Projet Clash Loyal version2024

Présentation

Nous allons réaliser un remake très simplifié d'un jeu sur téléphone portable : « Clash Royale » de SuperCell.

Le jeu consiste à voir avancer ses unités vers le camp adverse et lorsque celles-ci sont à portée d'une unité ennemi : elles engagent le combat. Vous trouverez beaucoup de vidéos sur youtube pour vous donner une idée du jeu de base « Clash Royale ».

Le jeu s'arrête quand la tour du roi d'un des joueurs est détruite.

Notre jeu sera automatique (ordinateur vs ordinateur).

Une interface graphique vous sera fournie (sur le même principe que le mini-projet 2048), **votre travail est de produire le code qui régit le jeu** (version simplifié) tel que défini dans ce sujet. **Les types sont imposés.**

Le type Tunite

Une unité pourra être à la fois une tour, la tour du roi, un archer, un dragon, un chevalier ou une gargouille. Ce sont les valeurs des champs qui différencient les unités et le champ « nom », de type « TuniteDujeu » :

```
typedef enum{tour, tourRoi, archer, chevalier, dragon, gargouille} TuniteDuJeu ;
```

Les unités peuvent attaquer une cible au sol, en l'air ou les deux, selon le type Tcible :

```
typedef enum{sol, solEtAir, air } Tcible ;
```

Le type Tunite est une structure :

```
typedef struct {
    TuniteDuJeu nom;
    Tcible cibleAttaquable;      //indique la position des unités que l'on peut attaquer
    Tcible maposition;           //indique soit « air » soit « sol », utile pour savoir
                                 //qui peut nous attaquer
    int pointsDeVie;
    float vitesseAttaque;        //en seconde, plus c'est petit plus c'est rapide
    int degats;
    int portee ;                 //en mètre, distance sur laquelle on peut atteindre une
                                 //cible

    float vitessedeplacement;    //en m/s
    int posX, posY;              //position sur le plateau de jeu
    int peutAttaquer;            //permet de gérer le fait que chaque unité attaque une
                                 //seule fois par tour ;
                                 //0 = a déjà attaqué, 1 = peut attaquer ce tour-ci
                                 // à remettre à 1 au début de chaque tour
    int coutEnElixir;
} Tunite;
```

Les joueurs

Les joueurs seront vus comme des listes (simplement chainées dans la version de base, ou via votre bibliothèque de listes faite en TP, avec un mécanisme d'encapsulation) d'unités (Tunite) de type TListePlayer.

```
typedef struct T_cell{
    struct T_cell *suiv;
    Tunite *pdata; //pointeur vers une unité
} *TListePlayer;
```

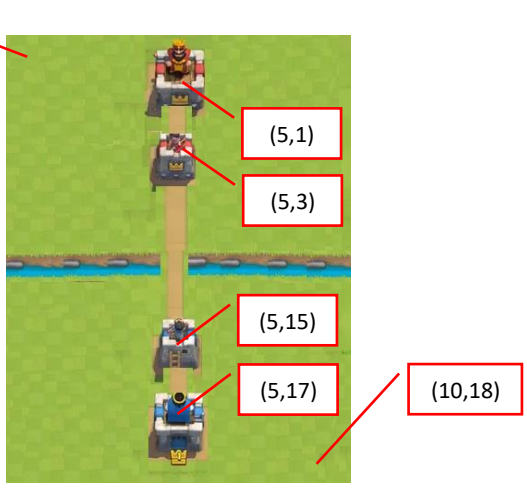
Le plateau du jeu

Le plateau du jeu est un tableau a deux dimensions contenant des pointeurs vers des unités.

```
typedef Tunite* **TplateauJeu ;
```

Le plateau du jeu aura 11 colonnes et 19 lignes. Une tour du roi et une tour défensive par camp feront partie de la liste des unités de chaque joueur. Les unités (autres que les tours) apparaissant juste devant la tour de leur roi (on ne choisira pas le lieu d'apparition de façon interactive) et elles emprunteront un unique chemin central pour aller les unes vers les autres.

Voici un aperçu de ce que pourrait donner l'interface de notre jeu, avec les coordonnés de certains points utiles :



Le plateau du jeu (un tableau 2D contenant des pointeurs vers des unités donc) sera reconstruit à chaque tour du jeu grâce aux deux listes d'unités des joueurs. Chaque case du plateau sera initialisée avec la valeur NULL (absence d'unité par défaut). Le tableau du jeu ne sert que de base pour gérer l'affichage du jeu (en mode console ou en mode graphique). Les combats seront gérés via les listes des unités de chaque joueur.

Chaque unité ayant ses coordonnées (posX et posY), il suffit d'aller à ces coordonnées dans le tableau du plateau pour y écrire l'adresse vers l'unité en question (à la place de NULL).

Voici un exemple de ce qui pourrait se passer en termes de code pour la corrélation du plateau avec les listes des unités des deux joueurs :

```
TplateauJeu  jeu ;
Jeu = AllogeTab2D(11,20) ;  //malloc dans cette fonction…
initPlateauAvecNULL(jeu) ;   //inscrit une valeur NULL dans chaque case du plateau
(absence d'unité)

PositionnePlayerOnPlateau(player1,jeu) ; //réalise le lien entre le tableau et chaque
unité : jeu[x,y]=unitecourante-> pdata ; player1 étant une liste d'unités
PositionnePlayerOnPlateau(player2,jeu) ;  //idem pour le joueur 2
```

```
affichePlateauConsole(jeu,11,20) ; //un affichage console du jeu (à coup de printf)
```

Déroulement d'une partie

Une boucle principale while (fournie dans le squelette du projet code-block) tourne jusqu'à la fin de la partie, qui se déclenche par la destruction d'une tour du roi d'un des deux joueurs.

Une Fonction utile, à se coder sans doute :

```
Bool tourRoiDetruite(TListePlayer player);
```

A chaque tour :

- **Phase combat**, qui se décompose en :
  - Détection de qui peut taper sur qui (qui est à portée). La fonction suivante est à coder, et **à appliquer à chaque unité de chaque camp** :

```
TListePlayer quiEstAPortee(TListePlayer player, Tunite *uneUniteDeLautreJoueur) ;
```

La fonction `quiEstAPortee` crée et renvoie la liste des unités qui peuvent attaquer `uneUniteDeLautreJoueur`

  - Optionnel : tri de la liste pour déterminer qui tape en premier (paramètre `vitesseAttaque` des unités).
  - Attaques (soustraction des dégâts aux points de vie) des unités **de** la liste générée par `quiEstAPortee` **sur** `uneUniteDeLautreJoueur` , en vérifiant si chacune peut encore attaquer ce tour-ci (champ `peutAttaquer`). Voici l'entête :

```
TListePlayer combat(TListePlayer player, Tunite *uneUniteDeLautreJoueur) ;
```

  - Vous aurez besoin de supprimer une unité (quand elle a 0 point de vie) :

```
Void supprimerUnite(TListePlayer *player, Tunite *UniteDetruite) ;
```

Si on supprime la dernière unité d'un joueur, sa liste vaut NULL (donc obligation de passer un pointeur sur cette liste qui peut devenir égale à NULL).

- **Phase déplacement** : On n'acceptera qu'une seule unité par case du plateau de jeu. Si la case de destination est déjà occupée, alors l'unité avance d'une case en moins.

- **Phase création** d'une nouvelle unité par camp. On va appliquer une probabilité qu'une création ait lieu : 50%. Ceci en plus du coût en élixir pour faire naître chaque unité.
  Nous aurons donc besoin de mémoriser la quantité d'élixir de chaque joueur :
  `int elixirPlayer1, elixirPlayer2 ;`
  Comme toutes les unités n'ont pas le même coût, il faudra faire un tirage aléatoire pour déterminer quelle unité va être ajoutée, et seulement si le joueur a assez d'élixir et cette action ne se fera qu'avec une probabilité de 50% (on n'est pas obligé de dépenser tout de suite son élixir). Vous êtes libre de proposer vos fonctions pour gérer si un camp fait apparaître une unité et si oui laquelle (random parmi celles que l'on peut payer avec l'élixir possédé par le joueur). Au final vous fournirez `AcheteUnite` qui retourne l'unité achetée ou NULL si pas d'achat, et décrémente le prix en élixir à `elixirEnStockduJoueur`:

```
Tunite AcheteUnite(int *elixirEnStockduJoueur) ;
```

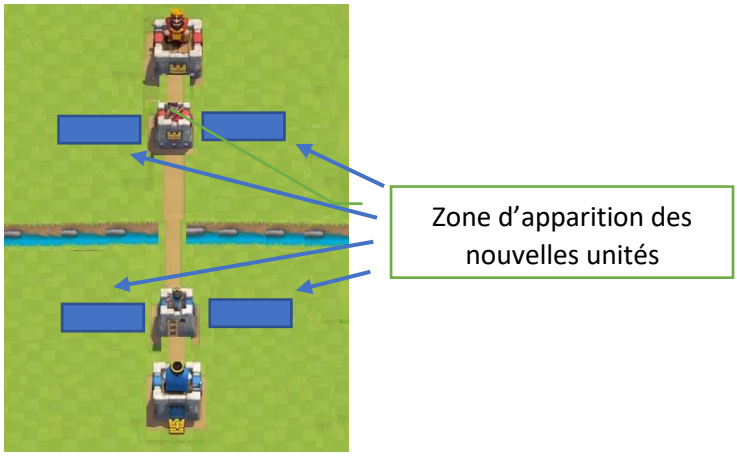
Vous aurez besoin d'une fonction pour ajouter cette nouvelle unité :

```
Void AjouterUnite(TListePlayer *player, Tunite *nouvelleUnite) ;
```

Remarques : on n'ajoutera jamais d'unité dans une **TListePlayer** vide, car ce joueur serait donc détruit et la partie terminée. Par ailleurs, ajouter une première unité revient à donner une adresse à la liste (celle de la première unité justement), d'où l'obligation de passer l'adresse de la liste en paramètre.

Où apparaissent les unités ?

Vous êtes libres de choisir où vous ferez apparaître les nouvelles unités. Par défaut, elles peuvent « naître » sur la ligne de la tour défense, cad sur sa gauche ou sa droite puis elles doivent converger vers le chemin central (mouvement en diagonal).



- **Phase élixir** : une quantité d'élixir aléatoire sera attribuée à chaque camp, entre 1 et 3 (vous pourrez changer ces valeurs pour améliorer les parties selon vos avis).

- **Phase Affichage du jeu**:
  Vous appellerez votre fonction `void affichePlateauConsole(TplateauJeu jeu, int largeur, int hauteur)` pour un affichage dans la console.
  Les routines d'affichage « graphique » suivantes doivent être appelées à chaque tour :

```
efface_fenetre(pWinSurf);
prepareAllSpriteDuJeu(jeu,LARGEURJEU,HAUTEURJEU,pWinSurf);
maj_fenetre(pWindow);
SDL_Delay(300);
```

Métrique du jeu

Par défaut, on fixera 1m = 1case du plateau.

Point de départ du projet

Un projet CodeBlock « projetClashLoyal » vous est proposé comme point de départ.

Si vous ne voulez/pouvez pas coder sous codeblock vous n'aurez pas accès aux affichages liés à la SDL (affichage graphique). Vous pourrez alors simplement faire un affichage dans la console (printf).

Pour que la compilation avec la SDL se passe bien, vous devez utiliser codeblock 20.03 et avoir indiqué que minGW était dans le dossier de codeblock 20.03 (lire et appliquer la doc sur updago « Installation nécessaire pour la SDL2 (pre-requis mni Projet et projet ClashLoyal) »).

Le projet projetClashLoyal contient déjà les types demandés et certains prototypes de fonctions.

Vous serez amené à créer d'autres fonctions **dans** les fichiers **clashcoyal.h** et **clashcoyal.c (selon vos besoins)**, que vous utiliserez dans main.c.

Travail demandé

Vous devez coder le noyau du jeu (hors interface graphique donc) et faire en sorte que deux « joueurs type ordinateur» jouent automatiquement.

Vous devez proposer également une sauvegarde du jeu (et la restauration d'une partie sauvegardée) via :

- Un fichier binaire, que l'on nommera « partiebin.clb »
- Un fichier séquentiel que l'on nommera « partieseq.cls »

Une documentation « fichiers binaires et fichiers séquentiels en C » sera ajouté sur updago pour vous aider.

La boucle while prévoit la lecture de quatre touches :

- « s » pour lancer la sauvegarde dans « partiebin.clb »
- « c » pour charger le fichier de sauvegarde « partiebin.clb »
- « d » pour lancer la sauvegarde dans « partieseq.cls »
- « v » pour charger le fichier de sauvegarde « partieseq.cls »

Travail en binôme

Vous devez faire ce projet en binôme (pas d'exception).

Vous disposez de trois semaines de TP, d'une semaine au-delà , suivis de deux semaines de « vacances »: date de rendu sur updago : **vendredi 26 avril 20h**.

Vous devez rendre un zip de votre projet, en ayant indiqué **vos deux noms** dans un **commentaire dans main.c**. **Un seul dépôt sur updago svp**.

Evaluation

Vous serez évalué lors d'un oral, nous vérifierons que vous maitrisez votre code et nous prêterons attention à :

- La **lisibilité** de votre code et à vos **commentaires**.
- Votre attention à la **complexité** de vos fonctions.
- A la réutilisation de votre bibliothèque de listes réalisée en TP, avec un mécanisme **d'encapsulation**.

Annexe : Valeurs des unités

Vous pouvez vous créer des fonctions qui renvoient un « type » d'unité en particulier, par exemple un chevalier, qui apparaîtra aux coordonnées posx et posy :

```
Tunite *creeChevalier((int posx, int posy);
```

Ou encore :

```
Tunite *creeTour(int posx, int posy); //les coordonnées seront fonction du camp
Tunite *creeTourRoi(int posx, int posy);
Tunite *creeArcher(int posx, int posy);
Tunite *creeGargouille(int posx, int posy);
Tunite *creeDragon(int posx, int posy);
```

Ces fonctions allouent sur un pointeur une structure initialisée avec les caractéristiques de l'unité souhaitée (posx, posy, points de vie, dégâts, portée, etc.) et retournent ce pointeur.

Tableau des caractéristiques :

nom	tour	tourRoi	chevalier	archer	dragon	gargouille
cibleAttaquable	solEtAir	solEtAir	sol	solEtAir	solEtAir	solEtAir
maposition	sol	sol	sol	sol	air	air
pointsDeVie	500	800	400	80	200	80
vitesseAttaque	1.0	1.2	1.5	0.7	1.1	0.6
degats	100	120	250	120	70	90
portee	3	4	1	3	2	1
vitessedeplacement	0	0	2.0	1.0	2.0	3.0
coutEnElixir	0	0	4	2	3	1

Remarque : les tours et tours du roi ne sont créés qu'au lancement du jeu et n'ont donc pas de coût en élixir.

Annexe : Affichage « démo » à modifier

Quand vous compilerez et exécuterez le squelette du projet, vous verrez apparaître les tours et tours du roi de chaque camp. Attention, ces unités sont « fantômes » et ne sont rattachées à aucune liste d'unité des joueurs. Elles apparaissent car elles ont été rajoutées « arbitrairement » dans la fonction `initPlateauAvecNULL` :

```
void initPlateauAvecNULL(TplateauJeu jeu,int largeur, int hauteur){
    for (int i=0;i<largeur;i++){
        for (int j=0;j<hauteur;j++){
            jeu[i][j] = NULL;
        }
    }

    //POUR LA DEMO D'AFFICHAGE UNIQUEMENT, A SUPPRIMER
    //(les tours ici ne sont pas liées aux listes des unités de vos joueurs)
    jeu[5][3]=creeTour(5,3);
    jeu[5][1]=creeTourRoi(5,1);
    jeu[5][15]=creeTour(5,15);
    jeu[5][17]=creeTourRoi(5,17);
    //FIN DEMO AFFICHAGE
```

}

Cette fonction `initPlateauAvecNULL` est dans le fichier clashloyal.c, vous devez supprimer les lignes indiqués pour faire apparaître les unités de VOS listes.

Annexe : code « main » fourni

Vous devez **créer vos variables** dans la zone prévue et signalée par un encadré sous forme de commentaire C, de même pour **vos appels de fonctions**.

Vos fonctions sont à définir dans clahloyal.h/.c bien sûr.

Capture écran du main qui vous est fourni dans le zip du projet :

```
/*********************************************************************/
/*                                                                   */
/*            DEFINISSEZ/INITIALISER ICI VOS VARIABLES               */
/*                                                                   */
// FIN de vos variables                                                */
/*********************************************************************/

// boucle principale du jeu
int cont = 1;
while ( cont != 0 ){    //VOUS DEVEZ GERER (DETECTER) LA FIN DU JEU -> tourRoiDetruite
        SDL_PumpEvents(); //do events


        /*********************************************************************/
        /*                                                                   */
        /*                                                                   */
        //APPELEZ ICI VOS FONCTIONS QUI FONT EVOLUER LE JEU
        /*                                                                   */
        /*                                                                   */
        // FIN DE VOS APPELS
        /*********************************************************************/
```