

Rapport de projet

Gabin Dudillieu Yago Iglesias Tony Ly Soan Mathusan Selvakumar
Yanis Lacenne

March 31, 2025

Contents

1	Objectif	2
2	Résumé	2
3	État de l'art	2
3.1	Enregistrement Vocal	2
3.2	Paramétrisation	3
3.2.1	SPro	3
3.2.2	Alternatives	3
3.3	Reconnaissance de l'individu	5
3.3.1	ALIZÉ	5
3.3.2	Alternatives	6
3.4	Reconnaissance de la parole	7
4	Implémentation	8
4.1	Le labyrinthe	8
4.1.1	Le Modèle	8
4.1.2	La Vue	12
4.1.3	Les Contrôleurs	16
4.2	Le son	22
4.2.1	Généralités	22
4.2.2	Alizé	24
4.2.3	Whisper	25
4.2.4	La vue	25
5	DataBase	29
6	Résultats	30
7	Conclusion	30

1 Objectif

L'objectif de notre projet est de créer un jeu vidéo qui puisse reconnaître les joueurs et leurs commandes par reconnaissance vocale. Nous nous limitons à 4 commandes (**UP**, **DOWN**, **LEFT** et **RIGHT**) et les joueurs reconnus seront les 5 créateurs du projet.

2 Résumé

Dans le cadre de ce projet, nous avons décidé de faire un jeu de parcours de labyrinthe. Nous disposons de deux modes de jeu différents: le mode classique et le mode Blackout, un mode de jeu où le labyrinthe devient invisible. Les technologies utilisées pour l'interface graphique sont les librairies *AWT* et *SWING* de java.

Du côté de la reconnaissance vocale, nous utilisons principalement quatre technologies:

- *Java Sound* : pour l'enregistrement du son.
- *SPro* : pour la paramétrisation du son.
- *ALIZÉ* : pour la reconnaissance du joueur.
- *Whisper* : pour la reconnaissance de la commande.

3 État de l'art

3.1 Enregistrement Vocal

Pour notre projet nous avons décidé d'utiliser *Java Sound* pour l'enregistrement vocal, cependant, ce n'était pas notre premier choix. Nous avons également étudié *ALSA*.

ALSA

ALSA est une abréviation de "**A**dvanced **L**inux **S**ound **A**rchitecture". C'est une bibliothèque de son de bas niveau qui fournit un support pour la lecture et l'enregistrement audio sur Linux.

En utilisant *ALSA*, les développeurs peuvent créer des applications audio puissantes et fiables pour **les utilisateurs de Linux**, offrant une expérience audio de qualité supérieure.

JavaSound

JavaSound est une bibliothèque de son de haut niveau pour Java. Elle fournit un support pour la lecture et l'enregistrement audio sur les systèmes d'exploitation **Windows**, **Mac OS X** et **Linux**. Elle est fournie avec le JDK de Java.

Comparaison

La librairie *ALSA* est **rédigée en C** contrairement à **JavaSound** qui est une bibliothèque **faite pour Java** et capable d'être intégrée sur plusieurs systèmes d'exploitation.

ALSA peut être difficile à prendre en main malgré un panel et une diversité d'options de configuration bien supérieure à celle de *JavaSound*. Toutefois, nous sommes **limité** à une programmation **exclusivement via Java**. De plus, *JavaSound* nous offre **parfaitement ce dont**

nous avons besoin et cela à un coût réduit, notamment par le fait que cette dernière est, elle, **de haut niveau**.

D'un point de vue global, les deux bibliothèques se valent et peuvent être intéressantes dans des cadres différents. Le choix revient donc aux programmeurs de choisir en fonction de leurs besoins et de leurs envies.

3.2 Paramétrisation

La **paramétrisation** consiste en l'extraction des paramètres d'un audio, avec l'objectif de l'analyser après. Les paramètres peuvent être modélisés comme des **vecteurs contenant les données des audios**. Cette paramétrisation est utile car ces données seront utilisées après pour la reconnaissance de l'individu. Il existe plusieurs bibliothèques de paramétrisation, dont nous avons choisi ***SPro*** pour notre projet.

3.2.1 SPro

La bibliothèque ***SPro***¹ en C est une bibliothèque logicielle de pointe conçue pour fournir des capacités avancées de traitement du signal aux programmeurs en C. La bibliothèque est conçue pour être **hautement efficace, rapide et facile à utiliser**, ce qui en fait un choix idéal pour une large gamme d'applications de traitement du signal.

La bibliothèque est construite sur une **architecture moderne et modulaire** qui permet aux utilisateurs de sélectionner et d'utiliser facilement les composants qui conviennent le mieux à leurs besoins. Elle fournit un ensemble complet d'outils et d'algorithmes pour le traitement des signaux, y compris le filtrage, l'extraction de caractéristiques, l'analyse spectrale et plus encore.

L'une des forces principales de la bibliothèque ***SPro*** est son **efficacité**. Elle est conçue pour minimiser l'utilisation de la mémoire et minimiser le temps de traitement, ce qui la rend bien adaptée pour une utilisation dans des environnements en temps réel et limités en ressources. Elle est aussi **hautement optimisée pour les performances**, en utilisant des techniques avancées telles que le multithreading, les instructions SIMD et les algorithmes vectorisés.

En plus de ses avantages en termes de performance, la bibliothèque ***SPro*** est également **très conviviale pour l'utilisateur**. La bibliothèque fournit une API simple et intuitive qui facilite aux développeurs de démarrer rapidement et de construire des applications de traitement du signal puissantes. La bibliothèque comprend une **documentation complète** et un grand nombre de programmes d'exemple, qui fournissent un excellent point de départ pour les utilisateurs qui découvrent le traitement du signal ou la bibliothèque.

Dans l'ensemble, ***SPro*** est un **excellent choix** pour toutes les personnes qui ont besoin de capacités avancées de **traitement du signal** dans leurs applications. Avec sa haute performance, sa facilité d'utilisation et son ensemble complet d'outils et d'algorithmes, elle a une large gamme d'applications et d'utilisations.

3.2.2 Alternatives

- **HTK**

HTK (HMM Tool Kit) est un **kit d'outils populaire pour le traitement du signal et de la parole**. Il fournit un ensemble complet d'algorithmes et d'outils pour la reconnais-

¹<https://pdfs.semanticscholar.org/1bc1/47e07028cbe6d2841e825c70abb60f8e1a25.pdf>

sance de la parole, la reconnaissance de l'orateur et la synthèse de la parole, entre autres choses.

Avantages

- **Ensemble complet d'outils et d'algorithmes pour le traitement du signal et de la parole**
- **Bien établi et largement utilisé** dans l'industrie et l'académie
- Grande et active communauté d'utilisateurs
- **Bonne documentation** et ressources de support.

Inconvénients

- **Peut être difficile à utiliser pour les débutants**, car le kit d'outils a une courbe d'apprentissage raide
- **Peut ne pas être aussi efficace que certaines des autres bibliothèques**, car il est conçu pour une large gamme d'applications et non spécifiquement pour les performances

• *MARF*²

Avantages

- **Architecture modulaire** : *MARF* a une architecture modulaire qui vous permet d'ajouter, de supprimer ou de remplacer facilement des composants pour répondre à vos besoins spécifiques.
- **Traitement audio** : *MARF* fournit une gamme de composants de traitement audio, y compris des techniques d'extraction de caractéristiques et des algorithmes d'apprentissage automatique, ce qui en fait un bon choix pour une variété de tâches de traitement audio.
- **Basé sur Java** : *MARF* est écrit en Java, qui est un langage de programmation populaire avec une large communauté de développeurs et une richesse de bibliothèques et d'outils.
- **Flexibilité** : La bibliothèque est conçue pour être flexible et personnalisable, vous permettant de l'adapter facilement à votre cas d'utilisation particulier.

Inconvénients

- **Complexité** : *MARF* a une architecture complexe et peut être difficile à comprendre pour les développeurs qui sont nouveaux dans le traitement audio et l'apprentissage automatique.
- **Performance** : *MARF* peut ne pas être aussi rapide que d'autres bibliothèques, telles que Alizé ou *SPro*, car elle a été conçue pour être plus axée sur la flexibilité et la modularité plutôt que sur les performances.
- **Communauté limitée** : *MARF* a une communauté plus petite par rapport à d'autres bibliothèques de traitement audio populaires, ce qui signifie que vous pourriez avoir plus de difficulté à trouver des réponses à vos questions ou un support pour la bibliothèque.

²<https://marf.sourceforge.net/docs/marf/0.3.0.6/report.pdf>

3.3 Reconnaissance de l'individu

L'objectif primordial de notre projet est la reconnaissance de l'individu qui parle. Comme nous l'avons vu dans la section 3.1, il est possible d'enregistrer un signal audio et de le traiter pour en extraire des informations utiles. Cependant, il est nécessaire de pouvoir reconnaître l'individu qui parle à partir de ces informations. Pour cela, nous avons étudié plusieurs bibliothèques de reconnaissance vocale, pour finalement choisir *ALIZÉ*.

3.3.1 ALIZÉ

*ALIZÉ*³ est une bibliothèque open source très puissante de reconnaissance vocale qui permet de développer rapidement et facilement des logiciels de reconnaissance vocale de haute qualité.

Elle est écrite en C++, et est munie d'une interface tant bien de bas niveau que de haut niveau, laissant ainsi une certaine liberté à l'utilisateur.

ALIZÉ fournit des outils nécessaires et utiles pour la conception de logiciels utilisant la reconnaissance vocale. Elle permet par exemple de traiter des signaux audios pour pouvoir obtenir des informations utiles. Elle comprend des algorithmes de détection de début et de fin de parole, de normalisation, de filtrage de bruit, de reconnaissance de la parole (à partir d'un fichier ou en temps réel), ...

Sa force est qu'elle met à disposition des modèles de reconnaissance très performants, des outils de développement complets, une interface haut niveau bien plus simple à utiliser, et elle prend en charge plusieurs langues telles que le français, l'anglais ou l'espagnol. Elle est aussi multi-plateforme, fonctionnant ainsi sur Linux, Windows et Mac OS.

Elle est aussi performante, surtout pour la reconnaissance du français, et plus documentée que la plupart des bibliothèques de reconnaissance vocale.

Pour résumer, *ALIZÉ* donne accès à des outils complets pour la reconnaissance vocale. Elle est aussi multi-plateforme, et multilingues et donne accès à une interface haut niveau facilitant son utilisation.

Son installation est cependant assez complexe, ce qui peut occasionner des pertes de temps lors du début d'un projet, et elle donne très peu d'exemples pratiques de son utilisation.

Fonctionnement de *ALIZÉ*

ALIZÉ est un logiciel dont le but est de faciliter la reconnaissance vocale, qui travaille sur deux niveaux distincts:

1. Le premier possède les données acquises, l'entrée, et le stockage.
2. Le second, plus haut, a accès aux utilitaires et aux algorithmes qui seront manipuler par l'utilisateur (management de listes, initialisation du model, les map, ...)

De cette manière, *ALIZÉ* gère: le data, les features, la mixture / distribution et la statistic.

Architecture :

ALIZÉ possède une architecture à plusieurs couches

La base est *ALIZE-Core* qui est bas niveau, incluant les fonctions pour utiliser les Gaussian mixture, et des IO pour différents formats de fichier.

Au dessus du core, il y a *LIA_RAL*, offrant des fonctionnalités de plus hauts niveaux Il est composé de :

³<https://alize.univ-avignon.fr/> et <https://github.com/alize-Speaker-Recognition>

- ***LIA_SpkDet*** un set d'outils pour faire toutes les demandes requises par le système d'authentification du speaker, et du model training, normalization, ...
- ***LIA_SpkSeg*** qui contient des outils pour le séquençage
- ***LIA_Utils*** qui contient les outils pour manipuler les différents formats de données utilisés par *ALIZÉ* (GMMs, features, ...)
- ***LIA_SpkTools*** qui propose des fonctions plus haut niveau, au dessus du core

3.3.2 Alternatives

Cependant nous avons étudié d'autres bibliothèques de reconnaissance vocale, comme *Kaldi* et *Julius*.

Kaldi⁴ :

Avantages :

- Open source
- Grande variété d'outil pour la formation et l'évaluation de modèles de reconnaissance vocale.
- Hautes performances, elle est souvent utilisée pour les applications de reconnaissance vocale exigeantes en termes de performances.
- Large communauté d'utilisateurs ce qui implique de nombreuses ressources en ligne et un grand nombre de personnes qui peuvent aider à résoudre les problèmes rencontrés.
- Flexibilité car elle peut être utilisée pour de nombreux types de reconnaissance vocale, tels que la reconnaissance de la parole en temps réel, la reconnaissance de commandes vocales, la reconnaissance de mots clés, ...

Inconvénients :

- Complexe ce qui demande plus de technique et de ressources dans ce domaine.
- Documentation limitée
- Temps de développement plus long

Julius⁵ :

Avantages :

- Open source
- Hautes performances
- Large communauté d'utilisateurs
- Facilité d'utilisation

⁴<https://kaldi-asr.org/> et <https://github.com/kaldi-asr/kaldi>

⁵https://julius.osdn.jp/en_index.php et <https://github.com/julius-speech/julius>

- Support de nombreux systèmes d'exploitation

Inconvénients :

- Documentation limitée
- Performances plus faibles que certaines autres bibliothèques
- Configuration difficile

CMU Sphinx⁶ :

Avantages :

- Open Source
- Portabilité
- Large communauté d'utilisateurs
- Faible consommation de ressources
- Supports plusieurs langues
- Documentation complète
- Propose des modèles de reconnaissance pré-entraînés pour de nombreuses langues

Inconvénients :

- Performances limitées (surtout pour les environnements bruyants et pour la reconnaissance de langues plus complexes)
- Configuration complexe
- Bas niveau

Conclusion des comparaisons

Nous avons choisi d'utiliser *ALIZÉ* pour la multitude d'outils qu'elle donne. De plus, elle est souvent conseillée pour sa grande capacité de reconnaissance vocale du français, ainsi que pour sa documentation et sa faible complexité en comparaison à d'autres bibliothèques.

3.4 Reconnaissance de la parole

Une fois que nous avons réussi à reconnaître l'individu, l'objectif est de savoir quelle est la commande que celui-ci a donné. Pour cela, nous utilisons la reconnaissance de la parole grâce à la bibliothèque *Whisper*, plus précisément de sa version optimisée, *FasterWhisper*.

⁶<https://cmusphinx.github.io/> et <https://github.com/cmusphinx>

Whisper et FasterWhisper

Whisper, un **modèle de reconnaissance vocale** de pointe développé par OpenAI, est formé sur un vaste ensemble de données audio diversifiées. **Polyvalent**, il est capable de gérer plusieurs tâches telles que la reconnaissance vocale multilingue, la traduction automatique de la parole et l'identification de la langue. Cependant, l'utilisation de *Whisper* présente une contrainte majeure : **son temps de traitement, qui peut aller de 10 à 15 secondes sur une machine locale**. Pour résoudre ce problème, *FasterWhisper* a été développé en utilisant *CTranslate2*, une bibliothèque d'optimisation pour les modèles de traduction automatique. Grâce à cette optimisation, le temps de traitement est réduit **environ de 1 à 3 secondes**, rendant la reconnaissance vocale **quasi-instantanée**. *FasterWhisper* est également **doté d'un CLI (Command-Line Interface)**, qui facilite l'interaction avec le modèle en permettant aux utilisateurs d'exécuter des commandes directement depuis l'invite de commande.

Avantages et limites

Le modèle *Whisper* offre de nombreux atouts. Il est **extrêmement polyvalent**, applicable à un large éventail de cas d'utilisation. De plus, il **fonctionne sur une grande variété d'appareils**. Cependant, ce modèle de reconnaissance vocale présente aussi des limites. L'une des principales contraintes est sa **consommation importante de puissance de traitement** pour un fonctionnement optimal, ce qui peut restreindre son utilisation sur des appareils aux capacités de traitement limitées. Par ailleurs, même si *Whisper* peut opérer en mode hors ligne, il requiert tout de même une connexion Internet pour la traduction de la parole et l'identification de la langue (ce qui n'est pas utilisé dans notre projet).

Alternatives

Whisper peut être employé dans diverses applications, telles que les assistants virtuels multilingues, la traduction de la parole et la transcription de conférences. Cependant, il existe **plusieurs alternatives à ce modèle, comme Google Speech-to-Text, Amazon Transcribe et Microsoft Azure Speech Services**. Chacun de ces modèles a ses avantages et ses inconvénients, et le choix dépendra des besoins spécifiques de chaque cas d'utilisation. Toutefois, contrairement à *Whisper*, ces alternatives **nécessitent** une connexion Internet pour fonctionner.

En conclusion, le modèle de reconnaissance vocale *Whisper* d'OpenAI et son optimisation *FasterWhisper* offrent des avantages significatifs pour les tâches de reconnaissance vocale pertinentes dans le cadre de notre projet.

4 Implémentation

La réalisation de notre projet suit le motif d'architecture logicielle Modèle-Vue-Contrôleur (MVC).

4.1 Le labyrinthe

4.1.1 Le Modèle

Nous avons une interface *MazeModel* avec son implémentation *MazeModelImplementation* qui servent à modéliser le Labyrinthe. Afin de représenter le labyrinthe, nous avons utilisé une matrice de booléens où *'true'* représente un chemin et *'false'* représente un mur.

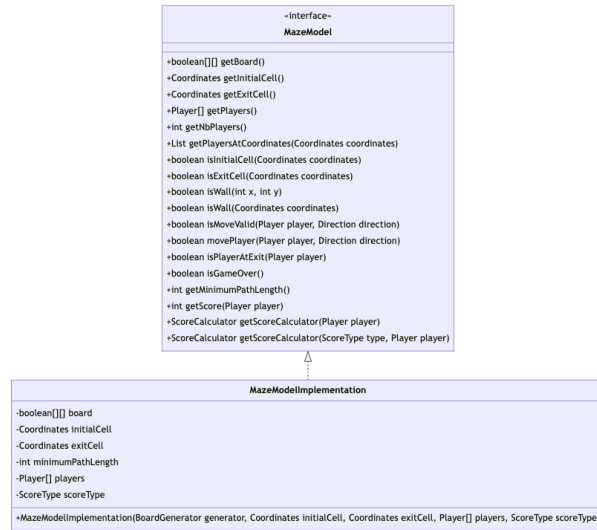


Figure 1: L'interface **MazeModel** et son implémentation **MazeModelImplementation**

Les labyrinthes sont générés grâce à un générateur qui est géré par l'interface **BoardGenerator** et son implémentation **DepthFirstGenerator**. L'algorithme de génération utilisé est le Randomized Depth-First Search.

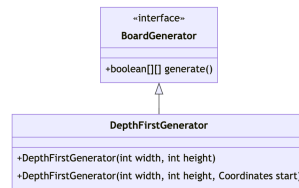


Figure 2: L'interface **BoardGenerator** et son implémentation **DepthFirstGenerator**

L'enum **GameMode** permet de définir les différents modes de jeu. Chaque type de jeu a sa propre implémentation de l'interface **GameModeData** qui sert à définir les données des paramètres de jeu (taille, type de score, difficulté) et la classe **GameData** se charge de regrouper ces données avec la liste des joueurs.

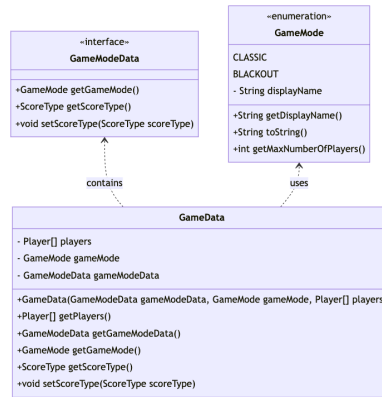


Figure 3: L'enum GameMode, l'interface GameModeData et la classe GameData

Les labyrinthes sont créés grâce aux méthodes de la classe **MazeModelFactory** qui se charge de mettre à disposition des méthodes qui permettent de créer des labyrinthes de différentes tailles selon les différents modes de jeu.

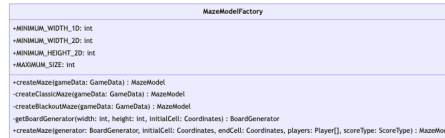


Figure 4: La classe MazeModelFactory

En ce qui concerne les joueurs, nous avons une interface **Player** avec son implémentation **PlayerImplementation** qui permettent de représenter un joueur. Ces derniers sont caractérisés par un nom, une couleur, des coordonnées, un statut, un nombre de mouvements et un temps qui correspond à la durée nécessaire pour finir le labyrinthe. Ces deux dernières données servent à calculer le score d'un joueur.

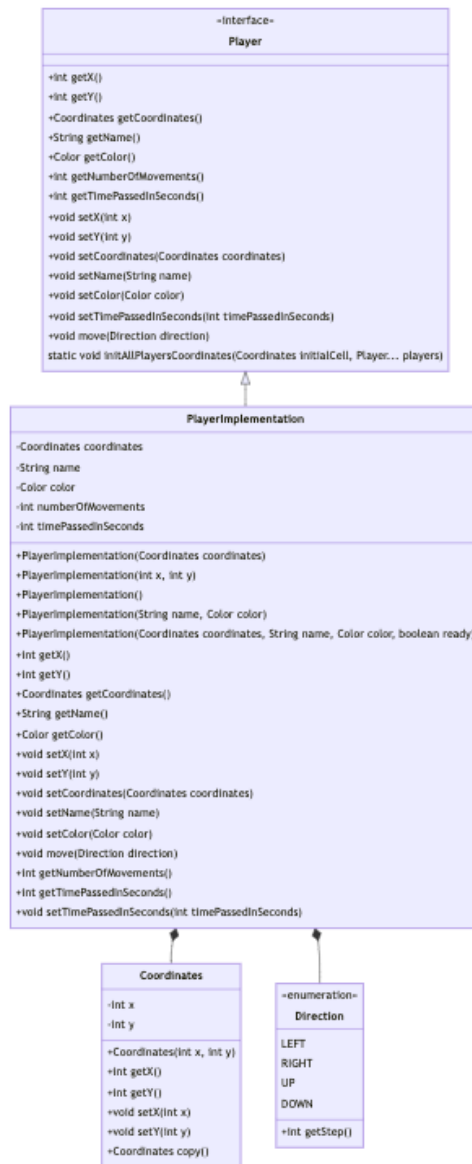


Figure 5: La classe Player et son implémentation

Pour pouvoir calculer le score des joueurs, nous avons plusieurs éléments qui utilisent les données évoquées précédemment. L'interface **ScoreCalculator** et ses implémentations permettent de calculer le score d'un joueur à partir d'une implémentation de **ScoreInfo** qui stocke les informations nécessaires. La classe **ScoreCalculatorFactory** se charge de créer des instances d'implémentations de **ScoreCalculator** en fonction du type de score et des informations selon le type de score (respectivement grâce à l'enum **ScoreType**, et **ScoreInfo**). Le calcul du score se fait mathématiquement en fonction du nombre de mouvements, du temps passé et de la difficulté du labyrinthe.

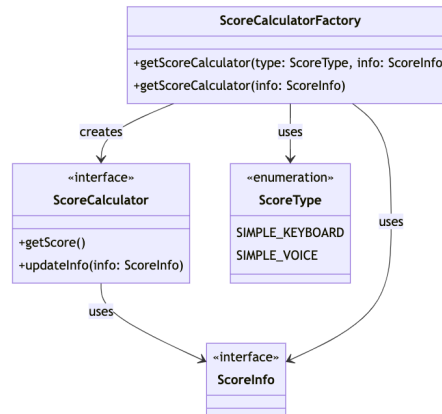


Figure 6: Les interfaces ScoreCalculator et ScoreInfo, la classe ScoreCalculatorFactory et l'enum ScoreType

L'enum *Direction* permet de définir les quatre directions de déplacement.

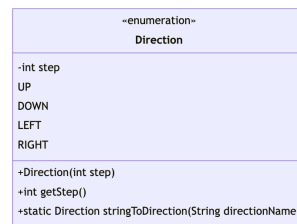


Figure 7: L'enum Direction

4.1.2 La Vue

Chacun des deux modes de jeu possèdent une vue qui lui est propre. Cela a été rendu possible grâce aux plusieurs classes et implémentations.

Principalement, nous avons la classe abstraite *MazeView* qui est étendue par la classe *MazeViewImplementation*. Cette dernière possède comme attribut une instance de *MazePanel* qui est chargée de dessiner le labyrinthe à partir de cellules créées grâce à la classe *Cell* qui se base sur les données du modèle.

Les labyrinthes ont un point de départ et un point d'arrivée que tous les joueurs doivent atteindre.

Le mode Classique

La vue du mode classique (8) est gérée par la classe *MazeClassicView* qui est une extension de *MazeViewImplementation*.

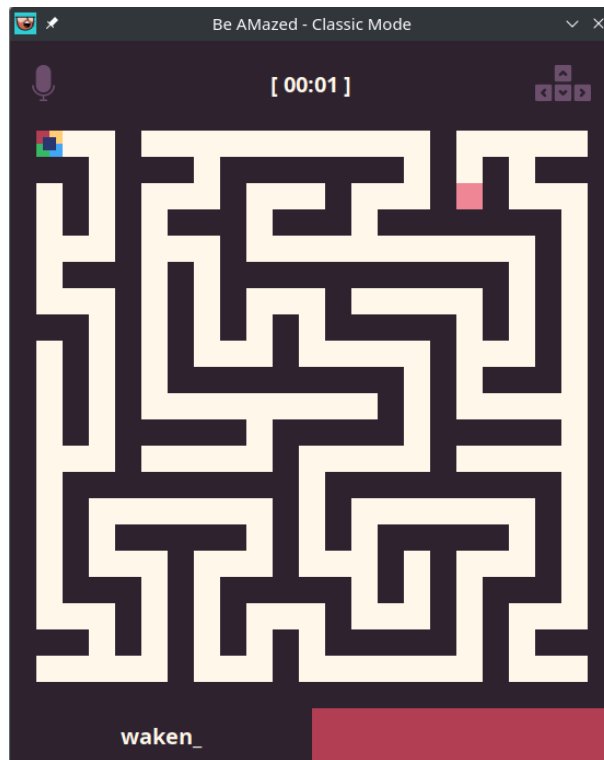


Figure 8: Image du mode Classique

Le mode Blackout

La vue du mode Blackout (9) est gérée par la classe *MazeBlackoutView* qui est une extension de *MazeViewImplementation*.

C'est un mode de jeu où les joueurs ne peuvent pas voir le labyrinthe périodiquement. Nous avons donc deux types de vues pour ce mode de jeu : une vue où le labyrinthe est visible, et une vue où le labyrinthe est caché.

Lorsque le labyrinthe est caché, les déplacements du joueur laissent derrière lui une traînée afin de l'aider à se repérer dans le labyrinthe.

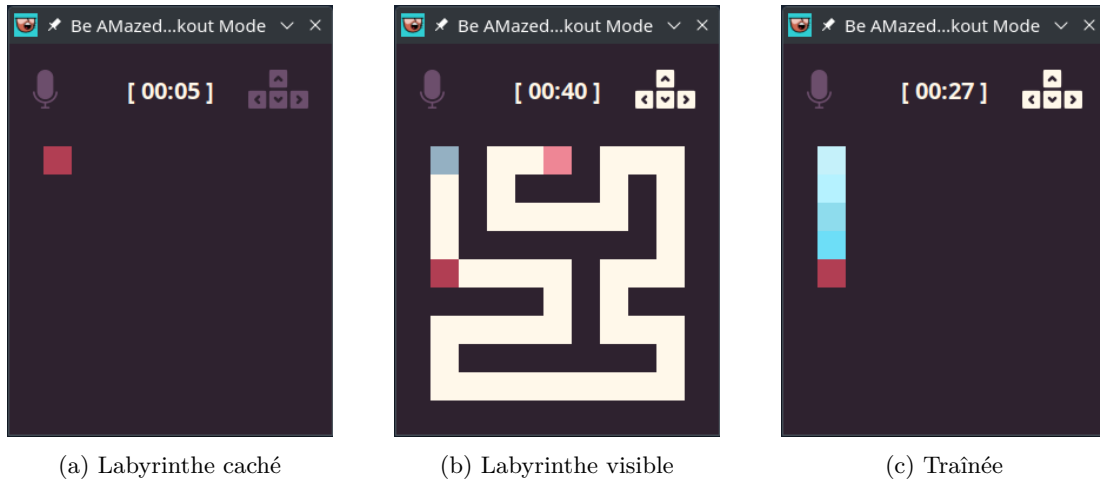


Figure 9: Images du mode Blackout

Les joueurs du labyrinthe

L’affichage des joueurs (10) est géré par les classes *MazePanel* et *Cell* en fonction de leurs positions dans le labyrinthe. Ils sont représentés par des rectangles de couleurs différentes qui changent de taille en fonction du nombre de joueurs présents sur une seule case. Nous avons choisi cette représentation car elle est simple et facile à comprendre d’un point de vue ergonomique.



Figure 10: Images des joueurs dans le labyrinthe

Nous avons en bas du labyrinthe une banderole qui indique le joueur qui doit jouer. Cette banderole est gérée par la classe *PlayerInfoPanel*.

Le décompte et le timer

Le décompte et le timer (11) sont affichés en haut au centre de chaque partie de jeu. Le décompte affiché en rouge au début de la partie permet aux joueurs de se préparer avant le début de la partie. Une fois le décompte terminé, le timer affiché en blanc commence à compter le temps écoulé depuis le début de la partie. Leurs affichage est géré par la classe *TimerPanel*.

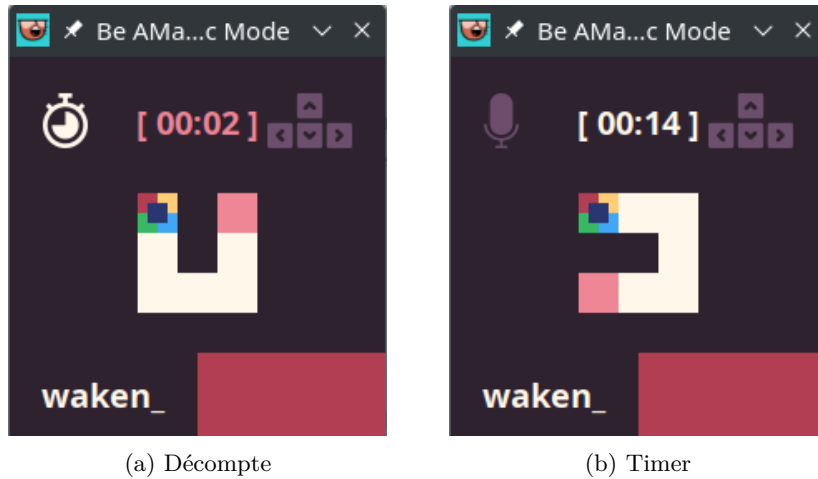


Figure 11: Images du décompte et du timer

Les indicateurs

L’affichage graphique du labyrinthe (8) contient un indicateur sur chaque extrémité du bandeau supérieur. L’indicateur du coin supérieur gauche représente le status du micro et de l’enregistrement audio. Celui du coin supérieur droit représente, quant à lui, si le déplacement des joueurs dans le labyrinthe via le clavier est activé. Il est possible d’activer chacune des icônes en cliquant dessus, ou bien d’utiliser les touches (SpaceBar) et T respectivement pour débiter l’enregistrement audio et activer/désactiver les déplacements via le clavier.

Fin de jeu

Lorsque la partie est terminée, après une courte animation de transition, une fenêtre de fin de jeu (12) s’affiche avec le tableau des scores de chaque joueur avec des boutons en bas pour retourner au menu principal, rejouer et quitter le jeu. Cette fenêtre est gérée par la classe *GameOverPanel* et *ScoreBoardPanel*.

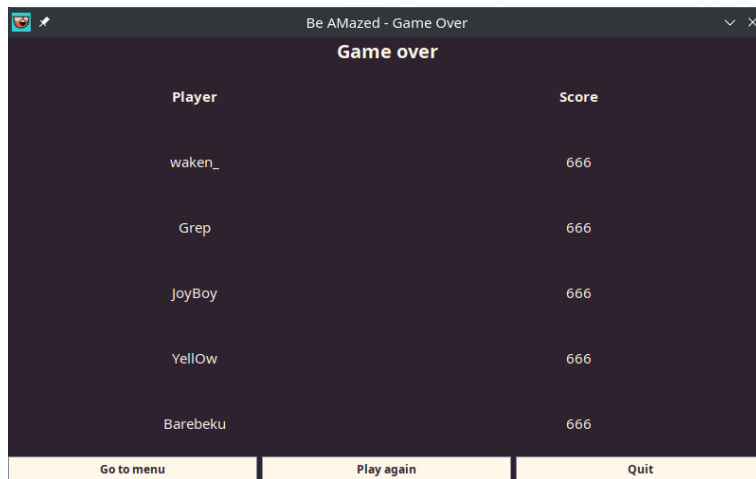


Figure 12: Image de la fenêtre de fin de jeu

Diagramme de classe de la vue

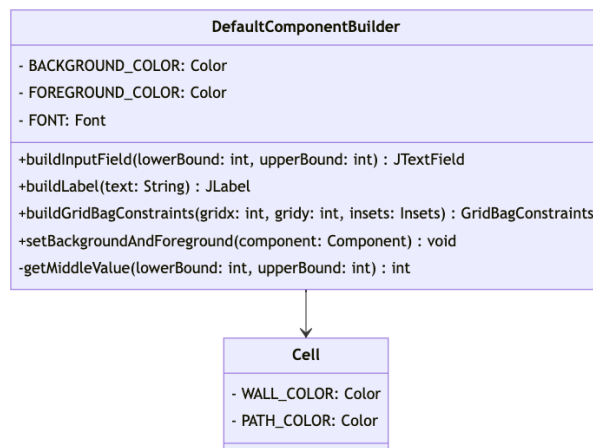


Figure 13: La classe DefaultComponentBuilder

4.1.3 Les Contrôleurs

Afin de garantir le bon fonctionnement de notre application, nous avons implémenté un contrôleur qui permet de gérer les interactions entre les différentes parties de notre application. Le contrôleur est composé de plusieurs classes qui permettent de gérer les différents menus, les paramètres audio, les règles du jeu, la création de partie, la partie en elle-même, etc. C'est ce qui lie notre

MVC et permet de faire fonctionner notre application.

Les mouvements Nous avons décidé de créer une interface *MazeController* qui est implémentée par *MazeControllerImplementation*. Cette interface contient toutes les méthodes qui permettent de gérer les mouvements du joueur.

Pour déplacer les joueurs nous utilisons principalement la voix, mais nous donnons aussi au joueur le choix d'utiliser le clavier. Pour gérer ces deux types de mouvements nous utilisons deux classes:




- *VoiceMovementController* qui permet de gérer les mouvements du joueur avec la voix
- *KeyboardMovementController* qui permet de gérer les mouvements du joueur avec le clavier

Ces deux classes sont utilisées en composition dans la classe *MazeControllerImplementation* qui permet de gérer les mouvements du joueur.

Les menus Afin de gérer la fenêtre de notre jeu et de naviguer entre les différents menus, nous avons créé la classe *MenuFrameHandler*. Cette classe se charge de changer le *JPanel* de la fenêtre principale de notre application, qui est une *JFrame*.

L'écran d'accueil (14)

Nous avons un écran d'accueil, géré par la classe *StartMenu*, qui possède des boutons pour:

- Aller au menu de création de partie 
- Aller au menu des paramètres audio 
- Afficher les règles du jeu 

Les différents éléments graphiques tels que le GIF et les images des boutons sont gérés par la classe *JPanelWithImages*.

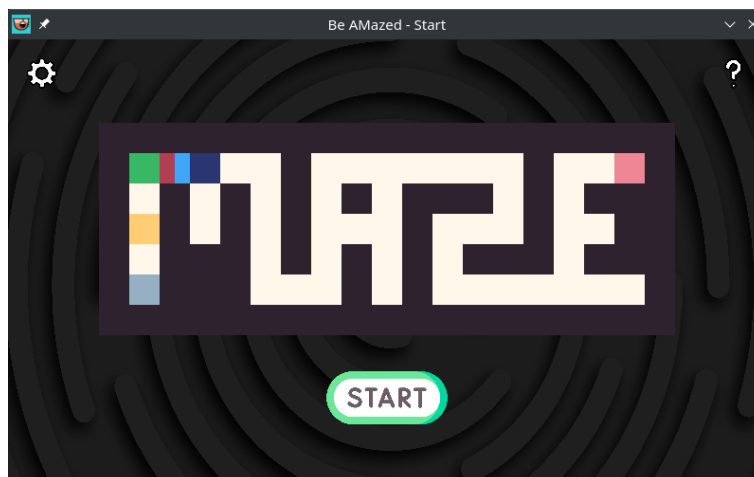


Figure 14: L'écran d'accueil

Les paramètres audio (15)

La vue des paramètres audio sera abordée dans la section 4.2.

Les règles de jeu Les règles du jeu sont affichées en anglais dans une fenêtre qui se superpose à celle du menu principal.

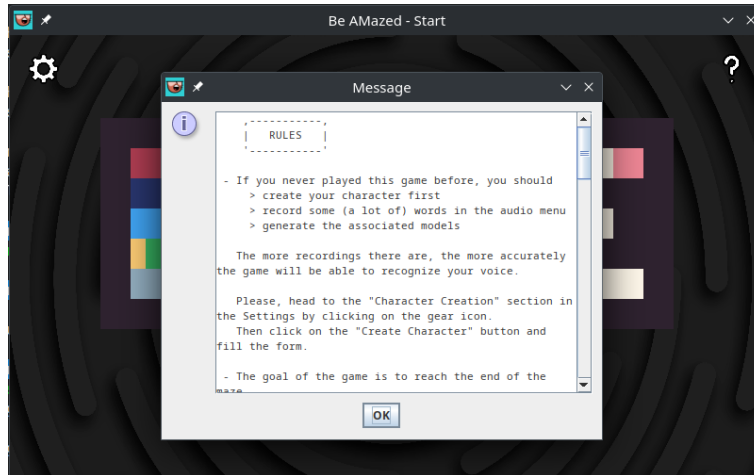


Figure 15: Les règles du jeu

La création de partie La création de la partie se décompose en plusieurs étapes et dispose de plusieurs éléments graphiques afin de permettre à l'utilisateur de choisir les paramètres de la partie qu'il souhaite jouer (nombre de joueurs, difficulté, etc.) le plus facilement possible. Cela est rendu possible par la classe *SettingsMenu* qui gère l'ensemble des menus de paramétrage de la partie.

Le choix du mode de jeu (16)

Le menu déroulant permet de choisir entre les différents modes de jeu disponibles. Il est géré par la classe *SelectionPanel*.

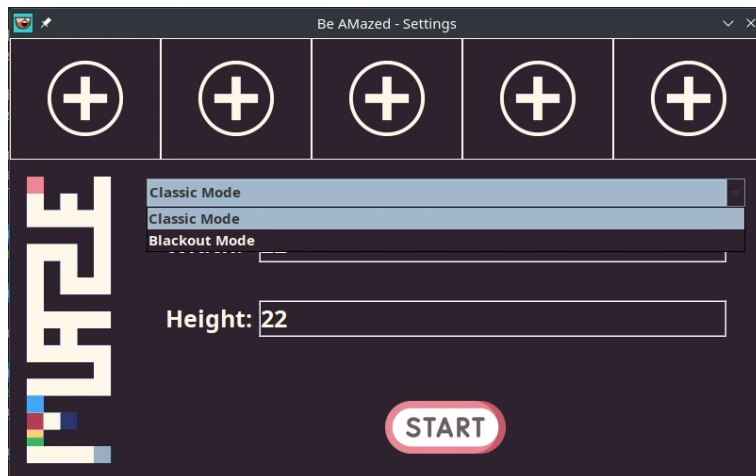


Figure 16: Menu déroulant des modes de jeu

Pour les afficher les menus des différents modes de jeu, nous avons des classes qui permettent de gérer les différents panels concernés. Ces classes sont des implémentations de l'interface *PanelHandler* et sont créées grâce à la classe *PanelHandlerFactory*.

Le mode Classique (17)

Grâce à la classe *ClassicPanelHandler*, nous avons à disposition deux entrées texte permettant de choisir la taille du labyrinthe.



Figure 17: Paramètres du mode classique

Le mode Blackout (18)

Grâce à la classe *BlackoutPanelHandler*, le mode Blackout possède 3 difficultés : *EASY*, *MEDIUM* et *HARD* qu'on peut sélectionner grâce à un menu déroulant.



Figure 18: Paramètres du mode Blackout

La préparation des joueurs (19)

Grâce aux classes *PlayerPanel*, *PlayerSelectionPanel* et *UserSelector*, nous pouvons choisir le nombre de joueurs, leur couleur et leur nom parmi la liste des joueurs qui ont déjà été créés précédemment.

Tout d'abord, pour ajouter un joueur, on clique sur une case "+". Une case de couleur s'affichera. Ensuite on choisit le nom du joueur dans la liste déroulante. Enfin il faut confirmer que le joueur est prêt en cliquant sur sa case.

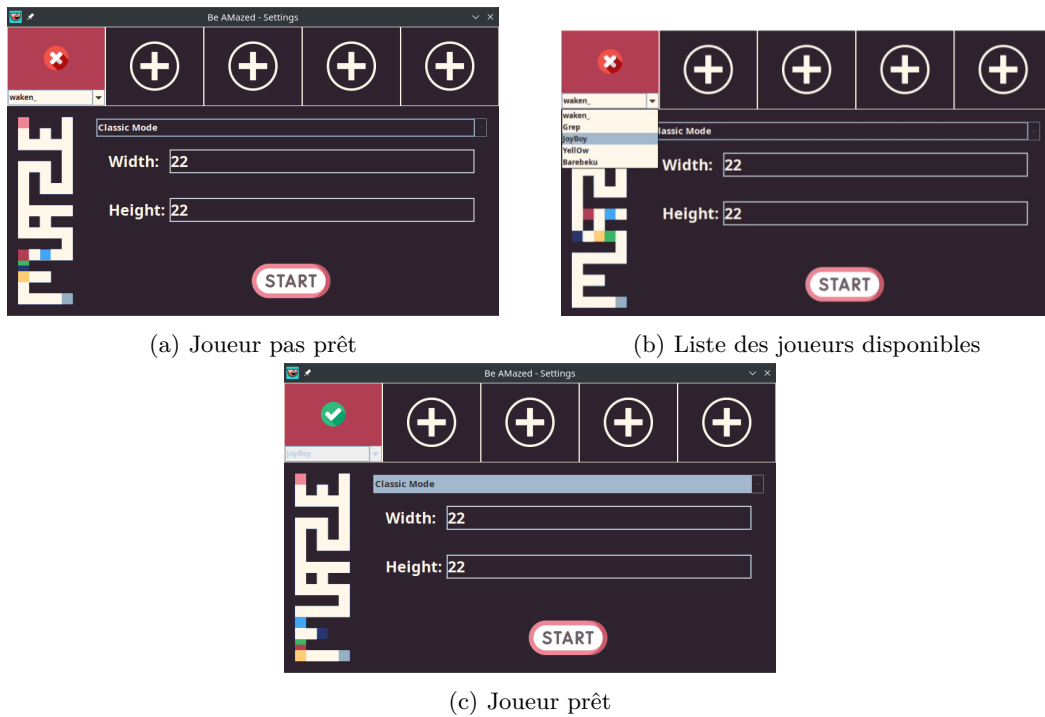


Figure 19: Ajout d'un joueur

Lancement de la partie Pour lancer la partie on clique sur le bouton **START** après que tous les joueurs soient prêts et que les paramètres précédents aient été choisis. Une fois la partie lancée on procède à la reconnaissance des joueurs. Avant de commencer à déplacer les personnages il faut que chaque joueur se fasse reconnaître par le jeu. À cette fin nous avons mis en place un menu qui permet de s'enregistrer jusqu'à 3 fois. Si le joueur n'est pas reconnu après 3 tentatives, il est éliminé de la partie. Ce menu est géré par la classe *RecognizeUserPage*.

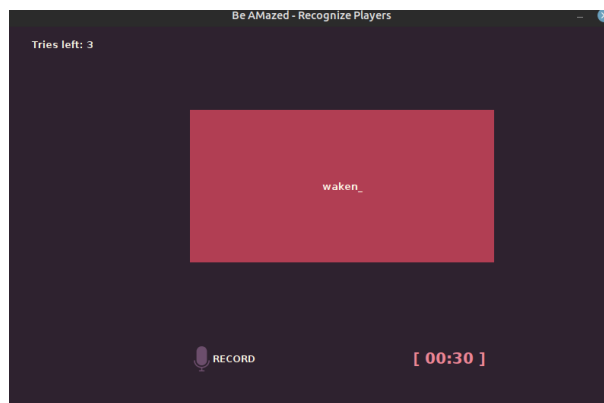


Figure 20: Reconnaissance des joueurs

Diagramme de classes pour le controleur

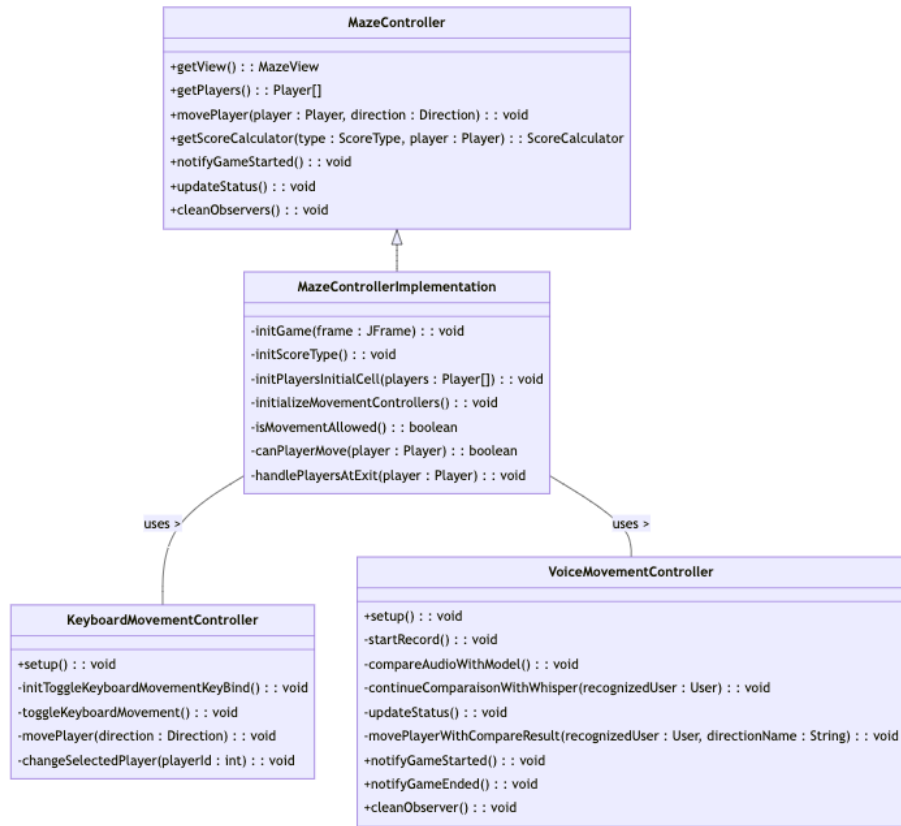


Figure 21: Diagramme de classes du controleur

4.2 Le son

4.2.1 Généralités

L'arborescence des fichiers audios se matérialise comme suit:

src/ressources/audioFiles/USER

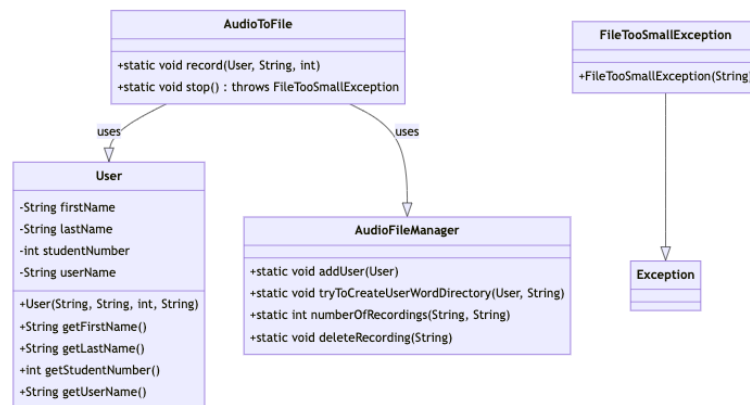
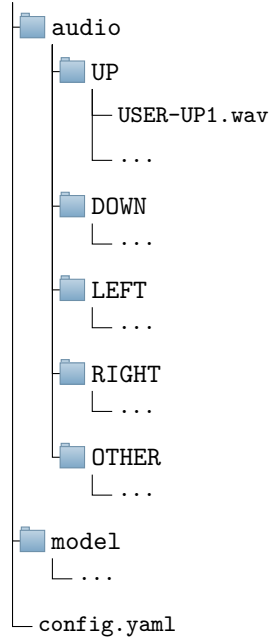


Figure 22: Les classes liées aux fichiers audio

4.2.2 Alizé

Modélisation :

Afin de reconnaître un utilisateur, nous avons tout d'abord besoin de modèles. Un modèle pour le monde qui prend les paramètres du plus d'audios possible pour enrichir son environnement. Un modèle pour l'interlocuteur qui prend les paramètres du plus d'audios possible pour un interlocuteur.

Le modèle du monde permet de créer le modèle de l'interlocuteur, en lui donnant l'environnement dans lequel il est. Plus les modèles sont enrichis, et plus ils seront efficaces.

Pour créer les modèles, nous avons tout d'abord besoin des paramètres fournis par *SPro*.

Ensuite, nous allons utiliser deux programmes afin d'améliorer la modélisation :

- ***EnergyDetector*** qui prend des paramètres d'audios et qui renvoie dans un fichier label les différents moments où l'interlocuteur parle (pour les différencier à quand il ne parle pas), d'extension *.lbl*.
- ***NormFeat*** qui prend les labels et les paramètres des audios afin de les normaliser, et qui renvoie de nouveaux paramètres normalisés d'extension *.norm.prm*.

On procède alors à la modélisation du monde avec tout les paramètres du monde normalisés et des labels, afin d'obtenir son modèle qui est un **GMM** (Gaussian Mixture Model), et qui a comme extension *.gmm*.

Pour cela, on utilise le programme TrainWorld de *LIA_SpkDet*.

A l'aide du modèle du monde, et des paramètres normalisés des interlocuteurs ainsi que des labels, on crée leurs modèles, qui seront aussi des **GMM**, d'extension *.gmm*. Pour cela, on utilise le programme TrainTarget de *LIA_SpkDet*.

Reconnaissance :

Maintenant que les modèles des interlocuteurs sont créés, on va les utiliser pour reconnaître un interlocuteur.

Pour cela, on utilise le programme ComputeScore de *LIA_SpkDet*.

Ce programme va prendre dans un fichier, les audios à comparer, suivis des interlocuteurs qu'on veut essayer de reconnaître. Il donnera à la fin dans un *.txt* toutes les données de comparaison et donc une valeur de comparaison.

Pour reconnaître un interlocuteur, on utilise toutes les valeurs données, afin de tirer l'interlocuteur qui a la valeur maximum de comparaison, et dépassant un certain seuil (choisir arbitrairement selon les tests résultant).

Cet utilisateur sera donc celui qui est reconnu.

Utilisation dans le projet :

Dans notre jeu, on réalise toutes les étapes précédentes, en prenant en compte la gestion de fichiers de notre jeu. Les listes utilisées pour la modélisation, afin de fournir les informations au programme de quels audios paramétrer, normaliser et modéliser, sont actualisées au moment où la modélisation a besoin d'être faite (c'est-à-dire quand il y a une modification dans les audios).

Lors de cette étape, tout les audios sont paramétrisés et normalisés, le modèle du monde est créé avec tout les audios disponibles. Les modèles des interlocuteurs (ou des mots d'interlocuteur) sont à la suite créés avec les audios séparés par utilisateur (et par mot) dans le gestionnaire.

La modélisation se fait dans fichier *src/main/exe/model* qui contient tout les répertoire séparant la modélisation.

La reconnaissance, enfin, se fait en comparant tous les modèles des interlocuteurs avec les paramètres normalisés de l'audio émis pour une commande dans la partie : `currentAudio`.

À la fin, si un interlocuteur qui a la valeur la plus haute de comparaison, atteint un certain seuil, ce joueur jouera la commande donnée.

Intégration en Java :

Afin de gérer toute la partie audio, nous avons créé une classe **ModelManager** qui s'occupe de la modélisation et de la reconnaissance. Elle peut exécuter les programmes de *ALIZÉ* et de *SPro* grâce à la classe **RunSH**. L'objectif de **ModelManager** est, non seulement l'exécution des programmes, mais aussi la gestion des fichiers et des listes de fichiers à utiliser pour la modélisation et la reconnaissance. Ces fonctions sont partagées avec la classe **AudioManager** qui s'occupe de la partie audio.

Comment obtenir un résultat? Une fois que *ALIZÉ* a fait son travail il faut interpréter le résultat. Ce procédé est réalisé par les classes **AlizeRecognitionResultParser**, qui s'occupe purement de parser le résultat de la reconnaissance, et **AudioRecognitionResult** qui s'occupe de la gestion des résultats de reconnaissance.

4.2.3 Whisper

Utilisation du client FasterWhisper :

Après l'installation de *FasterWhisper*, un interface en ligne de commande (CLI) est disponible depuis le terminal. Cette interface permet d'interagir avec le modèle *FasterWhisper*, qui a été amélioré à partir du modèle *Whisper* d'OpenAI. Afin de configurer les paramètres du modèle et de simplifier l'utilisation de l'interface en ligne de commande, un script bash appelé **whisper.sh** a été créé.

En faisant appel au CLI de *FasterWhisper* sur un fichier audio, un fichier JSON est généré, contenant les informations cruciales extraites du fichier audio.

Intégration avec Java :

L'application accède aux fonctionnalités de *FasterWhisper* grâce à la classe **Whisper**, qui prend en charge l'exécution du script bash et la récupération du résultat de la transcription. Une fois le processus effectué par *FasterWhisper*, le texte transcrit est récupéré à l'aide d'un parseur JSON provenant de la bibliothèque Jackson, préalablement intégrée au projet. Ce parseur exploite le modèle **WhisperResult**, faisant office de structure pour le fichier JSON, pour atteindre son objectif. Tout cela se fait de manière *Thread-Safe*, pour éviter les problèmes de concurrence. La classe **ThreadedQueue** est utilisée pour gérer un système de file d'attente, permettant de gérer les requêtes concurrentes.

4.2.4 La vue

Pour pouvoir se déplacer dans le labyrinthe, l'existence de fichiers audios stockant nos voix est nécessaire. Et pour permettre à n'importe quel utilisateur de pouvoir créer ses propres fichiers audios, il faut qu'il y ait une interface graphique qui lui facilite la tâche.

On note ainsi les deux points clés essentiels que doit pouvoir faire un utilisateur:

- Ajouter / Modifier / Supprimer un joueur.
- Enregistrer / Écouter / Supprimer un audio.

Le Menu Audio

Le menu audio est la première interface graphique que l'utilisateur voit lorsqu'il se rend dans les paramètres du jeu. Il permet de gérer la quasi-totalité des besoins précédemment énoncés, et envoie vers les pages qui gèrent les autres. La classe *AudioMenu* tient les ficelles de ce menu.

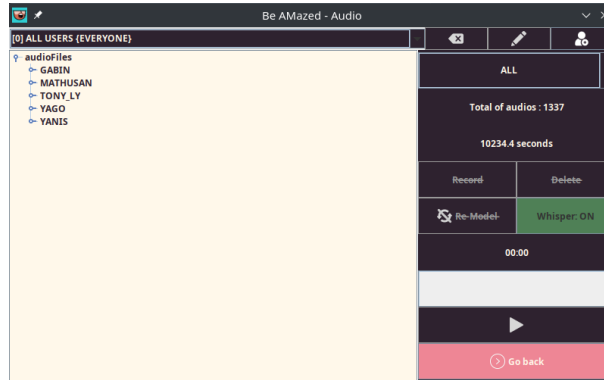





Figure 23: Image du menu Audio

De là, on remarque que le menu est divisé en trois parties. Le centre de l'écran qui représente l'arborescence des fichiers audios (4.2.1). Puis sur la partie supérieure de l'écran, on retrouve les boutons qui permettent la gestion des utilisateurs. Enfin, sur la droite de l'écran, on retrouve les boutons qui permettent une vue globale et une gestion des fichiers audios.

L'arborescence des fichiers audios L'arborescence des fichiers audios est représentée par un objet de la classe *FileTree* qui étend la classe *JTree* de Java. Elle permet une personnalisation accrue de l'arborescence, notamment en permettant de changer la couleur des noeuds, ou encore de changer l'icône des noeuds feuilles. De plus, à l'aide d'une *regex*, nous avons pu afficher uniquement les fichiers audios qui intéressent l'utilisateur.

La gestion des utilisateurs Cette partie du menu se décompose en quatre éléments

- Sélection des utilisateurs dans la liste
- Ajouter un utilisateur 
- Modifier un utilisateur 
- Supprimer un utilisateur 

La gestion des fichiers audios Sans compter le bouton de retour au menu principal, cette partie du menu se décompose en 8 éléments

1. Le choix du mot
2. Le nombre d'audios enregistrés
3. La durée totale déjà enregistrée
4. Un bouton d'enregistrement (4.2.4)

5. Un bouton de suppression
6. Un bouton de modélisation
7. Un bouton ON/OFF concernant l'utilisation de whisper
8. Un timer
9. Une barre de progression
10. Un bouton d'écoute

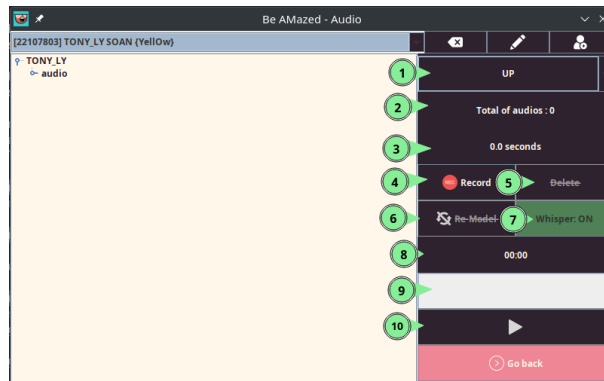


Figure 24: Image de l'interaction avec les fichiers audios

Le choix du mot est un élément qui permet de choisir le mot que l'on souhaite enregistrer. Il est composé d'une liste déroulante qui contient tous les mots disponibles dans le jeu (Voir la classe **WordToRecord**) et en y ajoutant le mot global **ALL**. Lorsqu'un mot est sélectionné, le nombre d'audios enregistrés et la durée totale sont mis à jour en fonction du mot choisi et du joueur sélectionné.

Lorsque un audio est sélectionné, les boutons de suppression et d'écoute s'activent. Si le bouton d'écoute est cliqué, alors l'audio est joué et au la barre de progression et le timer sont mis à jour en temps réel en fonction de la durée de l'audio.

Si le bouton de suppression est cliqué, alors l'audio est supprimé et l'arborescence est mise à jour.

Et pour pouvoir cliquer sur le bouton **Record**, il est nécessaire que le joueur et le mot aient été choisis.

La page d'ajout/de modification d'un utilisateur

Les pages de création et de modification de l'utilisateur sont identiques, à l'exception du bouton de validation qui est respectivement **Create** et **Save**. La classe **EditCreateUsersPage** gère ces pages.

La page d'enregistrement d'un audio

La page d'enregistrement permet d'enregistrer des audios pour un mot et un joueur donné. La classe **RecordPage** gère cette page. On y retrouve un timer, le nombre d'audios enregistrés depuis le début de la "session", un bouton d'enregistrement **Record**, un bouton **Discard**, un

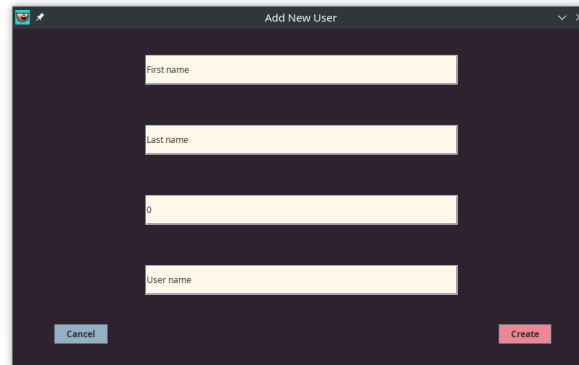


Figure 25: Image de la page d'ajout/de modification d'un utilisateur

bouton **Discard All** et un bouton de retour au menu audio. Lorsque le bouton **Record** est cliqué, l'enregistrement commence, le countdown du timer commence et le bouton **Record** devient **Stop**. L'enregistrement s'arrête lorsque le bouton **Stop** est cliqué ou lorsque le décompte se termine. Si le bouton **Discard** est cliqué pendant que l'enregistrement est en cours, alors ce dernier se stoppe et l'audio est supprimé, sinon le dernier audio est directement supprimé. Si le bouton **Discard All** est cliqué, alors tous les audios enregistrés depuis le début de la "session" sont supprimés après confirmation de l'utilisateur.

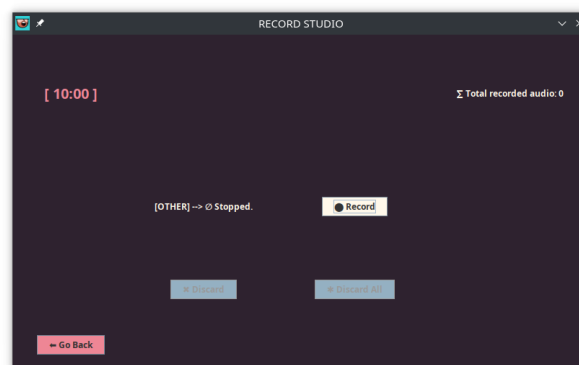


Figure 26: Image de la page d'enregistrement

Diagramme de classe de la vue de la partie audio

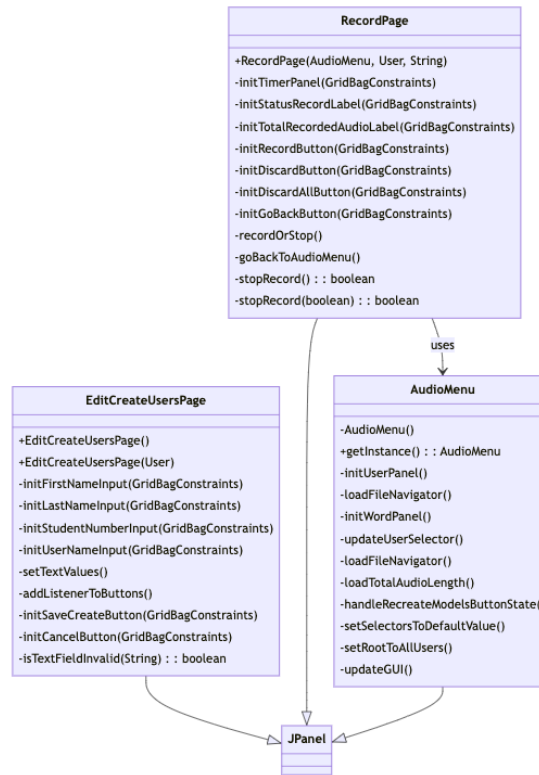


Figure 27: Vue de la partie audio

5 DataBase

Afin de pouvoir tester et créer des modèles efficaces, nous avons créé différentes bases de données d'audios.

Tout d'abord, dans le dossier *dataBaseOfWords*, nous pouvons trouver des audios de tests pour chaque mot indiquant une direction, provenant de 5 utilisateurs différents, soit un total d'environ 400 audios d'environ 3 secondes chacun. De plus, afin de disposer d'une base solide pour les modèles, nous avons enregistré dans les dossiers respectifs *src/resources/audioFiles/USER/OTHER*, " Pour mettre au bon endroit le mot pour chaque utilisateur, une banque d'audios contenant au moins 40 minutes chacune, découpées en fichiers audio de 10 secondes. Cela représente donc un total d'environ 4400 audios. Enfin, à l'intérieur du dossier *src/main/exe/model/gmm*, vous trouverez également des modèles déjà créés, offrant ainsi la possibilité à l'utilisateur d'éviter l'étape de modélisation à son arrivée sur le jeu, d'une durée d'environ 3 minutes.

Maintenant, lorsque l'utilisateur arrive sur le jeu, il dispose alors déjà d'un environnement lui permettant de créer efficacement des modèles, ainsi que des audios pour tester les différentes fonctionnalités du jeu.

6 Résultats

En ayant testé sur des audios de tests réalisés par nous-mêmes, disponible dans le dossier *DataBaseOfWords*, un taux de discrimination de l'utilisateur d'environ 60% est observable, pour un total d'environ 400 audios testés. Nous avons aussi remarqué que certains utilisateurs ne sont pas du tout reconnus, alors que d'autres le sont dans la quasi-totalité des cas.

Nous avons donc réussi à développer un jeu capable de différencier, et ce, de manière plus efficace que le hasard, les joueurs à partir de leur voix dans une partie grâce à ALIZE et d'interpréter les paroles prononcées comme des commandes grâce à Whisper, c'est-à-dire comme des actions à exécuter.

Ainsi, le joueur peut se déplacer dans le labyrinthe juste avec sa voix s'il est bien reconnu.

Toute la gestion des modèles est faite à l'intérieur du jeu, nous n'avons donc pas besoin de lancer un autre programme pour enregistrer la voix ou pour générer les modèles. Le délai de réponse est assez rapide, le joueur peut se déplacer dans le labyrinthe sans avoir à attendre généralement plus de 3 secondes après avoir prononcé une commande.

Enfin, nous avons réussi à faire en sorte que le jeu n'ait pas besoin d'un accès à internet pour fonctionner, même si certaines fonctionnalités de *Whisper* en ont besoin. De plus, nous avons également réduit la quantité de ressources nécessaires à un ordinateur pour exécuter correctement notre programme, permettant alors une certaine stabilité pour ceux étant peu performants.

7 Conclusion

Ce projet a été très enrichissant car il nous a permis, non seulement d'approfondir nos connaissances en Java, mais aussi d'apprendre à rechercher des informations sur des librairies inconnues. De plus, nous avons beaucoup apprécié le fait que le sujet porte sur la reconnaissance vocale et la gestion et manipulation de fichiers audio, ce qui n'est pas souvent abordé dans les cours de programmation.

Cependant, nous avons eu des difficultés dues à la complexité de certaines librairies, en particulier ALIZÉ. Même si, dès le début, on a mis l'accent sur la recherche, nous avons eu du mal à trouver des informations sur les librairies que nous avons utilisées, surtout parce qu'il manque des informations sur les forums et qu'elles ne sont plus maintenues à jour, et aussi des connaissances requises quant à la manipulation et à la prise des paramètres pour les audios. Du point de vue de l'implémentation du jeu tout s'est bien passé. Malgré cela, le fait de ne pas maîtriser les librairies nous a fait perdre du temps, ce qui a eu un impact sur la qualité du jeu car on n'a pas pu passer autant de temps que nous l'aurions souhaité sur l'aspect graphique. Toutefois, nous sommes très satisfaits du résultat final car nous avons réussi à implémenter un jeu de qualité, qui est très agréable à jouer et qui n'a pas besoin d'autres applications pour fonctionner: une fois installé tout se passe via l'interface graphique.