Projet tutoré

Moteur Doom-like basé sur le raycasting

Raycasting, génération procédurale, IA et multijoueur en Java

Marcelin Ray, Ilyès Lounici, Cyprian Chailan, Geoffrey Gros
Tuteur : Vincent Thomas

BUT 3 – RA-IL 1

Plan de la présentation

1. Contexte
2. Objectifs et périmètre
3. Étude de l’existant
4. Étude technique
5. Technologies utilisées
6. Conclusion & perspectives
7. Questions

Contexte du projet

Orientation du projet

Développement d’un moteur graphique simplifié en java.

Objectif

Explorer et expérimenter des briques techniques clés du développement de jeux vidéo.

Axes techniques étudiés

Rendu graphique ; IA ennemis/PNJ ; multijoueur

Objectifs et périmètre

- **Prototype technique**
  Livrable limité à la démonstration des concepts.
- **Moteur graphique raycasting + BSP**
  Simuler un monde 3D à partir d'une grille 2D optimisée.
- **Génération procédurale**
  Créez des labyrinthes parfaits via Prim et Kruskal.
- **IA ennemis & PNJ**
  Algorithmes de parcours et comportements conditionnels.
- **Multijoueur**
  Deux modèles : client-serveur et pair-à-pair.

Références historiques

Wolfenstein 3D (1992) et Doom (1993)

Wolfenstein 3D

Premier FPS « 2,5D » grand public

Carte = simple grille 2D (cases pleines / vides)

Raycasting basique pour projeter les murs

Doom

Moteur plus riche (id Tech 1)

Secteurs avec hauteurs, lumières, sols/plafonds texturés

Optimisation du rendu grâce aux arbres BSP

Ces moteurs introduisent les deux idées que nous réutilisons : simuler la 3D à partir d'un plan 2D (raycasting) et structurer la carte pour accélérer le rendu (BSP)

Projets open-source : Exemples

Mochadoom

- Langage : Java, proche de notre environnement de développement
- Montre comment séparer rendu, logique de jeu et chargement des ressources
- Référence possible pour : structure du projet, boucle de jeu, gestion des textures

https://github.com/AXDOOMER/mochadoom

Make Your Own Doom

- Explication pas à pas de la création d'un moteur 3D basé sur BSP
- Met en avant les structures de données : cartes, secteurs, noeuds BSP
- Réference pour : comprehension du BSP, pipeline de rendu, optimisation

https://youtube.com/playlist?list=PLi77irUVkDasPdenthU6O40xLKDvaSYtO&si=-qJiptc1lon4jc9Q

Analyse comparative

Critère	Wolfenstein 3D	Doom	Notre projet
Type de rendu	Raycasting	Raycasting + BSP	Raycasting + BSP
Génération procédurale	Non	Non	Oui (labyrinthe aléatoire)
Multijoueur	Non	Partiel (LAN)	Oui (client-serveur & P2P)
IA ennemis	Très simple	Moyenne	Évolutive (basique)
Objectif	Jeu complet	Jeu complet	Prototype technique

Notre projet s’inspire des bases historiques tout en intégrant la génération procédurale et le multijoueur.

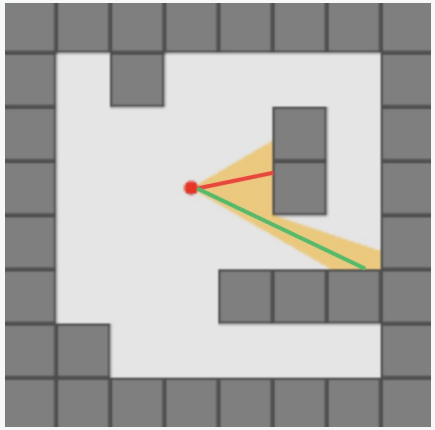
Raycasting

Technique qui simule la 3D depuis un monde 2D en traçant des rayons.

BSP (Binary Space Partitioning)

Organise l’espace pour savoir rapidement quelles surfaces afficher, réduisant les calculs.

Raycasting

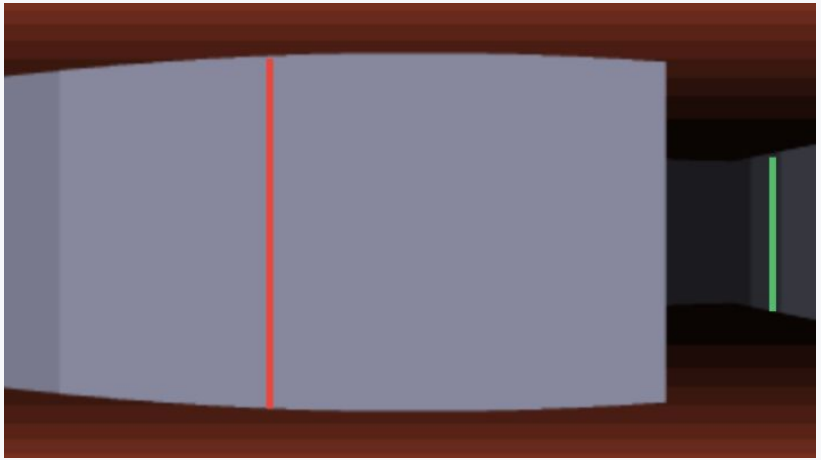


Scanner l'environnement:

Sur chaque angle, on mesure la distance entre le personnage et les murs.

Afficher les distances:

Pour chaque distance, on va dessiner un trait. Plus la distance est loin, plus le trait sera fin/sombre, générant une impression de 3D.



Binary space partitioning (BSP)

Principe

À cette étape du projet, le fonctionnement du BSP est assez confus pour nous, donc il ne sera pas détaillé ici.

L'idée principale est de diviser l'espace afin de réaliser un arbre avec les murs de la map.

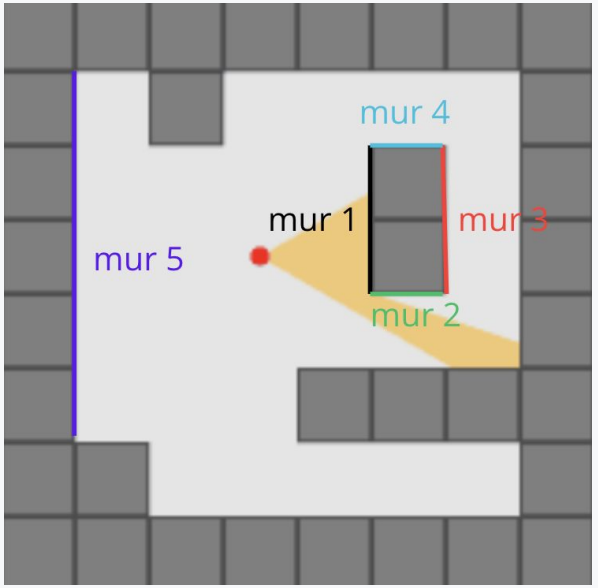
Ceci permet de filtrer les murs les plus susceptibles d'être rendu à un emplacement donné sur la map.

Limitation du Raycasting

En fonction de la taille de la map, le Raycasting est très énergivore.

Imaginons que la map est une liste de murs ->

Pour faire un rendu, quand ont scan une distance, on doit vérifier si le point est sur l'un de tous nos murs

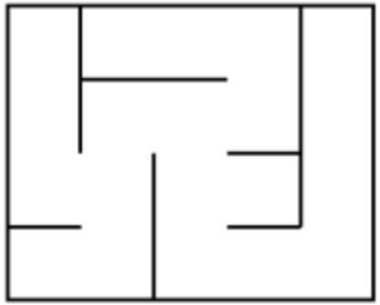


Optimisation avec l'arbre du BSP

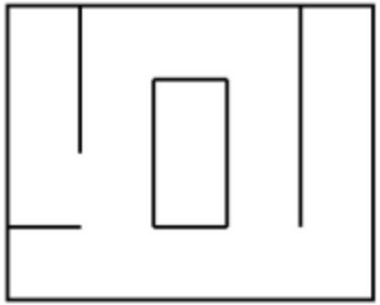
Grâce à l'arbre, on peut donc avoir la liste des murs les plus probable d'être affiché ->

On économise énormément de calcul en ignorant la vérification de position sur les murs inutiles.

Génération procédurale



Labyrinthe « parfait »



Labyrinthe « imparfait »

Labyrinthes parfaits

Tous les chemins sont connectés et il n'existe qu'un chemin entre deux cellules données.

Algorithmes :

- Prim
- Kruskal

Intelligence artificielle

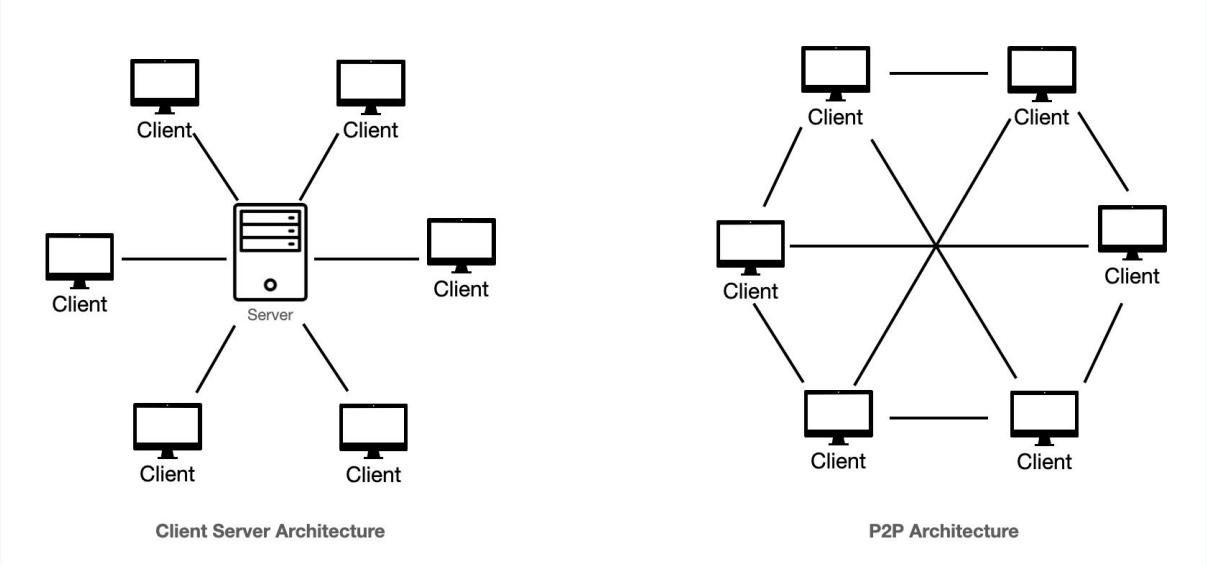
Ennemis

Parcours de graphes (BFS, Dijkstra) pour se déplacer.
Comportements : patrouille, poursuite, fuite.
Les ennemis réagissent aux actions du joueur, adaptant leur stratégie.

PNJ & conversations

Des modèles de langage (ex : Mistral) permettent de générer des dialogues dynamiques.
Le joueur peut demander son chemin ou lancer des quêtes générées par l'IA.

Système multijoueur



Client-serveur : un serveur central valide les actions et diffuse l'état du jeu.

Pair-à-pair : chaque client diffuse directement ses actions aux autres.

Technologies utilisées

Domaine	Technologie	Rôle
Langage	Java	Développement principal
Graphisme	Swing	Rendu 2D / pseudo-3D et entrées utilisateur
Réseau	Sockets Java	Communication client-serveur & P2P
IA	BFS & Dijkstra	Déplacement et comportements ennemis
Génération	Prim & Kruskal	Création automatique de niveaux
Architecture	MVC & Threads	Séparation claire des modules
Chatbot	API distante	Gestion des dialogues,

Conclusion & perspectives

Bilan du prototype

- Moteur de rendu Doom-like
- Génération procédurale de labyrinthes
- IA ennemis et PNJ
- Multijoueur client-serveur et P2P

Apports pédagogiques

Approfondissement des moteurs graphiques, de la programmation réseau et des algorithmes.

Perspectives

Améliorer le rendu et le level design, enrichir les comportements IA et ajouter des mécaniques de jeu avancées.

Questions ?

Échanges et remarques bienvenus !