, performance sur 50×50.
2. BSP : début de construction
    - Tâches : parser des segments/murs, choisir séparateur, découpe des segments traversants, création récursive des nœuds (front/back).
    - Livrable minimal : exporter l'arbre BSP d'un niveau simple et visualisation texte ou debug draw (pos/partition).
3. Intégration raycasting + A*
    - Tâches : permettre à un ennemi de suivre un chemin A* et se déplacer sur la carte (déplacement discret ou interpolé).

Objectifs optionnels

- Collision fine pour ennemis suivant le chemin (éviter d'entrer dans les murs).
- Test simple de pathfinding multi-ennemis.

Itération 3

Objectifs impératifs

1. BSP : rendu
    - Tâches : implémenter parcours d'affichage BSP.
2. Textures pour raycasting
    - Tâches : mapping UV basique pour murs, colonne par colonne -> sélectionner slice de texture selon collision.
    - Livrable : niveau texturé en raycasting.
3. Collision joueur améliorée
    - Tâches : collision continue (cercle/ellipse) avec murs, prévention de « clip through corners ».

Objectifs optionnels

- Support pour plusieurs textures par mur (orientation).

Itération 4

Objectifs impératifs

1. Behavior Trees (BT) pour ennemis
    - Tâches : ajouter structure BT (Selector, Sequence, Leaf nodes (conditions/actions)). Implémenter comportements de base : patrouille, poursuite, fuite, prise de dégâts/réaction.
    - Connexions : perception -> BT (vision/son/line-of-sight).
2. Perception & FSM sensorielles
    - Tâches : line-of-sight (raycast checks), hearing radius (événements), memory (cooldown/oubli).
3. Déplacement ennemi via pathfinding
    - Tâches : intégration A* avec BT actions (goto waypoint, chase player). Smooth movement and steering (avoidance basique entre ennemis).

Objectifs optionnels

- Implémenter RRT* prototype pour comparaison avec A* sur maps ouvertes.

Itération 5

Objectifs impératifs

1. PNJ : gestion de quête et dialogues
    - Tâches : système de quêtes (objectifs simples : récupérer objet, parler à PNJ, atteindre zone), état de quête (inactive/active/complétée), récompense simple.
    - Intégrer API de génération de dialogues. Préparer prompts/formats pour Mistral/Gemini si usage ultérieur.
2. Multijoueur P2P : améliorer stabilité & synchronisation
    - Tâches : robustifier échanges P2P :
        - tolérance à la latence (interpolation/extrapolation des positions),
        - gestion join/leave,

        - envoi périodique snapshots des entités (players & monsters),
        - simple autorité distribuée (*qui* calcule *quoi*).
    - Protocoles : définir messages (JOIN, LEAVE, SNAPSHOT, ACTION).
3. Tests réseau
    - Tâches : test 3–4 clients (local / LAN), simulations latence (si possible).

Objectifs optionnels

- Chiffrement basique / validation anti-spoof (checksums).
- Reconciliation authoritative fallback (un joueur/hôte authoritative).

Itération 6

Objectifs impératifs

1. Ajout d'ennemis + équilibrage
    - Ajouter 2 variantes d'ennemis (force/vitesse/IA mix) et tests d'équilibrage.
2. Tests finaux & packaging
    - Tests d'acceptation multi-plateformes (si applicable), construire un binaire/jar exécutable, README & manuel d'installation.

Objectifs optionnels

- Implémentation RRT* pour certains ennemis (comparaison avec A*).
- Ajout PNJ avec dialogues alimentés par vrai modèle (si accès API autorisé).
- Implémenter génération procédurale de labyrinthe (algorithme simple : DFS maze, ou cellular automata).