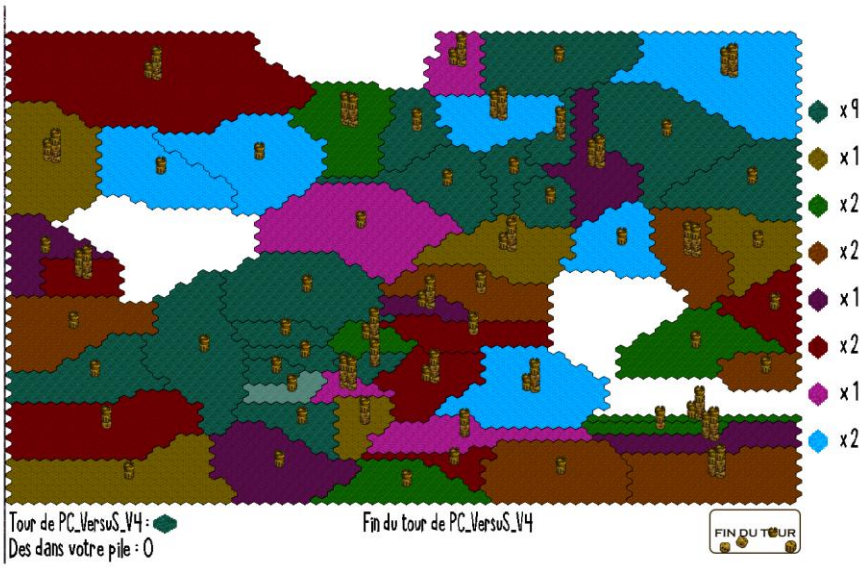
DiceWars

Simon LASSALLE, Pierre SAVATTE, Corentin MIMEAU, Vincent BRULE



Lancement du jeu et options supplémentaires

Suite au make all, il faut se déplacer dans le répertoire /RepertoireGit/Dicewars puis ./Dicewars nbPartie nbJoueurs IA/libia.so …

Vous avez différentes options réglables dans affichages suivantes expliquées ci-dessous :

-> DELAY_AFFICHAGE : s'il est à 1, les animations sont stoppées (0.3s) afin de laisser le temps à un joueur humain de voir où l'ia attaque et où sont placés les dés à la fin d'un tour. A 0, il n'y a plus ses délais.

->AVEC_AFFICHAGE : s'il est à 0, seulement l'écran de fin de chaque partie s'affiche montrant le gagnant. Si cette option est à 0, il faut mettre que des IAs car un joueur humain ne verra plus les animations.

->TOUR_ALEATOIRE : s'il est à 1, la distribution de l'ordre de jeu entre les différents joueurs est aléatoire.

Si vous désirez lancer 1000 parties avec que des IAs et avoir le résultat rapidement, il faut mettre AVEC_AFFICHAGE et DELAY_AFFICHAGE à 0 puis refaire un make all.

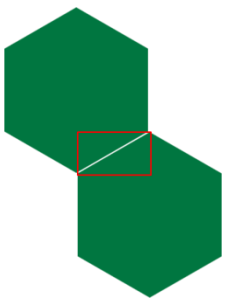
Pour le débogage et les fuites mémoires, nous avons utilisé Valgrind pour s'assurer qu'il n'y a aucune fuite mémoire.

Arbitre

L'arbitre va gérer le fonctionnement du jeu global. Il va s'occuper de faire jouer les IAs, vérifier que leurs coups sont autorisés, gérer le tour des joueurs humains et la fin d'une partie.

Nous allons expliquer les fonctions principales et surtout celle qui nous ont posé problème.

La fonction gestionEvenement() va se charger de récupérer les interactions avec l'utilisateur humain. Elle va commencer par vérifier si son clic appartient à l'un de ses territoires. Pour cette partie, un problème s'est posé à nous. Étant donné que nos sprites sont des carrés un clic entre deux hexagones pouvaient correspondre à deux hexagones(voir carré rouge dans l'image ci-dessous), c'est la que la fonction verificationCoordonnees() entre en jeu.



Cette fonction récupère les coordonnées du clic puis elle possède un switch case avec les équations de toutes les arêtes de l'hexagone, si le clic est du bon côté de l'arête, alors on sait quel hexagone est sélectionné, cela permet de corriger le problème de clic sur deux hexagones en même temps. Par la suite, elle vérifiera si le clic suivant est bien dans un territoire voisin au premier sélectionné.

La deuxième grosse fonction de l'arbitre est celle qui gère les IAs: gestionCompleteIA(). Cette fonction va commencer par vérifier que le coup reçu est autorisé, si ce n'est pas le cas, elle met fin au tour de l'IA, distribue ses dés et passe au tour suivant. Cette fonction s'occupe aussi de gérer les coups de l'IA, si c'est valide, déclencher une attaque, mettre à jour le graphe et enfin rappeler l'IA pour voir si elle a un autre coup à jouer. Le problème que nous avons rencontré dans cette partie est le fait de garder interactif la fenêtre lorsque l'IA joue. C'est-à-dire que si nous mettons que des IAs, il faut continuer à vérifier par exemple que l'utilisateur n'appuie pas sur la croix pour quitter le jeu. Pour se faire, la fonction interactionQuit() est appelée tout au long du déroulement de gestionCompleteIA() afin de vérifier si l'utilisateur n'a pas interagi avec la fenêtre.

Concernant la gestion du jeu en elle-même, nous bouclons tant que le joueur n'a pas quitté où qu'il reste des parties. En fonction du tour en cours (variable tourDeJeu) nous appelons soit l'IA soit un joueur humain. Puis à la fin de chaque tour, nous incrémentons le tour de jeu modulo le nombre de joueur pour passer au joueur suivant. Lors de la fonction qui gère la fonction fin tour, nous avons rencontré un problème pour savoir combien de territoire contigus maximum possède le joueur. La fonction maxTerritoireContigu() va explorer tous les chemins à partir d'un territoire jusqu'à bloquer sur un territoire qui n'a que des territoires ennemis, puis cela est réitéré sur tous les territoires du joueur, nous récupérons la valeur maximale. À chaque fin de tour, nous écrivons le joueur gagnant dans le fichier resultat.txt et à la fin du programme nous mettons dans resultats_finaux.txt les résultats globaux du programme par joueur.

Maintenant que nous avons décrit les points importants qui nous ont été problématiques concernant l'arbitre, passons au tableau d'hexagone qui nous a facilité l'affichage par la suite.

Le Tableau d'Hexagone

Les structures et fonctions décrites ici sont stockées dans *tableau.h* et *tableau.c*.

Nous avons choisi de représenter les territoires sous forme de groupe d'hexagones. Nous stockons tous ces hexagones dans un tableau. Plus exactement, chaque hexagone est de type Case, une Case étant une structure comprenant un id (int) et les id des six voisins (int[6]), pour faciliter certains calculs. Un Tableau est un *typedef* qui correspond en réalité à Case**, un tableau à deux dimensions de Case.

Dans un premier temps, la fonction CreerTableau() est appelée. Elle créée un tableau de taille prédéfinie dans des constantes globales et renvoie le pointeur vers l'espace alloué en mémoire. Toutes les cases sont initialisés avec l'id à -1, ce qui signifie dans la génération actuelle une case vide (non générée).

Une fois le tableau créé, il faut le peupler de territoires. Pour cela, nous utilisons une fonction GenererTerritoire(). Pour être plus précis, GenererTerritoire() est une première version médiocre de la génération, et nous utilisons actuellement GenererTerritoire2(), qui génère un diagramme de Voronoi dans l'espace 2D représenté par le Tableau. La fonction est donc relativement simple : elle choisi des germes au hasard (le nombre de germes est donné en paramètre) et parcours toutes les cases pour leur attribuer l'id du germe le plus proche.

Une autre version de la génération aléatoire du tableau pour les territoires était en cours de développement, mais l'incompatibilité entre le tableau en sortie de la génération et le tableau d'entrée de la création du graphe en a retardé l'utilisation. Au final elle a été abandonnée, d'autant plus que la version GenererTerritoire2() est efficace et le rendu dans l'interface graphique nous convient.

Le fichier contient de nombreuses autres fonctions helper, certaines utilisées pour la première version de la génération. On trouve parmi ces fonctions voisinsCase() qui renvoie les six cases voisines d'un hexagone, et d'autres fonctions de distance ou de génération de séquence de chiffre.

Le Graphe de Jeu

Conformément à l'interface commune, le graphe du jeu est modélisé par une structure SMap définie dans *interface.h.* Les fonctions décrites dans cette section se trouvent dans *graphe.h* et *graphe.c*. Ce graphe est généré à partir du tableau d'hexagones créé auparavant dans le programme.

Pour créer ce graphe, la fonction CreerMap() est utilisée. Elle s'occupe d'allouer tout l'espace nécessaire et de créer un SCell par territoire du tableau. Elle s'occupe aussi de répartir les territoires (les SCell de la SMap) équitablement entre les joueurs, et d'en laisser quelques uns au joueur 0 qui représente des territoires neutres, non jouables. Elle gère aussi la répartition des dés équitables. Nous sommes partis sur le fait que chaque territoire possède un dé puis par joueur nous distribuons 3 * (le nombre de territoire de chaque joueur) dés.

Ce faisant, elle s'assure que le graphe reste connexe, c'est-à-dire que les territoires neutres n'isolent pas une partie du graphe. Si par malchance le graphe n'est plus connexe, la fonction recommence la distribution des territoires jusqu'à ce que le graphe soit connexe. Bien que cette solution paraît peu pratique (potentiellement une génération à l'infini), en réalité il ne faut pas plus d'un ou deux essais en général pour obtenir un graphe valide. La génération paraît toujours instantanée pour un humain.

Ainsi le tableau SMap->Cells est correctement rempli, et chacune des SCell y appartenant se voit calculer et attribuer ses voisins grâce — entre autre — à la fonction voisins(), qui renvoie les id des voisins d'un territoire donné.

Pour s'assurer que les intelligences artificielles ne modifie pas le graphe de jeu elles-mêmes, la fonction copierMap() permet de copier une SMap. L'utilisation de memcpy s'étant révélé assez hasardeuse sur les pointeurs et structures que nous utilisions, nous avons préféré copier tous les champs à la main dans une SMap fraîchement allouée.

Cependant, nous avons rencontré un problème majeur en cours de création de la copie : nous utilisons les territoires avec owner = 0 comme territoires neutres. Or, l'IA n'est absolument pas au courant qu'elle doit traiter le joueur 0 différemment. Comme le graphe intègre à un niveau fondamental le joueurs 0, il nous est très difficile de créer une copie sans celui-ci. Ainsi, nous avons opté pour une solution plus simple : lors de la création de la copie, tous les voisins des territoires du joueur 0 sont supprimés et tous les voisins des territoires possédant le joueur 0 comme propriétaire sont eux aussi enlevés. Concrètement, le joueur 0 n'a plus que des territoires isolés, inaccessibles, ce qui empêche une IA de tenter de les capturer.

Un autre problème que nous avons eu lors de la répartition des owners est le fait que certain groupe commencent les owners de 0 à 7 tandis que d'autres partent de 1 à 8. Par conséquent, dans la copie que nous envoyons, nous faisons partis les owners de 0 à 7 et les territoires neutres sont de owner = -1.

Une fonction DetruireMap() se charge de détruire et libérer les ressources d'une SMap, que ce soit l'originale utilisée par l'arbitre ou une copie générée avec copierMap().

Diverses autres fonctions sont présentes dans *graphe.c*, dont notamment :

- nbTerritoiresContigus() qui renvoie le nombre de territoires contigus (de même propriétaire) pour une SCell donnée
- nbTerritoireConnexe(), quasiment identique mais renvoie le nombre de territoires contigus en dehors de ceux du joueur 0 (et donc indique si le graphe est connexe ou non, car il est connexe si le renvoi correspond au nombre de territoire non neutre)
- maxTerritoiresContigus(), qui utilise nbTerritoiresContigus() pour donner le nombre maximal de territoires contigus d'un joueur et qui est utilisé pour la distribution des dés en fin de tour

On notera de plus une fonction helper qui vérifie si un élément est dans un tableau, et deux fonctions permettant d'afficher le graphe dans la console sous forme de texte (liste des cellules et de leurs voisins).

L'IA

La stratégie de l'IA est d'affaiblir le meilleur joueur ou si c'est elle, éliminer la concurrence. Avant le début de la partie, elle explore le graphe (avec les fonctions déjà utilisées dans le moteur du jeu) et regarde quel est le meilleur joueur : l'indicateur est le nombre de territoires connexes, ce qui correspond au nombre de dés que le joueur récupérera au tour suivant, c'est à dire la puissance supplémentaire qui lui sera attribuée. Le but de notre IA est donc de devenir la plus puissante.

A chaque tour l'IA explore chaque cellule lui appartenant et trouve toutes les cibles potentielles de cette cellule, seulement si la cellule n'est pas en danger (c'est à dire que la moyenne de dés autour de la case est inférieur au nombre de dés de la case). Les cellules adverses sont ciblées si elles sont plus faibles (écart de 2 en faveur de l'IA) que la cellule de l'IA. Seulement, l'IA possède une seconde vision : Lorsque l'IA n'a pas attaquée, elle devient plus agressive et l'écart de dés des cellules entre l'IA et l'adversaire est réduit à 0.

Si l’IA possède une pile de dés importante, dans ce cas elle ne vérifie plus et attaque (elle entre dans ce mode lorsque la partie est à la fin car au début sa pile est à 0).

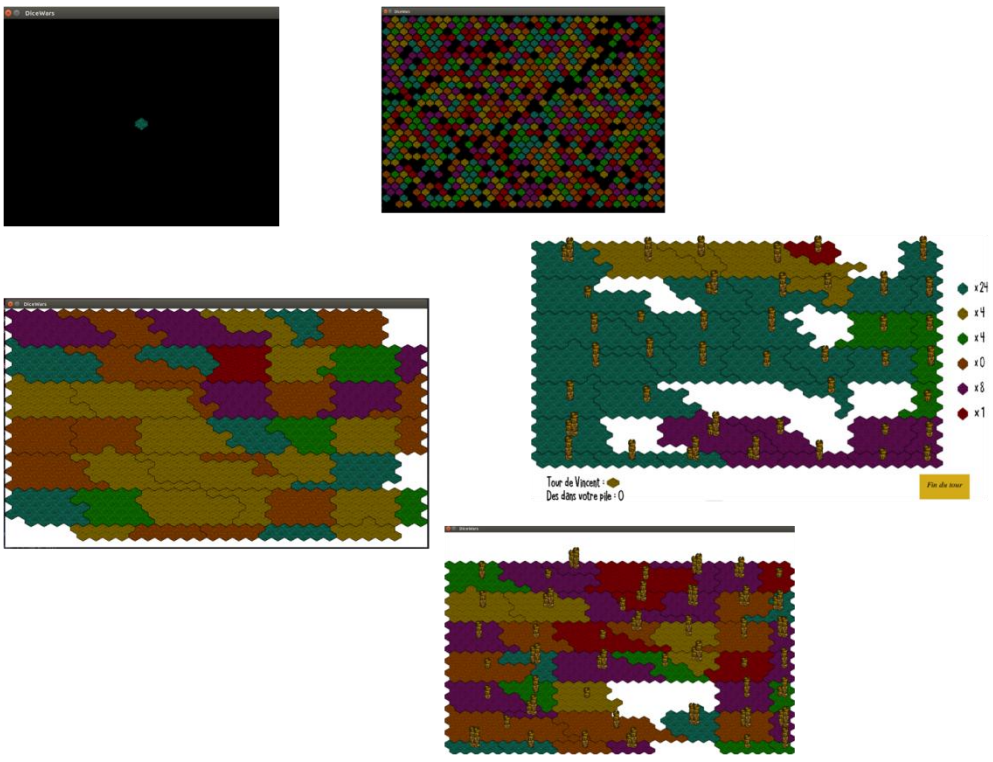
A la fin de la sélection des cibles elle calcule laquelle est la meilleure en regardant si sa “cible préférée” fait partie des propriétaires des cellules. Si ce n’est pas le cas elle prendra la cellule la plus forte afin de maximiser les dégâts fait aux adversaires.

Le nom de l'IA n'a pas de signification particulière. PC_VersuS a été trouvé en assemblant les premières lettres de nos prénoms: Pierre, Corentin, Vincent et Simon. Il a le bonus d'être intimidant, mais cela n'est efficace que contre des joueurs humains malheureusement.

Les graphismes

Nous avons imaginé, pour rendre notre jeu plus agréable à jouer, des graphismes dans un style médiéval. Tous les sprites de l'affichage ont été réalisés sous Photoshop. Pour alléger le temps de chargement des images, nous les avons faites directement de petite taille (environ 30 pixels de côté). Avec cela nous est venue l'idée de les faire en pixel art. Nous avons aussi décidé de garder les deux principes visuels du jeu de base, à savoir la représentation des territoires avec des hexagones (c'est ce qui nous a conduit à créer la map sous forme d'un tableau dont les cases ont six voisins), et de représenter les dés empilés les uns sur les autres.

Concernant l’implémentation des graphismes, nous avons choisi d’utiliser la version 2 de SDL simplifiant des mécanismes d’affichage. Notre premier problème a été d’appréhender cette librairie qui nous était inconnu. Nous sommes passés par de nombreuses étapes (voir image ci-dessous). Le résultat final étant la photo de la page de couverture.



Nous avons choisi d'utiliser un renderer car c'est un des plus de la SDL2 donc au temps en profiter.

La fonction affichagemap() va blitz tous les éléments de la map puis va mettre à jour le rendu. Concernant l'affichage des dés ou des hexagones, nous avons un tableau de SDL_Rect afin de calculer une fois leur position puis à chaque affichage nous réutilisons ce tableau afin de savoir où blitz. La fonction la plus problématique a été celle qui affiche les frontières entre les territoires. Pour ce faire, nous calculons les voisins de chaque case et si nous trouvons un owner différent d'elle-même nous traçons un trait noir. La position des traits a été calculée directement à partir de la taille de l'image.

Pour l'affichage des différents textes comme le résultat d'une attaque, nous avons plusieurs chaînes de caractère représentant les textes à afficher puis grâce au renderer nous créons une surface contenant le texte puis on la blitz. Un problème qui est apparu était le fait d'avoir une fuite mémoire assez importante, suite a des recherches, nous nous sommes rendu compte que les surfaces étaient chargées sans être détruite, par conséquent nous utilisons DestroyTexture() fourni par SDL2 afin de les désallouées à chaque fois.

Lorsque la partie se déroule avec un joueur humain, nous mettons à jour l'affichage à chaque fois que le joueur clic afin de ne pas mettre à jour l'affichage en boucle inutilement.

Pour le reste de l'affichage, c'est toujours le même mécanisme, nous regardons les sprites qui ont été chargés au début du programme, on récupère la position où ils doivent être placés puis on blitz et on met à jour l'affichage quand tout est blitz.

Cette partie n'a pas été compliqué en soit mais il nous a fallu comprendre les subtilités de la SDL afin de faire le programme le plus optimisé et le plus joli possible.

Makefile

Un Makefile permet de compiler rapidement le projet. Les commandes suivantes sont disponibles :

- "make DiceWars" pour compiler le jeu arbitre
- "make libIA" pour compiler l'IA
- "make all" pour compiler les deux
- "make clean" pour nettoyer les fichiers obj
- "make distclean", qui fait un clean puis supprime en plus les résultats de la compilation (les fichiers DiceWars et libIA.so)

L’utilisateur du Makefile a besoin d’avoir SDL2, SDL2_image et SDL2_ttf d’installés sur son ordinateur pour pouvoir compiler. Si les bibliothèques ci-dessus ne sont pas installées, “# apt-get install libSDL2-*” installera tous les paquets nécessaires sur Debian et ses sous-distributions.

Le Makefile est en grande partie inspiré de celui présent sur Madoc, adapté pour compiler les deux parties du projet avec les bons arguments.

Lors de la création de ce Makefile, nous nous sommes heurté à un problème : modifier un .h ne déclenchais pas une re-compilation. Pour palier à ce problème, notre solution est de forcer un distclean avant les cibles DiceWars et libIA, cependant cette solution recompile tout le projet à chaque fois, même quand aucun fichier n’a été modifié. La compilation du projet étant rapide, nous n’avons pas jugé cela très gênant.

Conclusion

Tous les membres du groupe appréciant les jeux vidéo, en créer un a été agréable. Le point le plus intéressant, et nouveau qui plus est, a été l'interface commune à tous les groupes. Cela nous permet de mieux imaginer le travail en équipe dans une grande entreprise qui nous fixe des contraintes et un cahier des charges. Ainsi la compétition des IA entre les différents groupes a été un point stimulant durant le projet.

Nous avons aussi pu améliorer nos compétences dans l'utilisation de git, dans l'apprentissage en autonomie lorsque nous utilisions de nouvelles librairies, notre communication lors de la résolution de problèmes, et enfin dans le développement dans le langage C.

C'est une bonne expérience à mettre dans nos CV, et le résultat nous plaît. Nous continuerons sûrement à développer des fonctionnalités pour le jeu, comme des animations, et pourquoi pas un multijoueur en ligne !