QUINQUENEL Nicolas - SOYER Mathieu
OUTHIER Arthur - EHRESMANN Nicolas

Rapport de projet C - Dicewars

Introduction

Ce projet a pour but de réaliser un jeu appelé Dicewars. Pour cela il a fallu suivre une structure de projet d'ores et déjà définie et coder notre programme en langage C. Nous avons utilisé un gitlab pour faciliter notre travail en groupe.
Le but du jeu Dicewars est de contrôler tout les territoire, il se joue à un ou plusieurs joueurs et avec des IA. Chaque territoire possède des dés, si il veut prendre un territoire il mise ses dés contre ceux du territoire attaqué et si il fait un score plus gros, il récupère le territoire sinon il perd tous ses dés sauf un. Chaque joueur joue à son tour, et n'a pas la possibilité d'attaquer si il n'a qu'un seul dé sur son terrain. A la fin de son tour, le joueur reçoit un nombre de dés proportionnel à la taille de son plus gros territoire, si son territoire est coupé en deux par un adversaire, le nombre de dés ne sera que lié à la plus grosse moitié de son territoire. Si le joueur n'a plus de territoire il est éliminé.
Pour réaliser ce projet nous avons dû utiliser la librairie SDL2. Elle permet d'afficher une interface graphique grâce à un renderer qui gère une fenêtre à partir de laquelle on peut afficher des images. Il est également possible de gérer des événements d'input comme un appui sur une touche du clavier ou un clic de la souris.

Explications de notre interface :

Boutons

-Bouton Valider : permet de valider le choix de la carte
-Bouton Suivant : passer à la carte suivante (aléatoire)
-Bouton Pause IA : met en pause l'IA lorsqu'elle joue, appuyer une nouvelle fois permet de relancer l'IA où elle s'était arrêté
-Bouton Nouvelle partie : Relance une nouvelle partie
-Bouton Passer le tour : Passe le tour du joueur

Attaque des territoires

Lorsque l'utilisateur sélectionne son territoire avec lequel attaquer, un contour blanc se dessine. Il disparaît s'il clique ailleurs. Si l'un des joueurs gagne son attaque, le nombre de dés change et la couleur disparaît. Des sécurités sont présentes pour éviter des mouvements interdit.

Nombre de parties

Dans le cas où il n'y a que des IA qui jouent entre elles, il est nécessaire de valider la première carte puis elles joueront jusqu'à avoir fini le nombre de parties à faire. Lorsqu'une IA gagne une carte, il y a une pause de 5 secondes avant de recommencer sur une nouvelle carte, automatiquement.

Dans le cas où il y a au moins un joueur, il est nécessaire de valider la nouvelle carte après chaque victoire.

Lorsque le nombre de partie atteint 0, le jeu se ferme au bout de 5 secondes.

Répartition des tâches

Stratégie de jeu: Mathieu SOYER et Arthur OUTHIER
Interface graphique: Nicolas QUINQUENEL et Nicolas EHRESMANN

Arborescence du projet

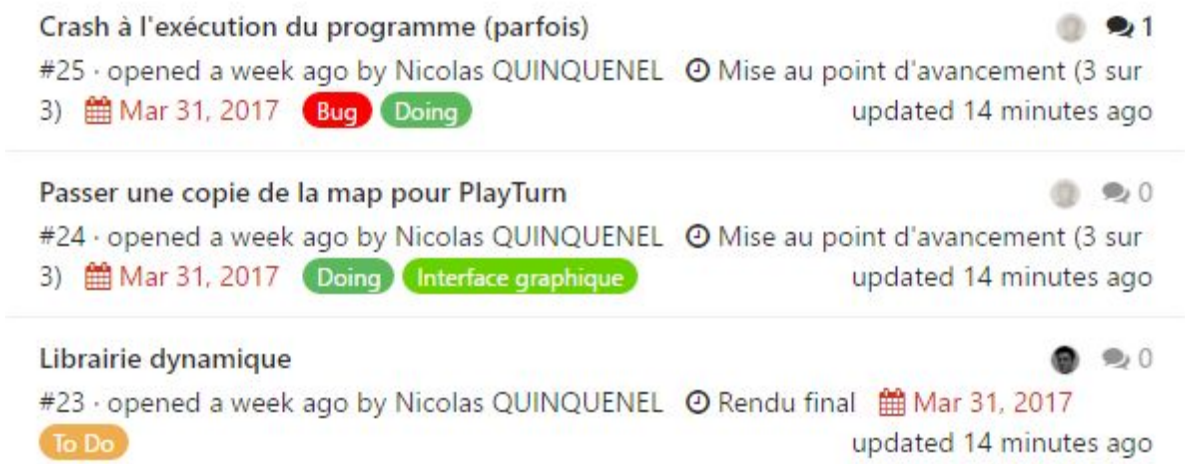
Nous avons structuré notre projet de manière à faciliter la lecture, la modification et la compréhension, nous avons donc à la racine plusieurs dossiers et fichiers qui sont les suivants:

- Le **fichier readme** qui est un fichier explicatif du Projet DiceWars il répond à plusieurs questions essentielles telles que qu'est ce que le projet DiceWars ? Comment y jouer? Comment lancer l'exécutable ? Comment compiler ce projet ? Quelle organisation a été mise en place ?
- Un **dossier Images** qui contient l'ensembles des images utilisées par notre programme principalement les images de nos dés.
- Un **fichier Makefile** qui permet de compiler l'ensemble du projet et de nettoyer les fichiers créés lors des précédentes exécutions.
- Un **fichier exécutable** appelé DiceWars.exe qui permet de lancer le jeu.
- Un **dossier src** qui contient l'entièreté de notre programme celui-ci lui même divisé en plusieurs sous parties.
    - Un **dossier Interface** contenant tous les fichiers .c et .h relatifs à l'interface
        - **fenetre.c** : fichier contenant le "jeu", c'est à dire toute l'interface
        - **fenetre.h** : fichier contenant notre librairie statique pour la fenetre
        - **free_memoire.c** : fichier contenant toutes les fonctions pour l'interface
        - **generation.c** : fichier contenant les méthodes nécessaires à la création d'un graphe
        - **generation.h** : fichier contenant notre librairie statique pour la création d'un graphe
        - **jouer.c** : fichier qui contient les fonctions nécessaires à la mise à jour de l'affichage du jeu lorsqu'un joueur sélectionne un territoire et attaque un territoire.
        - **jouer.h** : fichier contenant la librairie statique pour les fonctions de mise à jour de l'affichage.
    - Un **dossier Librairies** qui contient notre librairie statique prédéfinie avec toute la promotion ainsi qu'une librairie statique de méthodes utilisées qui sont propres à notre groupe.
    - Un **dossier Strategie** contenant tous les fichiers .c et .h relatifs à la stratégie
        - **arbitre.c** : fichier qui vérifie que l'IA fait des coups réglementaires.
        - **idéeStratégies.txt** : fichier explique brièvement les marches à suivre selon nous pour avoir une IA la plus efficace possible.

Utilisation du gitlab

Bug	Issue ayant rencontrée un bug pendant pendant son développement et/ou après sa validation
Doing	Issue en train d'être développée par le développeur assigné
Interface graphique	Développement de l'interface graphique (SDL2) et des interactions avec le jeu
Optionnel	Issue pour une fonctionnalité optionnelle
Stratégie de jeu	Développement de la partie Stratégie de jeu (IA, échanges, etc.)
To Do	Issue à faire en priorité / prochainement

En plus d'avoir réparti préalablement le travail au sein de notre équipe, nous avons utilisé le gitlab notamment pour différencier les types de problèmes à régler et leur priorité. Nous avons agencé notre planning pour pouvoir réaliser notre travail dans les temps et de manière optimale. Vous pouvez voir l'image au dessus, les différents types qui ont été utilisé.
L'utilisation de git nous a permis de travailler sur plusieurs branches simultanément, pour notre projet nous avons donc créé deux branches, une pour la stratégie et une pour l'interface.



Ci-dessus, un extrait des problèmes rencontrés.

QUINQUENEL Nicolas - SOYER Mathieu
OUTHIER Arthur - EHRESMANN Nicolas

Méthodes utilisées

Génération de la map :

Nous générons tout d'abord un nombre de points égaux au nombre de territoires souhaité, ici 50. Chacun de ces points possède une ID qui lui est spécifique.

Nous parcourons ensuite tous les pixels de la carte et nous calculons la distance de Manhattan avec chacun des points générés précédemment. Le pixel possède l'ID du point le plus proche. Lorsque chaque pixel à une ID qui lui est propre, nous pouvons définir des couleurs et des identifiants de joueurs en fonction de ces ID, par exemple, les ID (territoires) 4 9 11 et 20 appartiendront au joueur 1.

Un pixel qui possède un voisin avec un ID différent de lui devient une bordure.

A partir de ces ID, nous avons créé la matrice d'adjacence de la carte. Nous pouvons ainsi construire les cellules puis la map à l'aide de cette dernière. Les joueurs sont mélangés aléatoires au début du jeu, une IA peut ainsi commencer ou non suivant la chance.

Affichage des dés et mise à jour :

Afin d'afficher les dés, nous utilisons les textures et surface de base de SDL2, nous n'avons pas importé d'autres librairies. SDL2 n'autorise que des .bmp de base, et il est possible d'avoir une seule couleur de transparence. Le contour de nos dés est égal à 255, 255, 255 (blanc complet) tandis que le corps de nos dés est égal à 254, 254, 254. Ainsi, nous pouvons masquer la couleur 255, 255, 255 (contour) sans masquer l'intérieur.

Pour optimiser le jeu, lorsqu'un joueur attaque un territoire et qu'il gagne, seulement les 150 pixels autour du territoire de l'attaquant est remise à jour. Ainsi, la totalité de la carte n'est pas changé et nous gagnons beaucoup de temps sur l'exécution. Il est très peu probable qu'un territoire s'étende à plus de 150 pixels autour de l'attaquant, pour plus de sécurité, nous avons forcé la mise à jour de l'affichage pour l'attaquant ET celui attaqué.

Représentation d'une grappe de cellules alliées voisines:

Pour représenter une grappe de cellules alliées pour la stratégie libStratégie5.so, nous avons décidé de créer une structure SCluster qui contient comme attributs:

- id: l'id du cluster
- cells: un tableau de pointeurs vers les cellules dans la grappe
- nbCells: le nombre de cellules dans la grappe

Déroulement du jeu

Une fois la map générée, la structure permettant d'avoir le contexte de chaque joueur est initialisée de façon à avoir la plus grosse zone de territoires que chaque joueur possède, afin de pouvoir faire la distribution des dés correctement à chaque tour. Ensuite on se sert d'une boucle while pour faire tourner le jeu. Une vérification est faite pour savoir si le joueur qui commence est une IA. Si oui, alors l'IA commence son tour en utilisant la stratégie qui lui correspond. Après chaque attaque, si elle est valide, la taille de sa plus grosse zone de territoire ainsi que celle du joueur propriétaire de la zone qui a été attaquée est recalculée et l'affichage de la map est mis à jour. A la fin de chaque tour d'une IA, forcé ou non, les dés sont redistribués si nécessaire et l'id du joueur actuel est incrémentée pour signifier que l'on passe au joueur suivant. Une vérification est faite pour savoir si les attaques effectuées ont éliminées ce joueur. Si il n'a plus de dés, son id est ajoutée à un tableau contenant la liste des perdants, et on passe au joueur suivant. L'input des joueurs humains est géré via un event handler qui traite plusieurs cas grâce à un switch case. Si la croix rouge en haut à droite de la fenêtre est cliquée on ferme la fenêtre et on arrête le jeu. Si il sélectionne un territoire, le programme vérifie que le territoire lui appartient, et si c'est le cas la couleur des bordures autour du territoire pour indiquer qu'il est sélectionné. Ensuite si il sélectionne un autre territoire, on considère qu'il commence son tour et lance une première attaque. La structure STurn contenant les informations du tour en cours est mise à jour avec les ids des territoires concernés et une attaque est effectuée. Comme pour l'IA, l'affichage est mis à jour selon le résultat de l'attaque. Pour passer son tour, le joueur peut cliquer sur un bouton ou appuyer sur la touche entrée du clavier(SDLK_RETURN).

Pour l'affichage des dés, une fonction prend en paramètre les coordonnées du point définissant le territoire et le nombre de dés à afficher dessus, et en fonction de ce nombre, va appliquer une texture venant d'une image bmp pour dessiner la pile de dés représentant le nombre.

Pendant chaque tour, une chaîne de caractères est construite contenant les informations des attaques réalisées pendant le tour avec une tabulation entre chaque. A la fin du tour, cette chaîne est rajoutée dans un fichier texte appelé “logfile.txt”.

Explication stratégie IA

Stratégie optimale :

- Essayer d'avoir des grappes de 8 territoires comme le nombre maximum de dés par territoire est de 8.
- Joindre les territoires alliés: Mieux vaut avoir 1 grosse grappe que plusieurs petites. Comme il n'y a aucun moyen de choisir quels dés de renforcement ira dans le territoire qu'on veut défendre, avoir + qu'une seule zone affaiblit toute stratégie. Le premier objectif est donc de liés les territoires ensemble, non seulement pour la défense, mais puisque le renforcement de dés à chaque fin de tours est basé sur le nombre de territoires de la plus grosse grappe.
- Diviser les territoires ennemis: Inversement, diviser un territoire ennemi affaiblit considérablement son contrôle. Au début du jeu, il est sage de décider d'une base d'où se développer, et ainsi ignorer les autres territoires. Ou, si des territoires ne sont pas trop espacés, essayer de les liés.
- Ne pas passer son tour au début du jeu, ou tout du moins, pas trop longtemps.
- Si un joueur a plus de la moitié des territoires, l'attaquer et délaisser les territoires des autres joueurs, quitte à passer son tour pour gagner des dés. Mais attaquer quand même un plus faible si cela permet de rejoindre une autre grappe de territoires alliés pour gagner plus de dés à la fin du tour, ou si nos territoires ont déjà 8 dés.
- Quand on est le leader, et donc que tous les autres joueurs nous attaquent, ne pas faire trop de lignes avec peu de dés en attaquant. Sauf si tous (ou presque) nos territoires ont 8 dés.
- Ne pas attaquer si on a moins de dès qu'un territoire adverse, voir même quand on en a autant.
- Essayer de ne pas bloquer des cellules avec beaucoup de dés dans un coins

Stratégies mise en place :

libStratégie1.so

Celle-ci est la meilleure stratégie que nous avons pour le moment.
Nous attaquons premièrement les cellules avec laquelle il y a le maximum de différence de dés entre la cellule attaquante et la cellule attaquée.
Elle attaque également quand la cellule attaquante à autant de dés que la cellule attaquée.

libStratégie2.so

Cette stratégie est la même que la première stratégie, à la différence qu'elle n'attaque pas quand il y a un nombre égal de dés entre la cellule attaquante et la cellule attaquée.

libStratégie3.so

Cette stratégie est la même que la première stratégie. Quand un joueur possède plus de la moitié des cellules de la carte, alors elle ignore totalement les autres joueurs pour attaquer ce joueur. Si elle ne peut pas attaquer, elle préfèrera passer son tour.

libStratégie4.so

Cette stratégie est la même que la première stratégie. Quand un joueur possède plus de la moitié des cellules de la carte, alors elle ignore totalement les autres joueurs pour attaquer ce joueur. Si elle ne peut pas attaquer, elle préfèrera passer son tour.

libStratégie5.so

Cette stratégie aurait été la meilleure des 5, mais nous n'avons pas pu la finir par manque de temps et à cause de bugs inexpliqués.
Cette stratégie consistait à choisir la cellule attaquée pour se rapprocher d'une grappe de cellules et ainsi former une grosse grappe de cellules pour avoir plus de dés distribués à la fin du tour.

Problèmes rencontrés

Lors de notre projet, nous avons rencontré un certain nombre de problèmes notamment de fuites de mémoire, cela nous a pris du temps de savoir à quel endroit, la libération de mémoire avait été oublié, après des recherches, nous nous sommes rendus compte qu'il fallait libérer les images bmp après leur utilisation, dans un premier temps nous avons chargé les images bmp à chaque tour ce qui explique la mémoire grandissante que prenait notre programme, suite à une modification dans l'optique de régler ses fuites, la fuite de mémoire est devenue très faible, il est probable qu'une partie de la fuite de mémoire restante soit liée à l'IA.

Une autre des difficultés qui s'est présentée à nous pendant notre projet a été la séparation des classes ainsi que la définition du rôle exact de chacune des classes tel que le rôle de l'arbitre, on ne savait pas dans un premier temps si l'arbitre devait dire qui attaquait quoi ou si il se limitait à dire si le coup est valide ou non.

Et enfin un autre problème était la mise en commun des deux parties de nos codes respectifs, il a fallu faire la liaison pour les deux parties ce qui parfois s'est avéré ardu. Tout le monde n'ayant pas obligatoirement la même approche et la même idée pour faire les choses.

La non utilisation d'autres librairies telles que SDL_image nous a aussi obligé à charger nos propres images dans notre programme, nous avons considéré que toute librairie non indiquée était non utilisable au sein de notre projet.

Au niveau de la gestion mémoire, nous avons essayé de limiter un maximum les fuites mémoires, mais chaque tour en génère tout de même. Un des problème le plus important que nous avions fut à propos des images, nous avions oublier de libérer la mémoire des textures et des surfaces, cela a été une grande amélioration.

Conclusion

Notre projet a été implémenté en respectant les différentes spécificités imposées, notre approche de faire des IA relativement simples et d'avoir une organisation bien structurée nous a permis de réaliser l'ensemble du projet et de ne pas faire de ne pas faire l'impasse sur différentes parties du projet.