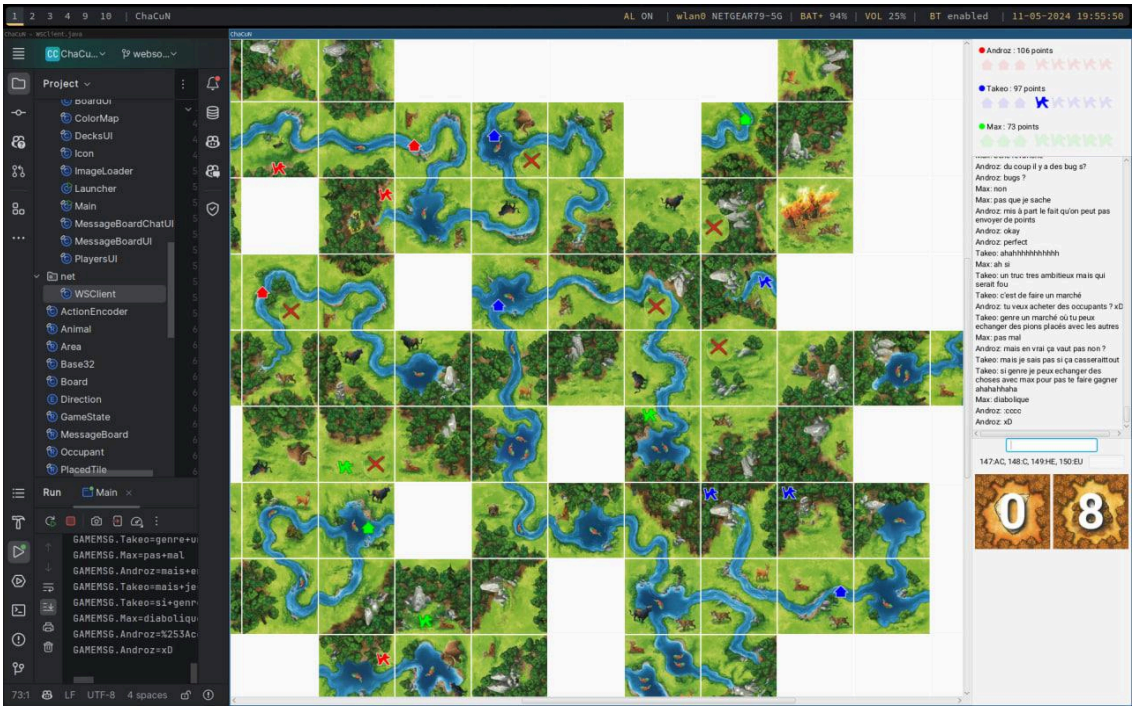
# ChaCuN Ensemble (ChaCuNE)

Ce projet est une version électronique du jeu Chasseurs et cueilleurs au Néolithique, abrégé ChaCuN, pour le cours CS-108 à l'EPFL. L'énoncé se trouve ici.



Les fonctionnalités et API décrites ci-dessous correspondent à des rendus bonus. Notre version améliorée ChaCuNE permet de jouer à plusieurs, d'échanger des messages directement dans le jeu, et contient plusieurs améliorations graphiques.

L'objectif de cette extension n'est pas d'ajouter de nouvelles règles, ou des fonctionnalités fantaisistes mais de rendre d'un point de vue général les parties de ChaCuN plus agréables.

## Auteurs

Le développement de cette extension est assez lourd. Nous avons :

- développé notre propre serveur implémentant le protocole WebSocket suivant la norme RFC 6455 sans librairie externe.
- écrit un fichier Docker pour le serveur permettant de le déployer sur un serveur.
- rédigé une documentation complète de l'API et du fonctionnement du serveur se trouvant à cette URL.
- implémenté cette API dans notre client ChaCuN en prenant garde à ne briser aucune règle d'encapsulation, et en préservant l'immuabilité de toutes les classes qui l'étaient dans la version originale (ce qui n'est pas évident étant donnée la réactivité des composants, nécessaire dans le mode multijoueur).
- créé les fichiers nécessaires pour construire les fichiers jar du serveur **et** du client avec Gradle.

Nous nous sommes donc associés à deux groupes pour réaliser cette extension (groupe de Maxence Espagnet, Balthazar Baillat et groupe de Valerio De Santis, Simon Lefort) comme autorisé par M. Schinz.

## Démarrer le client

Le client de ChaCuNE est exclusivement multijoueur.

Pour lancer une partie, il faut préciser :

- un paramètre nommé **player**, le nom du joueur local
- un paramètre nommé **game**, le nom de la partie souhaitée

Comme il est souvent pratique de tester en local et que le serveur n'autorise pas deux joueurs avec le même nom à rejoindre une même partie, il est possible d'inclure le flag nommé **debug** (un paramètre non nommé, donc) pour ajouter aléatoirement deux chiffres à côté du pseudo passé en paramètre et éviter tout conflit de pseudo lors de tests.

**Une table récapitulative des modifications apportées :**

| | **ChaCuN (original)** | **ChaCuNE ✨** |
|---|---|---|
| **Mode multijoueur** | ❌ (local) | ✅ (serveur de jeu) |
| **Messages de chats entre joueur durant la partie** | ❌ (via une app externe) | ✅ (intégrés) |
| **Mise en évidence de la dernière tuile posée (utile pour les gros plateaux multijoueur)** | ❌ | ✅ |
| **Retours sonores** | ❌ | ✅ (lors de la pose d'une tuile, fermeture d'une forêt menhir, gain de points) |
| **Anti-cheat** | 🟠 (N/A) | ✅ (validation des actions côté serveur, même un client modifié ne pourrait pas permettre la triche) |
| **Gestion de potentielles déconnexions de joueurs en cours de partie** | 🟠 (N/A) | ✅ (réinitialisation du plateau, affichage d'un message de notification pour les joueurs restants) |
| **Interface adaptative en fonction de la possibilité de jouer ou non** | 🟠 (N/A) | ✅ (désactivation de l'entrée d'actions, de la pose de tuiles, d'occupants, màj de la frange) |

## Rapport technique - Client ChaCuNE

La base du client est la même que dans le jeu original. Voilà une liste non exhaustive des modifications principales.

### Classe WSClient

Cette classe représente le client WebSocket.

Elle permet de se connecter au serveur et est responsable de l'analyse des messages reçus par le serveur pour en récupérer l'action (définie au moyen d'une énumération et représente l'arrivée d'un nouveau joueur dans la partie, la déconnexion d'un joueur, la validation d'une action par le serveur, un message de chat, etc.). Pour éviter de redévelopper un parser JSON ou un format de données solide nous envoyons simplement nos chaînes dans un format convenu à l'avance (généralement des valeurs séparés par des virgules).

Elle permet enfin de définir des consommateurs appelés lorsqu'un événement a lieu (ce qui permet un code un peu plus simple et léger qu'un vrai gestionnaire d'événements).

### Classe SoundManager

Cette classe permet de gérer le son joué dans le jeu. Elle dispose d'un attribut permettant de contrôler le contenu actuellement joué, d'une énumération représentant tous les sons jouables, et d'une méthode play permettant d'écraser le son joué par un nouveau son.

Contrairement à la solution simple qui serait de donner le *SoundManager* au *GameState*, nous exposons plutôt un nouvel attribut *nextSound* à chaque fois qu'un *GameState* est créé et renvoyé au *Main*. Ainsi c'est le Main qui appelle play du SoundManager sur ce son (et nous sommes certain qu'un GameState "mort" ne peut plus jouer un son).

### Classe MessageBoardChatUI

Cette classe représente un champ d'entrée de message de chat (envoyé via un consommateur au main, qui transfère le message au serveur).

### Classe Main

Nous avons créé une méthode saveStateDispatchPlaySound qui permet de transmettre une action au serveur websocket et de jouer le son du dernier GameState.

Nous avons créé un TextMaker observable qui est donné à PlayersUI pour permettre d'ajouter des joueurs à la volée.

Nous avons créé une propriété observable qui permet d'obtenir le statut du joueur local (le fait que ce soit son tour ou non). Cette propriété est passée au ActionUI, au BoardUI pour empêcher le joueur d'entrer une action ou de voir la frange cliquable si ce n'est pas son tour.

**Classe BoardUI**

Nous avons décidé de ne pas afficher la frange pour les joueurs dont ce n'est pas le tour (il est pratique de voir très facilement si c'est notre tour, et la frange est une incitation à cliquer ce qui est parfois perturbant).

Un cadre jaune est aussi affichée sur la dernière tuile posée (il est très complexe dans une partie de 3-4 joueurs avec une centaine de tuiles posées de suivre le jeu autrement).