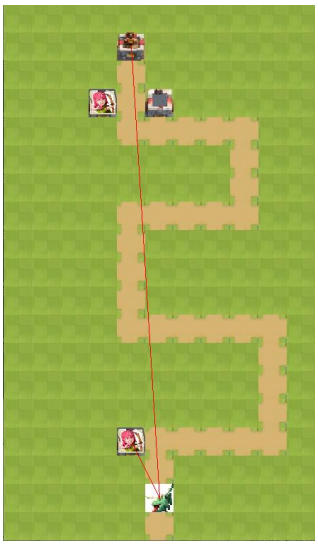
Projet Tower Defend

Présentation



Nous allons réaliser un TowerDefend.

Le jeu consiste à voir avancer des unités assaillantes le long d'un chemin connu, défendu par des tours.

Le but est de défendre la tour du roi, placé à l'extrémité du chemin.

Le jeu s'arrête quand la tour du roi est détruite (0 point de vie).

Notre jeu sera automatique (placement des tours de défense établi par un algorithme de votre choix et apparition d'unités assaillante aléatoires au début du chemin).

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** tel que défini dans ce sujet. **Les types sont imposés.**

Le type Tunite

Une unité pourra être à la fois une tour aérienne (ne tire que sur des unités aériennes), une tour terrestre (ne tire que sur des unités au sol), la tour du roi (tire sur les unités terrestres et aériennes), 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{tourAerienne, tourTerrestre, 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

} Tunite;
```

Deux joueurs : le roi et la horde.

Le premier joueur dit « le roi » ne disposant que de tours terrestres et aériennes.

Le second joueur dit « la horde » sera composé de gargouilles, dragons, archers et chevaliers.

Les joueurs seront vus comme des listes (simplement chainées) 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.

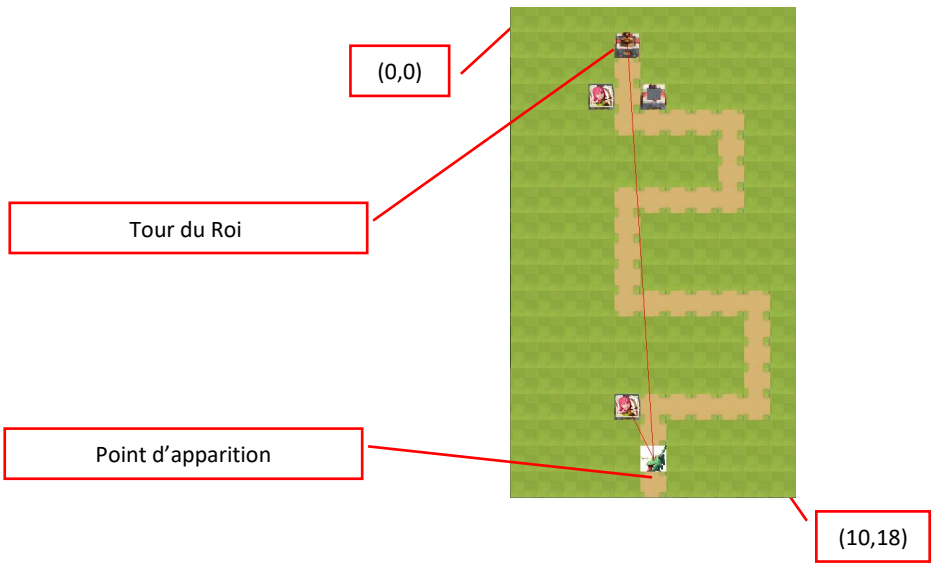
```
typedef Tunite* **TplateauJeu ;
```

Le plateau du jeu aura 11 colonnes et 19 lignes. Une tour du roi sera présente dès le début de partie au bout du chemin. A son début apparaitront des unités aléatoires qui, en suivant ce chemin essayeront de faire tomber cette tour du roi (0 point de vie).

Le long du chemin, des tours de défenses inattaquables et soit anti-aériennes soit terrestres apparaitront tous les n tours (la valeur de n étant à définir pour équilibrer le jeu). L'emplacement d'apparition des tours de défense sera défini par un algorithme de votre choix.

Les tours de défense ont une portée restreinte et ne peuvent pas être placées sur le chemin.

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 « Horde » et « Roi ». 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 en regardant les unités proches dans le plateau du jeu (type **TplateauJeu**) si besoin.

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 = AlloueTab2D(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 de la tour du roi.

Une Fonction utile, à se coder sans doute :

```
Bool tourRoiDetruite(TListePlayer playerRoi);
```

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(TplateauJeu jeu, Tunite *UniteAttaquante) ;
```

La fonction `quiEstAPortee` crée et renvoie la liste des unités qui peuvent être attaquer par `UniteAttaquante`

  - Optionnel : tri de la liste (via un tableau ou tri de la liste avec un pointeur de fonction cf étapes partie2 en réutilisant votre librairie ListeDouble par exemple) sur le nombre de points de vie restant. L'unité la plus faible est attaquée par `UniteAttaquante`
  - Attaques (soustraction des dégâts) par `UniteAttaquante` sur une (seule) des unités de la liste `player` générée par `quiEstAPortee` (perte de points de vie), en vérifiant le champ `peutAttaquer`). Voici l'entête :

```
Void combat(SDL_Surface *surface , Tunite * UniteAttaquante, Tunite * UniteCible) ;
```

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

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

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).

Remarque : les unités de la horde n'attaqueront que la tour du roi, elles ignorent les tours de défenses qui elles leur tirent dessus.

- **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, voire reste sur place.

- **Phase création** d'une nouvelle unité par camp (soit une tour de défense, soit une unité de la horde). On va appliquer une probabilité pour chaque camp qu'une création ait lieu : 15-50% pour les tours, 5-60% pour les unités de la horde. Ces pourcentages et cette règle peuvent être modifiés pour équilibrer le jeu.

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

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

Où apparaissent les tours ?
Vous êtes libres de choisir où vous ferez apparaître les nouvelles tours de défense. Par défaut, elles peuvent « naître » le long du chemin, de part et d'autre.

Où apparaissent les unités de la horde ?
Ces unités apparaissent au début du chemin et le suivront pour atteindre la tour du roi.

- **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);
```

Le chemin

Un tableau tabParcours est prédéfini dans le main. Il contient les cases du chemin. La case numéro 0 (première case) est la case où apparaissent les unités de la horde. La dernière case est occupée par la tour du roi.

Le nombre de case du chemin est défini par une constante **NBCOORDPARCOURS** (dans towerdefend.h).

Regarder le corps de la fonction **afficheCoordonneesParcours** (dans towerdefend.c) pour bien comprendre/voir comment manipuler ce tableau à deux dimensions **tabParcours.**

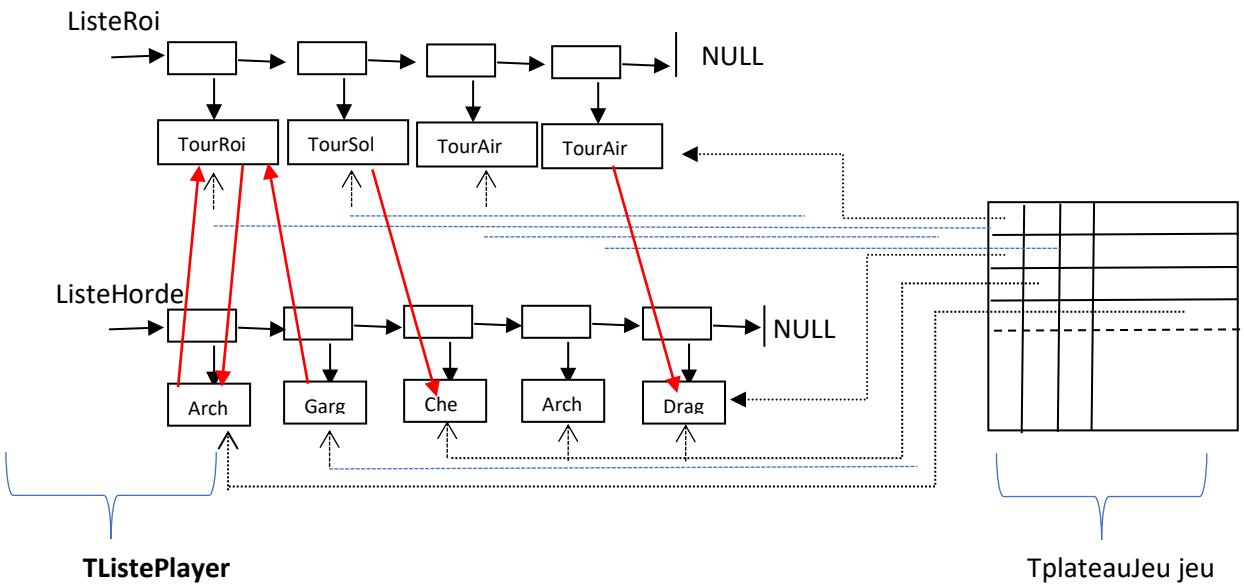
Le chemin est dessiné automatiquement mais il n'est pas codé comme une unité.

Cependant, lire le tableau tabParcours va vous permettre de positionner vos tours de defenses.

Métrique du jeu

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

La structure du jeu en mémoire



Certaines cases du jeu (type Tplateaujeu) pointent vers des unités de la listeHorde ou de la ListeRoi.

Supprimer une unité du jeu (si 0 point de vie) demande donc de retirer son lien présent dans le tableau jeu, puis de la supprimer dans la liste de son joueur.

Pour déterminer qui est à portée de qui, vous regardez dans le tableau jeu les éventuelles cibles d'une unité de chaque liste (horde ou roi).

Les flèches en rouges illustrent cette possibilité de combats entre les unités.

Animation « dessineAttaque »

Une fonction dessineAttaque permet une animation très simpliste.

Cette fonction est utilisée dans le projet de démo (point de départ) sur des unités « bidons ».

Vous pouvez l'utiliser dans votre fonction « combat », avec une contrainte : ajouter un paramètre lié à la SDL (librairie graphique2D utilisé dans ce projet) -> **SDL_Surface *surface.**

Regarder le code de dessineAttaque pour comprendre simplement que vous aurez l'obligation de fournir la variable pWinSurf (de type **SDL_Surface *, déclaré dans le main)** afin de la propager à dessineAttaque (cf ligne 97 du main de la démo).

Schématiquement :

```
Void combat(SDL_Surface *surface , Tunite * UniteAttaquante, Tunite * UniteCible) {
          //gestion du combats, soustraction dégâts, etc.
          // et animation :
          dessineAttaque(surface, UniteAttaquante, UniteCible);
          }
```

Point de départ du projet

Un projet CodeBlock « projetTowerDefend » 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 projetTowerDefend contient déjà les types demandés et certains prototypes de fonctions.

Vous serez amené à créer d'autres fonctions **dans** les fichiers **towerdefend.h** et **towerdefend.c (selon vos besoins)**, que vous utiliserez dans main.c. Au besoin, vous pouvez aussi modifier les prototypes des fonctions proposés, si vous avez une justification liée à votre conception du jeu.

Travail demandé

Vous devez coder le noyau du jeu (hors interface graphique donc) et faire en sorte que deux « joueurs type ordinateur» (« Horde » et « Roi ») 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 » est 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 deux semaines de TP encadrés et de trois semaines au-delà : date de rendu sur updago : **samedi 22 avril 23h59**.

Vous devez rendre un **zip** (pas de .rar ou autres formats +/- exotiques) de votre projet, en ayant indiqué **vos deux noms** dans un **commentaire dans main.c**. **Un seul dépôt sur updago svp**.

Des **oraux** sont organisés pour vous évaluer **entre le lundi 24 avril et le jeudi 27 avril**. Votre code doit être le plus lisible possible et commenté.

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 *creeTourSol(int posx, int posy); //les coordonnées seront fonction du camp
Tunite *creeTourAir(int posx, int posy);
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	tourSol	tourAir	tourRoi	chevalier	archer	dragon	gargouille
cibleAttaquable	sol	air	solEtAir	sol	solEtAir	solEtAir	solEtAir
maposition	sol	sol	sol	sol	sol	air	air
pointsDeVie	500	500	800	400	80	200	80
vitesseAttaque	1.5	1.0	1.2	1.5	0.7	1.1	0.6
degats	120	100	180	250	120	70	90
portee	5	3	4	1	3	2	1
vitessedeplacement	0	0	0	2.0	1.0	2.0	3.0

Annexe : Affichage « démo » à modifier

Quand vous compilerez et exécuterez le squelette du projet, vous verrez apparaître des tours, la tour du roi et un Dragon plus deux animations simplistes. Attention, ces unités sont « fantômes » et ne sont rattachées à aucune liste d'unité de joueur (horde ou roi). 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]=creeTourSol(5,3);
    jeu[3][3]=creeTourAir(3,3);
    jeu[4][1]=creeTourRoi(4,1);
    jeu[4][15]=creeTourAir(4,15);
    jeu[5][17]=creeDragon(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.

De plus, du code de démo (liste des coordonnées du chemin tabParcours et animation dessineAttaque) est présent dans le main, à regarder/comprendre puis à supprimer.

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 towerdefend.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
        /*************************************************************************/
```