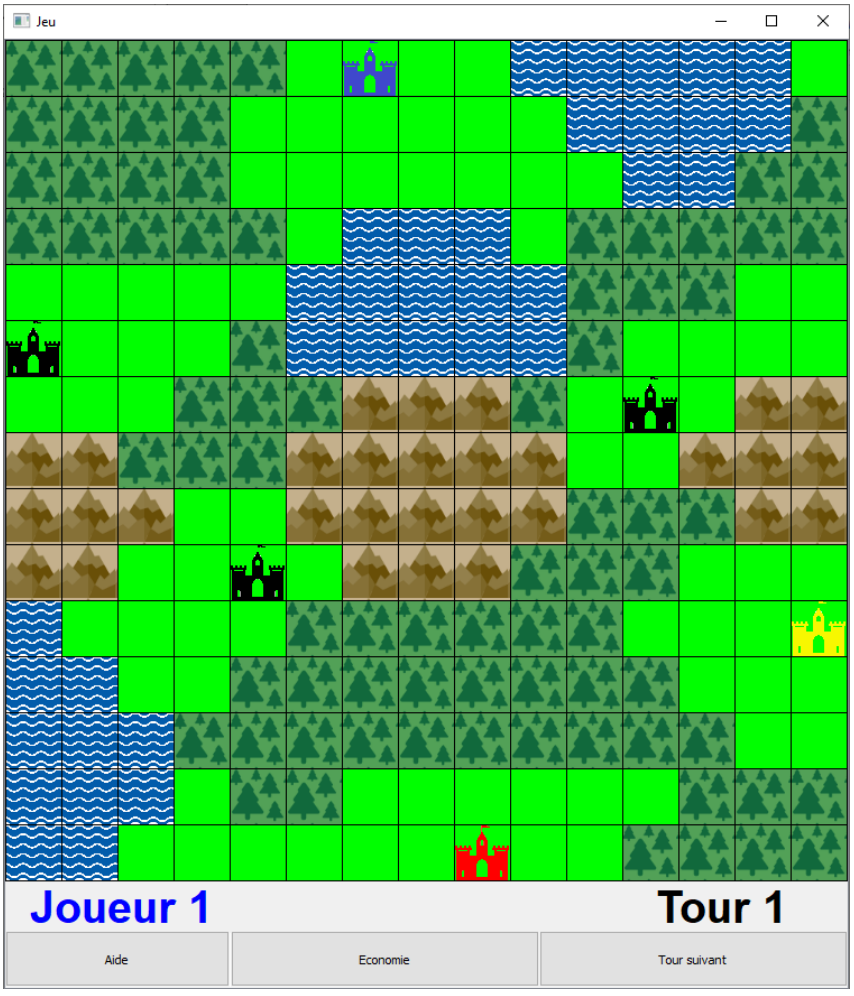
Projet C++

KillAndConquer



Projet C++

KillAndConquer

I. Introduction

Dans ce projet réalisé en C++, nous avons développé à l'aide de l'application QT un jeu de plateau tour par tour dans lequel peuvent s'affronter 2 à 4 joueurs. Ce jeu se nomme ***Kill & Conquer***, le principe du jeu est le suivant : chaque joueur aura au premier jour une ville ainsi qu'une économie de 20 pièces. Il est alors libre de créer l'unité de son choix. C'est seulement au tour suivant que les joueurs pourront bouger leurs unités. Pour gagner un joueur devra soit conquérir le plus de villes possibles dans les 100 tours impartis, soit capturer toutes les villes de ses adversaires. Pour arriver à son but, le joueur pourra créer, bouger ses unités ainsi que combattre avec contre les unités adverses. Pour améliorer leurs économies, les joueurs devrons capturer des villes. Ces villes permettront aussi de créer des unités. Il est important de noter qu'une ville ne pourra créer qu'une unité par tour. De même une unité ne pourra se déplacer qu'une seule fois par tour et ensuite attaquer si elle a la portée nécessaire. Si une unité tue trois unités alors elle regagne tous ses points de vie (seulement une fois). On peut dans certains cas fusionner deux unités (pour plus de détails voir la notice). Enfin le joueur pourra passer sa souris au-dessus d'une unité ce qui fera apparaitre une pop-up contenant la vie de cette dernière. Pour connaitre toutes les règles et astuces pour mener à bien votre partie, référez-vous à la notice de jeu ci-jointe.

Il existe dans le jeu quatre types d'unités : l'infanterie, les cavaliers, les archers et les archers montés (des archers sur des chevaux). Chaque unité possède un rayon de déplacement et une portée d'attaque qui lui est propre. Par exemple les cavaliers et cavaliers archers peuvent aller plus loin qu'une infanterie. De même, les archers et cavaliers archers ont une portée d'attaque supérieure à celle de l'infanterie.

Il existe aussi quatre types de terrain : la plaine, la forêt, la montagne et l'eau. Il n'est possible d'aller sur les cases de type montagne que pour les unités d'infanteries et les archers. Et de façon plus générale, aucune unité ne pourra se déplacer sur une case de type eau.

Chaque unité aura des bonus selon le terrain sur lequel elle se trouve, en effet chaque terrain peu conférer des bonus ou malus de déplacement et de dégâts aux unités. Par exemple personne ne peut se déplacer sur l'eau, les chevaux et archer montés sont rapides en plaine tandis que les archers et infanteries sont plus rapides en forêts et peuvent aller dans la montagne. De plus, par exemple, les attaques de l'infanterie et des cavaliers font plus de dégâts dans les plaines, car cet environnement leur est favorable.

Enfin les villes permettent de créer les unités (une par ville et par tour), si une unité est sur sa ville alors un bonus de défense lui est attribué.

II. Réalisation du projet

Notre projet est divisé en 3 parties : la partie *data*, la partie *logique* et la partie *graphique*. La partie *data* représente toutes les classes (unités, villes, cases…) que la partie *logique* a besoin pour simuler le jeu.

Lors du lancement d'une partie la partie *logique* va générer avec les éléments de la partie *data* le terrain de jeu (logique), pour pouvoir effectuer ses calculs. En parallèle la partie *graphique* affichera le terrain de jeu (visuel). Le terrain dans la partie *graphique* et *logique* a donc le même état à tout instant du jeu. Nous sommes partis du principe que la partie *logique* faisait confiance à la partie *graphique*.

La partie *logique* sert d'interface avec la partie *graphique*. En effet la partie *graphique* pour chaque évènement détecté va effectuer une série d'instruction et appeler des méthodes de la partie *logique*. Puis la partie *logique* fera ses calculs, modifiera ou non ses données puis elle dira à la partie graphique que faire. Par exemple, après l'appuie sur l'unité par l'utilisateur, la partie graphique va demander à la partie logique si l'unité peut attaquer, puis la partie logique lui renverra la réponse. Ensuite, si la réponse est positive, elle demandera à la partie *logique* les coordonnées où l'unité peut attaquer. Pour pouvoir s'échanger des informations, la partie *logique* et *graphique* se basent sur les coordonnées des cases de leurs terrains respectifs, qui sont liées.

Si on se concentre sur la partie *graphique*, nous aurons dans un premier temps la création de l'environnement de jeu. Pour cela le programme va lire un fichier texte (fourni par la partie *logique*) dans lequel nous avons détaillé la map sous forme de texte. Ensuite, lors de la lecture de chaque élément dans le fichier texte correspondant à une case de la map, nous allons la créer graphiquement : pour cela nous appelons le constructeur *Cases* qui est la redéfinition d'un *QGraphicsRectItem.* En effet on définit les caractéristiques de notre case pour en faire des carrés de taille 50 par 50 dans lequel on met une image de fond correspondant à l'élément lut dans le fichier (cela peut être soit une case de type eau, plaine, montagne ou forêt ou alors une ville). Pareil lorsque nous créons une unité via le shop, nous créons un carré de 50 par 50 dans lequel nous mettons une image au format png afin de préserver la transparence et ainsi voir sur quelle case nous nous trouvons. Nous ajoutons aussi différents attributs tels que *setZValue* à 1 pour que notre unité soit toujours positionnée au-dessus des autres éléments graphique (cela a son importance lorsque nous souhaitons capturer une ville puisqu'il faut alors créer une nouvelle case or par défaut sur QT l'élément graphique qui se trouve sur le dessus est le dernier élément créé). Il faut aussi pouvoir déplacer notre unité, donc on ajoute des flags lors de la création de l'unité comme *ItemIsSelectable* et *ItemIsMovable.*

Ensuite, le rôle de la partie *graphique* est de faire le lien entre l'utilisateur et la partie *logique*. En effet, lorsque l'utilisateur va sélectionner une unité pour la bouger, nous allons appeler la partie logique pour savoir si dans un premier temps si l'unité appartient bien au joueur en train de jouer, puis si cette condition est validée nous allons appeler une méthode permettant d'afficher différents indicateurs afin de savoir où nous pouvons déplacer l'unité. A ce moment, nous appelons une nouvelle fois la partie *logique* qui comme nous le verrons par la suite va renvoyer une liste de coordonnées où afficher l'indicateur. Lors du relâchement de l'unité sur n'importe quelle case on va alors vérifier avec la partie *logique* que l'unité est sur une case valide. Si l'unité ne l'est pas ou que l'unité n'a pas le droit de bouger alors on repositionnera l'unité à sa précédente position.

La dernière fonction principale de la partie graphique va être de proposer différentes interfaces telles qu'une page pour donner l'économie actuelle du joueur (atteignable avec le bouton « économie ») mais aussi de proposer une boutique. En effet le programme est conçu de sorte que lorsque l'on clique

sur une ville, on appel la partie *logique* pour savoir si la ville en question appartient bien au joueur actuel et si elle n'a pas encore créé d'unité. Si oui, alors on affichera une fenêtre permettant d'acheter des unités mais aussi d'afficher l'économie actuelle du joueur. Quand un joueur va cliquer sur une unité afin de l'acheter, on va alors vérifier auprès de la logique que l'économie du joueur est suffisante avant de créer l'unité de façon *logique* et *graphique*.

Ensuite coté *logique* et *data* nous avions au préalable commencé par réaliser un diagramme de classe (à retrouver ci-joint), nous y sommes restés à peu près fidèle, la partie game et level ont été fusionnées en une seule partie, la partie *logique*, le reste des classes appartiennent à la partie *data*. Nous avons également rajouté plusieurs méthodes. Les plus grosses fonctions se trouvent dans la partie logique, voici les plus importantes.

La fonction *getUnitMoveCoords* prend en paramètre les coordonnées d'une unité et renvoie les cases sur lesquelles elle peut se déplacer. Pour cela on calcule le rayon de déplacement de l'unité avec son propre attribut, auquel on rajoute le bonus conféré par la case. Puis on parcourt chaque case dans ce rayon. Une case est valide si aucune unité (alliée ou ennemie) ne se trouve dessus et si la case est atteignable (cad que de l'eau ne barre pas le chemin ou une montagne pour certaines unités), de plus si on change de type de case et que le bonus conféré est différent alors on ne rentre que d'une case dans cette région. Pour faire cela j'utilise plusieurs sous fonctions dont une fonction récursive (*recCaseBetweenBonusUnitIsSame*) afin de savoir si un chemin existe entre une case et la case de l'unité. Chaque coordonnée de case valide est insérée dans un vecteur puis le vecteur est renvoyé (à la partie *graphique*).

Il existe des fonctions similaires pour obtenir les cases sur lesquelles l'unité peut attaquer et sur lesquelles elle peut fusionner.

La fonction *attack* permet de calculer les conséquences d'une attaque entre unités. Les dommages donnés par l'unité dépendent de son ratio vie sur vie maximale, de son attribut dégât, du bonus conféré par sa case et de si l'adversaire se trouve sur une ville (pour plus de détails voir la notice). Ensuite on applique les dégâts, si l'unité adverse est morte et que l'unité peut prendre sa place, l'unité prend sa place (l'unité ne prend pas sa place si elle tire avec un arc (archers ou archers montés) ou si elle ne peut pas aller sur la case (montagne)). Si l'unité adverse n'est pas morte alors elle réplique (si elle tue son adversaire elle ne prend pas sa place). Pour remonter toutes ces informations à la partie graphique on utilise un système de flags (drapeaux), auquel on associe deux couples de coordonnées (a et b).

Flag	0	1	2	3
Signification	Bouge sur a	Bouge sur a enlève b	Enlève a	Enlève b

La fonction *attack* est elle-même une sous fonction de *moveUnit*, cette fonction se base sur le même système de flags. Elle permet de dire à *graphique* comment bouger une ou deux unités, et donc d'afficher le résultat de déplacements, d'attaques ou de fusions.

La fonction *nextPlayer* permet de passer au joueur suivant. Premièrement on regarde si le joueur suivant possède encore des villes ou des unités, sinon on le passe et on test le suivant. Ensuite on parcourt tout le tableau de cases pour réinitialiser tous les flags associés aux unités et cités (pour savoir s'ils peuvent bouger, attaquer, créer des unités). On regarde si une unité est sur une ville depuis un tour, si c'est le cas on capture la ville. Enfin on incrémente aussi l'économie du prochain joueur, en effet chaque ville rapporte un certain montant de pièces (5 par ville) par tour.

III. Objectifs

Tout au long du projet, que ce soit dans la partie *logique* ou *graphique*, nous avons utilisé les **vectors** de la bibliothèque STL, mais aussi des **tuples** et des **QPair**. Pour parcourir les vecteurs nous avons utilisé les itérateurs. Par exemple dans la partie logique (logic.cpp) le tableau de cases est un double vecteur (Création du tableau : fonction *CreateMap* l.61. Parcourt du tableau : Fonction *Readmap* l.585 ou fonction *nextPlayer* l.1190)

On utilise aussi des **exceptions** dans logique et graphique pour éviter d'avoir un vecteur null ou un dépassement de range dans un vecteur (dans logic.cpp, fonction *getCityOwner* l.689). Si on trouve une erreur on throw l'erreur (soit on a une exception déjà définie, soit ont créé notre erreur, grâce à notre classe *vect_error*). On utilise à plusieurs endroit cette fonction, par exemple dans logic.cpp à la fonction *getCityColor* l.717 on catch les erreurs.

Nous avons sur défini deux **opérateurs** dans la classe unit.cpp, l.241 et l.247. La sur définition de l'opérateur '==' permet de comparer deux unité grâce à leur propriétaire et leur type. La surdéfinition de l'opérateur '+' permet de fusionner deux unités, on fait la somme de leurs points de vie et on supprime la deuxième. On utilise l'opérateur '==' dans la fonction *getUnitMergeCoords* l.959 (logic.cpp). On utilise l'opérateur '+' dans la fonction *moveUnit* l.1145 (logic.cpp).

On utilise l'**héritage** multiple (héritage en diamant) pour créer la classe *MountedArcher* (MountedArcher.cpp/.h) qui hérite d'Archer et *Knight* qui eux-mêmes héritent de Unit.

On utilise également des **dynamic_cast** pour caster des pointeurs de type Building* en type Case*. On peut en retrouver à différents endroits dans notre code, notamment dans logic.cpp (l.488, l.550, l.759, l.1036 …).

IV. Conclusion

Pour la réalisation de ce projet, nous avons répartis les tâches de la manière suivante : naga-1 s'est chargé de la partie *data* et *logique* du programme alors que T. L. a réalisé la partie *graphique*. Nous nous sommes accordés sur les informations que nous devions nous échanger afin de faciliter la relation *logique/graphique*. Nous avons rencontré quelques difficultés durant ce projet notamment pour prendre en main QT, car en effet la partie graphique de QT était totalement nouvelle pour nous et nous avons dû tout apprendre en regardant notamment plusieurs tutoriels sur internet. De plus, nous avons eu du mal à mettre en place un double héritage utile.

Pour améliorer notre projet nous souhaiterions implémenter une vision particulière pour chaque joueur qui ne pourra donc contrairement à maintenant voire seulement les parties de la carte qu'il a découvert en cachant donc les positions ennemies s'il n'a pas la vision dessus grâce à ses unités. Cette partie est déjà incluses dans les classes de la partie *data*. Nous souhaiterions aussi implémenter au lancement du jeu un choix du nombre de joueurs et de cartes qui permettrait de jouer plus facilement sur différentes cartes. Avec un peu plus de temps cela aurait été notre prochain objectif, car la classe *mainGame* est généralisable très facilement.

Pour conclure, nous pouvons dire que nous sommes ravis et fier de ce que nous avons réalisé puisque nous ne savions pas faire cela auparavant et nous avons donc appris beaucoup de choses. Nous avons maintenant des connaissances sur la programmation objet en C++ (et les pointeurs surtout !) mais aussi de la programmation avec QT qui apporte beaucoup de possibilités.