Projet semestre 1 Jeu de Go

BOËDA Gautier LE ROUZIC Steven

ENSICAEN 2013-2014, 1A Info, Groupe TP 2

Contents

1 Environnement de développement

Le programme a été réalisé principalement sous Windows, avec l'IDE Eclipse et l'environnement MingW. Nous avons utilisé la librairie SDL 1.2 ainsi que ses addons SDL_image et SDL_ttf. La librairie matrice écrite en TP a été réécrite pour correspondre aux conventions établies pour ce projet. Nous avons utilisé Git comme système de versionnage afin de faciliter le travail depuis de multiples stations de travail, et GitHub comme dépôt central. (http://github.com/stevenlr/ProjetGO)

Chaque sous-projet était organisé comme ceci : un dossier `assets` contenant les images, les fontes et autres ressources, un dossier `lib` contenant les librairies nécessaires à la compilation, un dossier `include` contenant les fichiers d'en-tête des librairies et un dossier `src` contenant le code source et un dossier `include` contenant les en-têtes.

Des conventions d'écriture ont été adoptées : nous avons utilisé le CamelCase, le code était écrit le plus possible en français (à l'exception des mots `get` et `set`) et les prototypes de fonctions suivaient le style `Type_nomFonction` où `Type` est le type sur lequel opèrait la fonction.

Les fichiers sources étant organisés en sous-dossiers, un fichier `Makefile` a été placé dans chaque dossier en appelant un autre commun à tous ceux-ci contenant les instructions de compilation pour chaque sous dossiers, ainsi que les autres makefiles des sous-dossiers. Tous les fichiers objets générés étaient alors placés dans un même dossier `obj` d'où ils étaient assemblés en l'exécutable final. **Les instructions de compilation sont spécifiées dans le fichier README.md.**

Ce rapport a été écrit en Markdown puis compilé en PDF grâce à l'utilitaire Pandoc.

2 La libmatrice

La librairie matrice écrite en TP a été entièrement réécrite afin de suivre les conventions établies, en particulier l'encapsulation des données. En interne, une représentation linéaire des données a été utilisée au lieu d'un tableau bidimensionnel.

3 Le jeu de Go

3.1 Structures de données

3.1.1 Liste

3.1.2 Pile

3.1.3 Position

3.1.4 Couleur

3.1.5 Pion

3.1.6 Chaîne

3.1.7 Territoire

3.1.8 Plateau

3.1.9 Partie

3.1.10 Tutoriel

3.2 Algorithmes

3.2.1 Déterminer les libertés

3.2.2 Déterminer territoire

3.2.3 Déterminer séki

3.2.4 Déterminer les yeux d'une chaîne

3.2.5 Déterminer une chaîne

3.2.6 Réaliser une capture

3.2.7 Déterminer les chaînes entourant un territoire

3.2.8 Déterminer les chaînes capturées

3.2.9 Calcul du score

3.3 Intelligence Artificielle

Une intelligence artificielle relativement simple (mais non naïve) a été implémentée. Son exécution se déroule en plusieurs phase : la phase de passage de tour,

de défense, d'attaque et de jeu naïf.

3.3.1 Passage de tour

Si l'adversaire vient de passer son tour, alors l'ordinateur a une chance sur quatre de faire de même. Celà permet, lorsqu'un humain joue contre l'ordinateur, de terminer le jeu dans des temps raisonnables si le joueur décide qu'il ne peut plus rien faire qui pourrait l'avantager. En effet, cette IA ne réfléchira pas directement à la possibilité de passer son tour si elle n'a plus l'avantage nul part sur le plateau.

Si l'ordinateur n'a pas décidé de passer, alors la phase de défense commence.

3.3.2 Phase de défense

L'ordinateur va commencer par vérifier s'il est en danger quelque part sur le goban : si une de ses chaînes ne possède plus qu'une liberté. Si c'est le cas, il va placer un pion sur cette liberté afin d'essayer de se libérer. Le système de placement de pion permet de s'assurer que si le coup joué résulte en une chaîne n'ayant plus de libertés pour l'ordinateur lui-même, alors le coup est annulé, car illégal.

Si l'ordinateur n'a pas joué défensivement, il va passer à l'offensive.

3.3.3 Phase d'attaque

L'ordinateur va alors chercher sur le goban toutes les chaînes ennemies et sélectionner celle possédant le moins de libertés. S'il en trouve une, il va placer un pion sur une de ses libertés.

Si l'ordinateur n'a pas pu jouer de coup offensif, alors il va passer à la phase naïve.

3.3.4 Phase naïve

Le dernier recours de l'ordinateur est alors de placer un pion au hasard sur les cases restantes du goban. S'il n'a pas réussi à en placer, alors il passe son tour.

4 Architecture du programme

Le programme a été structuré de façon à rendre l'implémentation des contexte console et graphique la plus séparée de la logique du jeu possible et à éliminer le maximum de redondances.

4.1 Les écrans et leurs états

Le programme fonctionne comme une machine à états, chaque état étant un écran. Les écrans sont : le menu, le tutoriel, les options de partie et le jeu. Chaque écran est donc une partie différente du programme qui se caractérise par une logique, des entrées et des sorties différentes. Ainsi chaque écran possède sa fonction logique (exemple `EcranJeu_main`) et des fonctions d'interface dont nous reparlerons plus tard. Chaque écran possède également ses états, c'est-à-dire des variables statiques à l'écran en cours qui définissent par exemple si on doit continuer à exécuter l'écran actuel, dans quelle partie on se trouve actuellement, etc. Les états sont initialisés à l'initialisation de l'écran. Lorsque le programme doit passer à un autre écran, l'écran courant initialise l'écran suivant avec éventuellement les paramètres associés, puis s'auto-détruit. Le gestionnaire d'écran est alors notifié du changement et, à moins que la fermeture de l'application soit demandée, va lancer la fonction `main` du nouvel écran.

4.2 Les contextes

Etant donné le besoin de pouvoir exécuter le programme à la fois en console et en mode graphique, il a fallu séparer au maximum ce qui était spécifique au contexte d'exécution de la logique du jeu, afin d'éviter toute redondance. Ainsi deux contextes sont définis : le contexte graphique et le contexte console. Au début du programme, le contexte d'exécution est choisi et initialisé. L'initialisation permet par exemple dans le cas du contexte graphique d'ouvrir la fenêtre, de charger les textures, de créer les boutons, etc. De même, à la fin du programme le contexte est détruit. Une structure similaire aux états pour les écrans contient les variables utiles à l'exécution du contexte dans la suite de l'application. Par exemple, le contexte graphique contient le pointeur `SDL_Surface` représentant l'écran.

4.3 Les interfaces

Pour pouvoir interagir en entrée et en sortie avec l'utilisateur, nous avons défini des interfaces. Chaque écran possède une interface de sortie et une d'entrée pour chacun des contextes. Ces fonctions ont connaissance des états mais ne peuvent pas les modifier. En effet, elles ne gèrent strictement que les entrées et sorties et ne doivent en aucun cas avoir conscience de la logique du jeu.

4.4 Les événements

Afin donc de pouvoir faire évoluer les états de jeu, les interfaces appellent des fonctions événements. Chaque écran possède ses propres fonctions d'événements. Par exemple pour l'écran de jeu : un événement pour le pose d'un pion, un pour

l'enregistrement de la partie, un pour le retour à un tour précédent, etc. Ces fonctions sont alors le coeur de la logique du jeu.

5 Autres éléments du programme

5.1 Eléments d'interface graphique

Afin de faciliter les entrées utilisateurs, des structures ont été créées pour faciliter l'affichage et la gestion des éléments d'interface graphique. Chaque élément est instancié lors de la création du contexte graphique et stocké dans une variable globale afin de pouvoir être utilisé plus tard.

5.1.1 Les boutons

Le premier élément d'interface graphique est le bouton. Il lui est spécifié une taille, une position, un texte, une couleur de fond et une couleur de texte. Une fois créé il est possible lors d'un clique de souris de tester si celui-ci était sur le bouton afin d'engendrer d'autres événements.

5.1.2 Les boutons à choix multiple

Le second élément est un bouton à choix multiple. Il permet à l'utilisateur de choisir une option parmi une liste proposée. Il lui est spécifié une taille par choix, une position, des couleurs de fond pour l'état sélectionné et normal d'un choix et la couleur du texte. Les différents choix (au nombre maximal de 3) sont ensuite ajoutés. Une fois créé, on peut lors d'un clique de souris tester si celui-ci était sur le bouton à choix multiple et si c'est le cas, déterminer quelle option était choisie. Il est alors possible de récupérer le numéro de l'option choisie. Ces boutons sont utilisés lorsque l'utilisateur a un choix fini. Exemple : humain ou ordinateur, taille du plateau.

5.1.3 Les champs de texte

Enfin, le champ de texte permet à l'utilisateur d'entrer du texte au clavier après l'avoir sélectionné à la souris, comme avec une interface graphique classique. Une barre indique que le champ est prêt à être utilisé. La taille du texte contenu est limité, il n'est possible d'effacer des caractères depuis la fin uniquement et certains caractères spéciaux ne sont pas acceptés. Ces champs sont utilisés pour entrer le nom des joueurs.

Deux autres utilitaires plus simples existent : un pour gérer les textures (chargées lors de la création du contexte graphique) et un pour écrire du texte à l'écran, permettant de gérer le style et l'alignement du texte.

5.2 Les ramasses miettes

Afin de limiter le nombre d'allocations et de désallocations par unité de temps pour les structures qui en demandent beaucoup telles que `ElementListe` et `Position`, nous avons implémenté un ramasse miettes très simple nous permettant de recycler les précédentes instances de ces types dont voici le fonctionnement. Chaque type à recycler possède son ramasse miettes.

Une ramasse miettes est configuré avec le nombre maximal d'instances recyclées stockées et la taille d'une structure à allouer. Au début, aucun pointeur n'est recyclé. Lors de l'allocation d'un élément, le ramasse miettes regarde si on a des pointeurs disponibles à recycler. Si c'est le cas, il retourne celui au dessus de la pile des pointeurs recyclés, dans le cas contraire il alloue un élément de la taille spécifiée à la création du ramasse miettes. Lors de la désallocation, si la pile d'éléments recyclés n'est pas pleine, il place le pointeur en haut de la pile et dans le cas contraire désalloue l'élément. À la fin de l'exécution du programme, tous les pointeurs recyclés restants dans la pile sont désalloués.