Université de Versailles – Saint-Quentin 2019–2020
Licence 1

IN 200 – Polycopié de TD

Sandrine Vial – Sandrine.Vial@uvsq.fr
Version du 3 février 2020.

Table des matières

0 Introduction

Environnemnt de programmation

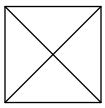
- La machine virtuelle à utiliser pour cette UE est IN100_IN200 qui est disponible sur le site du cartable numérique.
- Le présent poly est disponible dans la machine virtuelle et également sur l'espace `moodle` de l'IN200.
- L'environnement de programmation basé sur sdl est dans la machine virtuelle et est disponible également sur e-campus.

1 TD 01 – Rappels et premiers rebonds

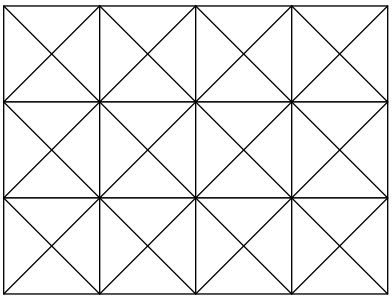
Le but de cette partie est de se familiariser avec l'environnement de programmation en faisant quelques fonctions élémentaires.

1.1 Rappels

Exercice 1 *Des Croix*
On souhaite effectuer le dessin suivant :



1. Écrire une fonction qui, étant donnés un point bg et une longueur lg, affiche la croix dessinée ci-dessus, avec : le point en bas à gauche du carré est bg et la longueur du carré est lg.
2. Écrire une fonction qui, étant donnés un point bg, une longueur lg, un nombre de lignes nbl et un nombre de colonnes nbc, affiche nbl lignes contenant le dessin en croix nbc fois.
   Par exemple, si $nbl = 3$ et $nbc = 4$, la fonction affiche :

   Le point bg est le point situé le plus en bas à gauche dans le dessin et lg est la longueur d'un côté d'un petit carré. Vous utiliserez la fonction écrite dans la question 1.

Exercice 2 *Triangle*

1. Écrire une fonction : `void my_draw_triangle(POINT P1, POINT P2, POINT P3, COULEUR C);`
   qui prend en argument trois points et qui affiche un triangle non rempli de la couleur passée en argument.
2. Écrire une fonction : `void my_draw_triangle_from_clic(COULEUR C);`
   qui attend 3 clics et qui affiche un triangle non rempli de la couleur passée en argument.
   Points importants à bien comprendre :

— Les deux fonctions ne font pas exactement la même chose, la première fonction prend des points en argument, il est donc inutile de faire des `wait_clic()` dans la fonction.
— La deuxième peut être écrite en utilisant la première

3. Dans la fonction `main` faites une boucle qui appelle dix fois la première fonction, les points passés en arguments étant issus de `wait_clic()`.
4. Dans la fonction `main` rajouter une boucle qui appelle dix fois la deuxième fonction.

1.2 Premiers rebonds

Exercice 3 *J'affiche, j'affiche pas*
On considère le programme suivant (déjà en place dans la machine virtuelle ou à récupérer sur e-campus (`exo3.c`) :

```
#include "uvsqgraphics.h"

int main()
{
POINT p;

init_graphics(900,900);

// Par défaut on est en "affiche_auto_on"
p = wait_clic();
draw_fill_circle(p,200,bleu);
p = wait_clic();
draw_fill_circle(p,200,rouge);


// On passe en off
affiche_auto_off();
p = wait_clic();
draw_fill_circle(p,200,vert);
p = wait_clic();
draw_fill_circle(p,200,jaune);
// Il faut faire affiche_all pour que ca s'affiche
affiche_all();
// On est toujours en off
p = wait_clic();
draw_fill_circle(p,200,gris);
p = wait_clic();
draw_fill_circle(p,200,orange);
// Il faut faire affiche_all pour que ca s'affiche
affiche_all();

// On passe en on
// Plus besoin d'affiche_all
affiche_auto_on();
p = wait_clic();
draw_fill_circle(p,200,cyan);
p = wait_clic();
```

```
draw_fill_circle(p,200,magenta);


wait_escape();
exit(0);
}
```

1. Comprenez le lien entre le code et ce qui se passe lors de l'exécution.

Exercice 4 *J'efface, j'affiche et ça bouge*

1. Écrire une fonction :
   `void efface_affiche(POINT ancien, POINT nouveau);`
   qui affiche un cercle noir de rayon 25 centré sur le point `ancien` et qui affiche un cercle rouge de rayon 25 centré sur le point `nouveau`.
2. Dans la fonction `main` simuler le déplacement d'un cercle en faisant des appels successifs à la fonction `efface_affiche`.
   — Le cercle doit se déplacer de 2 en x et de 3 en y à chaque itération.
   — Au départ le cercle est centré en $(50, 50)$.
   — Utilisez les fonctions `void affiche_auto_off();` et `affiche_all();` (voir la section A.2 pour avoir un affichage plus fluide).
3. Écrire une fonction : `POINT deplace_balle(POINT ancien, int dx, int dy);`
   qui prend en argument un point et renvoie le point déplacé de `dx` en `x` et de `dy` en `y`.
4. Écrire une fonction `void affiche_balle(POINT P);`
   qui prend en argument un point et affiche un cercle rouge de rayon 25 centré sur ce point.
5. Écrire une fonction `void efface_balle(POINT P);`
   qui prend en argument un point et affiche un cercle noir de rayon 25 centré sur ce point.
6. Réécrire la fonction `main` avec ces nouvelles fonctions.
7. Écrire une fonction : `POINT init_balle(int x, int y);`
   qui prend en argument des coordonnées et renvoie un point ayant ces coordonnées.
8. Réécrire la fonction `main` avec cette nouvelle fonction.

Exercice 5 *Sortir c'est rebondir*
Écrire une fonction : `int trop_haut (POINT P, int r);`
qui prend en argument un point et un rayon et qui renvoie 1 si le cercle centré en `P` et de rayon `r` dépasse en haut de l'écran et 0 sinon.

Réécrire la fonction `main` avec cette nouvelle fonction en testant à chaque affichage si la balle dépasse. Si elle dépasse, modifier le déplacement en y qui devra alors être de -2.

Le but de ce TD est d'afficher des dégradés de gris puis de couleur. On rappelle que la fonction :
`COULEUR couleur_RGB(int r, int g, int b)`
prend en argument trois entiers dans l'intervalle $[0..256[$ et renvoie la couleur correspondante.

Exercice 6 *Déclaration*
Déclarer en variable globale un tableau `T1` à une dimension de taille 256 et contenant des cases de type `COULEUR`.

Exercice 7 *Remplissage du tableau*
Écrire une fonction :
`void remplir_gris()`
qui remplit la $i^{\text{ème}}$ case de `T1` avec la couleur qui a comme valeur $(r, g, b) = (i, i, i)$.

Exercice 8 *Affichage du tableau*
Écrire une fonction :
`void afficher_horizontal()`
qui pour chaque case i du tableau affiche un rectangle de largeur 2, de hauteur la hauteur de la fenêtre et de couleur `T[i]`. Le coin en bas à gauche du rectangle de la case i est aux coordonnées $(2 \times i, 0)$.

Exercice 9 *Enchaînement 1*
Écrire le programme principal qui initialise la fenêtre graphique de taille 512×512 puis qui appelle la fonction `remplir` puis la fonction `afficher`. N'oubliez pas `wait_escape()`.

Exercice 10 *Enchaînement 2*
Écrire une fonction :
`void afficher_vertical()`
qui affiche le tableau à la verticale.

Exercice 11 *En rouge*
Écrire une fonction :
`void remplir_rouge()`
qui remplit la $i^{\text{ème}}$ case de `T1` avec la couleur qui a comme valeur $(r, g, b) = (i, 0, 0)$.
Afficher ce tableau.

Exercice 12 *En 2D*
Définir un tableau `T2` à deux dimensions chacune de taille 512. Écrire une fonction :
`void remplir_rouge_bleu()`
qui remplit la case (i, j) avec la couleur $(r, g, b) = (i, 0, j)$.
Afficher le tableau en deux dimensions.

Exercice 13 *En 2D encore*
Définir un tableau `T3` à deux dimensions chacune de taille 512. Écrire une fonction :
`void remplir_aléatoire()`
qui remplit la case (i, j) avec une couleur dont les composantes r,v et$b$ sont tirées aléatoirement.
Afficher le tableau en deux dimensions.

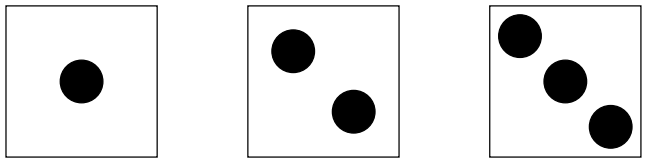
2 TD 03 – Jeu de NIM

Le jeu de Nim se joue avec N allumettes et deux joueurs. Au départ les N allumettes sont posées sur une table. Chaque joueur retire alternativement 1, 2 ou 3 allumettes. Celui qui retire la dernière allumette a perdu.

On suppose qu'il y a 23 allumettes au départ.

1. Quelle données doit-on stocker à chaque instant du jeu pour pouvoir gérer ce jeu ?
2. Ecrire une fonction :
   `void affiche_une_allumette (POINT P)`
   qui dessine une allumette dont la base est au point `P`.

3. Ecrire une fonction :
   `void efface_une_allumette(POINT P)`
   qui efface l'allumette dont la base est au point `P`.
4. Ecrire une fonction :
   `void affiche_allumettes (int N)`
   qui affiche `N` allumettes aux coordonnées (20,200) (40,200) (60,200) ...
5. Ecrire une fonction : `void affiche_123 ()`
   qui affiche en bas de la fenêtre les 3 carrés comme dans la figure ci-dessous :



6. Ecrire une fonction : `int get_valeur()`
   qui attend que l'on clique dans un des 3 carrés et qui renvoie 1, 2 ou 3 en fonction du carré cliqué.
7. Ecrire une fonction `jouer_humain` qui affiche le nombre d'allumettes restantes, qui attend que le joueur clique sur un des 3 carrés et qui met à jour les données et l'affichage.
8. Ecrire une fonction `jouer` qui demande à deux joueur humains de jouer en alternance et qui affiche le gagnant.

3 TD 04 et TD 05 – Balles et raquettes

Le but de ce TD est de comprendre la notion de `struct` *et de bien comprendre les passages d'arguments et retour de fonction.*

Exercice 14 *Avec un struct c'est plus propre*
Déclarer le type :

```
struct balle
       {
       POINT centre;
       int rayon;
       COULEUR coul;
       int dx, dy;
       };
```

Déclarer le synonyme de type :

```
typedef struct balle BALLE;
```

Écrire les fonctions :
— `BALLE init_balle ();` qui initialise une balle avec des valeurs qui vous semblent pertinentes.
— `BALLE deplace_balle (BALLE B);` qui déplace la balle en ajoutant aux champs `centre.x` et `centre.y` les valeurs des champs `dx` et `dy`.
— `void efface_balle (BALLE B);` qui efface la balle.
— `void affiche_balle (BALLE B);` qui affiche la balle.
— `BALLE rebond_balle (BALLE B);` qui modifie les champs `dx` et `dy` si la balle dépasse d'un bord.

3.1 Tableau de balles

Exercice 15 *Tableau de balles*

- Initialiser une fenêtre de 400×400.
- Déclarer un tableau de balles (la définition du type `BALLE` est la même que pour l'exercice précédent).
- L'initialisation de valeurs pour chaque balle est aléatoire :
  - position : aléatoire entre 100 et 300 en `x` et en `y`.
  - couleur : aléatoire.
  - rayon : aléatoire entre 5 et 30.
  - dx et dy : aléatoire entre -5 et $+5$ mais non nuls.
- Écrire un programme principal qui fait se déplacer $N = 100$ balles.
- Ajouter un mur vertical sans épaisseur, sur toute la hauteur, centré de tel sorte que les balles rebondissent sur ce mur.

3.2 Pong

Exercice 16 *Déclaration de type*
On veut pouvoir stocker une raquette rectangulaire comme dans un casse-brique élémentaire. La raquette ne se déplace pas toute seule, elle n'a donc pas besoin de stocker d'informations de déplacement comme le type `BALLE`.

Déclarer un type `RAQUETTE` en commentant l'usage de chacun des champs.

Exercice 17 *Effacer afficher la raquette*
Écrire deux fonctions, une qui affiche et une qui efface la raquette.

Exercice 18 *La fonction `get_arrow()` de la bibliothèque `graphics.h`*
Lorsque la fonction `get_arrow()` est appelée, cette fonction renvoie un `POINT`, donc un appel correct est :

```
POINT P;
...
P = get_arrow();
```

- Si l'utilisateur a appuyé une fois sur la flèche gauche, le champ `x` du point `P` vaudra -1.
- Si l'utilisateur a appuyé n fois sur la flèche gauche, le champ `x` du point `P` vaudra $-n$.
- Si l'utilisateur a appuyé une fois sur la flèche droite, le champ `x` du point `P` vaudra $+1$.
- Si l'utilisateur a appuyé n fois sur la flèche droite, le champ `x` du point `P` vaudra $+n$.
- Si l'utilisateur a appuyé ni sur la flèche droite ni sur la flèche gauche, le champ `x` du point `P` contiendra 0.

La fonction n'est pas bloquante, c'est à dire qu'elle n'attend pas que l'utilisateur appuie sur une flèche, elle renvoie juste quelle flèche a été appuyée ou pas.

Écrire une fonction `RAQUETTE deplace_raquette(RAQUETTE R, int dx)` qui déplace la raquette en fonction du `dx` passé en argument.

Écrire un programme principal qui fait se déplacer la raquette, chaque appui sur une flêche faisant se déplacer la raquette de -1 ou $+1$.

Exercice 19 *Jolie raquette*
Modifier le déplacement de la raquette pour que :

1. elle soit plus rapide.
2. elle ne puisse pas sortir de l'écran.

Exercice 20 *Balle et Raquette*
Reprenez le code des exercices précédents pour qu'une balle et la raquette s'affichent en même temps. La balle doit rebondir et la raquette doit se déplacer avec les flèches.

Exercice 21 *Choc balle raquette*
Écrire une fonction `choc` qui prend en argument une balle et une raquette et qui renvoie 1 si le cercle de la balle et le rectangle de la raquette ont une intersection et qui renvoie 0 sinon.

En reprenant le code qui a été écrit dans les exercices précédents, faites rebondir la balle sur la raquette.

Exercice 22 *Le jeu de base*
Programmer le jeu pour qu'il respecte les règles suivantes :
- Au départ la balle va vers le haut.
- Au départ le joueur dispose de 3 vies.
- La balle ne rebondit pas en bas mais disparait et dans ce cas, le joueur perd une vie.
- Chaque rebond de la balle sur la raquette fait marquer un point.
- Quand le joueur n'a plus de vie, le jeu se termine.

En utilisant la fonction :
`void aff_int(int n, int taille, POINT p, COULEUR C);`
qui affiche l'entier `n` de la taille de police `taille`, positionné en `p` et de couleur `C`
afficher le nombres de vies restantes et les points du joueur de manière jolie.

Exercice 23 *Options de jeu*
Implémenter ces differentes options

1. Au bout de 10 rebonds avec la même balle, une vie de plus.
2. Au bout de 20 rebonds avec la même balle, deux nouvelles balles apparaissent pour jouer simultanément avec 3 balles
3. À chaque rebond la balle accélère
4. À chaque rebond sur la raquette, la raquette diminue de taille (sans jamais être nulle)
5. À chaque rebond sur la raquette, le plafond descend.
6. À chaque rebond la balle a son rayon qui diminue
7. Inventez une option et programmez là.

3.3 Casse brique

Exercice 24 *Les briques*

Le but est d'ajouter des briques en haut de l'écran et de pouvoir détruire ces briques avec la balle. On se propose de diviser la largeur de l'écran en 10 briques et de mettre 5 rangées de briques en hauteur. Chaque brique a sa propre couleur.

A chaque rebond de la balle sur la raquette, la raquette et la balle échangent leur couleur.

A chaque rebond de la balle sur une brique, si la brique et la balle sont de la même couleur, la brique disparait.

A chaque rebond de la la balle sur une brique, si la brique et la balle ne sont pas de la même couleur, la balle change de couleur aléatoirement.

1. Proposez une structure de données pour stocker les briques.
2. Ecrire une fonction qui affiche les briques, en effaçant celles qui ont été détruites.
3. Ecrire une fonction `rebond` dont vous préciserez les arguments et qui gère le rebond de la balle sur les briques.
4. Quand il n'y a plus de briques, le jeu se termine et le joueur a gagné.

4 TD 06 et 07 – Horloge Analogique et Digitale

Vous allez programmer une horloge analogique et une horloge digitale. Dans une fenêtre de taille 900×600, vous partagerez votre écran en deux parties : le tiers du bas servira à afficher l'horloge digitale, les deux tiers du haut, l'horloge analogique.

Votre programme consistera en une boucle infinie qui réaffichera les 2 horloges à intervalles réguliers.

4.1 Horloge analogique

1. Ecrire une fonction `void afficher_cadran()` qui affiche le cadran de votre horloge analogique. Dans cette fonction vous ferez apparaitre les 12 repères de de votre horloge avec leurs chiffres. Il existe les fonctions `sinf()` (sinus) et `cosf()` (cosinus) qui prenne en paramètre un angle en radians et renvoie la valeur du sinus ou du cosinus de cet angle. Tester votre fonction.
2. Ecrire une fonction `void afficher_aiguille_heure(int h, int m, int s)` qui affiche la petite aiguille des heures en fonction de `h, m` et `s` sur le cadran. Tester votre fonction.
3. Ecrire une fonction `void afficher_aiguille_minute(int h, int m, int s)` qui affiche la grande aiguille des minutes en fonction de la valeur de `h, m` et `s` sur le cadran. Tester votre fonction.
4. Ecrire une fonction `void afficher_aiguille_seconde(int h, int m, int s)` qui affiche l'aiguille des secondes en fonction de la valeur de `h, m` et `s` sur le cadran. Tester votre fonction.
5. Ecrire une fonction `void afficher_aiguilles(int h,int m,int s)` qui affichent les 3 aiguilles sur votre cadran. Tester votre fonction.
6. Le main de votre programme doit ressembler à cela.

```
int main()
{
   int h,m;


  while(1)
  {
     h = heure(); m = minute();s = seconde();
     afficher_cadran();
     afficher_aiguilles(h,m,s);
   }
   return 1;
}
```

4.2 Horloge Digitale.

Dans cette partie, vous devez afficher la même heure que pour l'horloge analogique mais sous forme digitale c'est à dire : hh :mm :ss

Chaque chiffre de cette horloge sera représenté comme des segments allumés parmi les 7 du diagramme suivant :

```
 _
|_|
|_|
```

1. Ecrire une fonction `void transforme(int a, int segments[7])` qui pour un entier `a` entre 0 et 9, remplit le tableau `segments` avec des `0` pour les segments éteints et des `1` pour les segments allumés.
2. Ecrire les fonctions :
   (a) `void afficher_dizaine_heure(int segments[7])`
   (b) `void afficher_unite_heure(int segments[7])`
   (c) `void afficher_dizaine_minute(int segments[7])`
   (d) `void afficher_unite_minute(int segments[7])`
   (e) `void afficher_dizaine_seconde(int segments[7])`
   (f) `void afficher_unite_seconde(int segments[7])`
   et les tester une par une.
3. Ecrire une fonction `void deux_points()` qui affiche et éteint des " :" clignotants entre les heures et les minutes et entre les minutes et les secondes.
4. Ecrire une fonction `void affiche_digitale(int h, int m, int s)` qui affiche l'heure sous forme analogique sur votre écran.
5. Modifier votre `main` pour qu'il appelle cette fonction et affiche vos deux horloges simultanément.

4.3 Chronomètre

Nous allons rajouter une fonctionnalité à cette horloge : un chronomètre. Pour cela, vous allez rajouter un bouton dans la partie digitale qui va permettre de passer de la fonctionnalité "horloge" à la fonctionnalité chronomètre. Pour bien utiliser cette fonctionnalité nous allons devoir utiliser la structure de données suivante :

```
struct chrono {
     int h;
     int m;
     int s;
};

typedef struct chrono CHRONO;
```

Vous écrirez pour cela les fonctions suivantes :

1. `void afficher_bouton()` qui permet de passer d'un mode "horloge" à un mode "chronomètre".
2. `CHRONO init_chronomètre()` qui initialise un chronomètre à 0 ;
3. `void afficher_chrono(CHRONO c)` qui affiche sous forme digitale la valeur d'un chronomètre.
4. Lorsque vous êtes en mode "chronomètre", vous allez afficher 2 boutons supplémentaires un bouton `START/STOP` qui permet de démarrer et d'arrêter un chronomètre et un bouton `TOUR` qui permet d'afficher pendant 5 secondes le temps d'un tour.

5 TD 08 et 09 – Jeu du Boulier

Il s'agit de programmer un jeu graphique. Le jeu est composé d'un plateau rempli de taille $x \times y$ cases de boules multicolores. L'objectif est de supprimer toutes les boules du plateau en maximisant votre nombre de points. Pour cela, on définit des ensembles de boules comme étant un ensemble de boules de même couleur

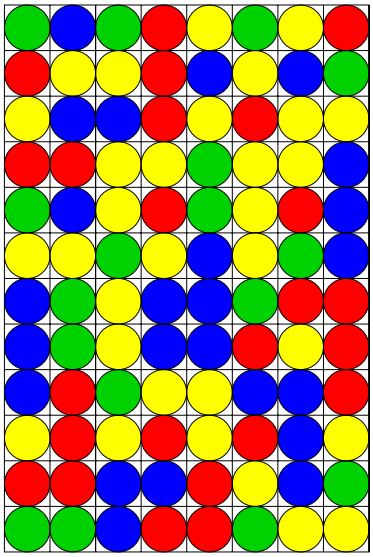


FIGURE 1 – Initialisation du plateau de jeu 8×12 avec 4 couleurs.

et adjacentes (par l'un des quatre côtés de la case sur laquelle est placée la boule). Un ensemble contient au moins deux boules.

Les règles sont les suivantes :

1. Un clic sur une boule supprime l'ensemble de boules auquel elle appartient.
2. Quand un ensemble de boules est supprimé, les boules qui se trouvent au dessus tombent dans les espaces vides en dessous. Si une colonne entière est libérée, les boules sont poussées vers la droite pour combler l'espace créé.
3. Chaque fois que vous enlevez un groupe de boules vous gagnez des points. Chaque groupe de n boules supprimé vous rapporte $n \times (n-1)$ points.
4. Si vous supprimez toutes les boules du plateau, vous obtenez un bonus de 200 points.

5.1 Ce que vous devez programmer

Voici une liste des fonctions à écrire, vous pouvez rajouter toutes les fonctions qui vous paraissent utiles et qui aident à la clarté et à la simplicité de votre code. Vous prendrez soin de tester chacune d'elles. Vous écrirez au départ une fonction `main()` minimale que vous enrichirez au fur et à mesure.

1. On se définit les constantes suivantes :

   ```
   #define LARGEUR 8
   #define HAUTEUR 12
   ```

   Définir en variable globale le tableau à deux dimensions `B` qui contient les boules (on ne stockera que la couleur des boules). Définir aussi une variable globale `score` qui contient le score.
2. Ecrire une fonction `COULEUR couleur_aleatoire()` qui tire au hasard une couleur parmi jaune, rouge, bleu ou vert.
3. Ecrire une fonction `void init_boules()` qui initialise le tableau `B` avec des boules colorées (les couleurs sont tirées aléatoirement).
4. Ecrire une fonction `void affiche_quadrillage()` qui affiche le quadrillage dans la fenêtre.

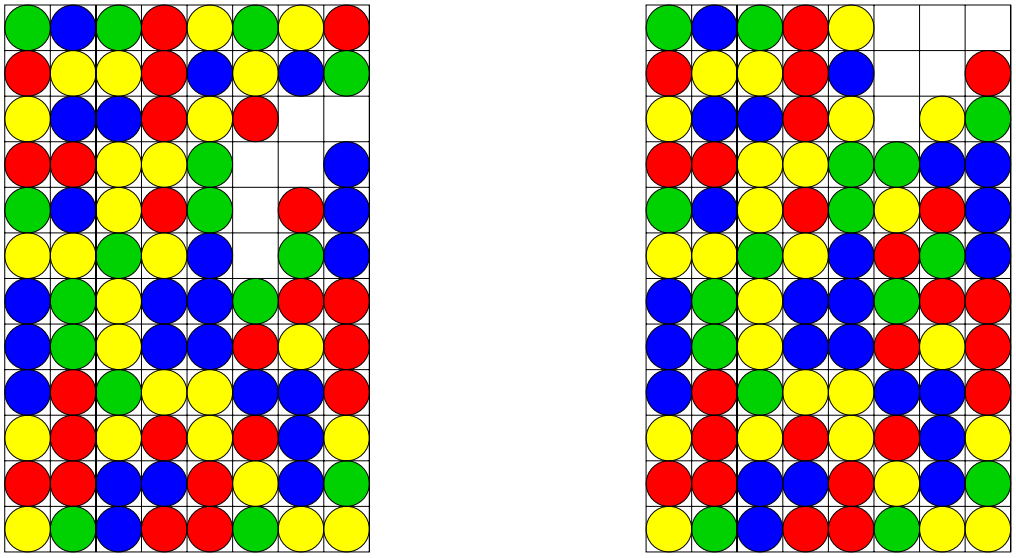


FIGURE 2 – Suppression d'un groupe et déplacement des boules restantes.

5. Ecrire une fonction `void affiche_boules()` qui affiche les boules dans la fenêtre.
6. Ecrire une fonction `void affiche_score()` qui affiche le score en haut de la fenêtre.
7. Ecrire une fonction `void affiche()` qui permet de faire l'affichage complet de la fenêtre (le score, le quadrillage et les boules).
8. Ecrire une fonction `int mettre_a_noir(COULEUR coul, int i, int j)` qui met la case de couleur `coul` de coordonnées `i` et `j` en noir. Cette fonction est appelée récursivement sur les cases voisines de la case de coordonnées (`i`, `j`) pour mettre en noir les cases voisines de couleur `coul`.
9. Ecrire une fonction `void gestion_clic()` qui attend un clic, supprime (met à noir et fait les déplacements nécessaires) les boules de l'ensemble sur lequel on a cliqué et met à jour le score.
10. Ecrire une fonction `void descendre_colonne(int j)` qui déplace les boules dans la colonne `j` de telle sorte à ce qu'elles soient tassées vers le bas.
11. Ecrire une fonction `void deplacement_droite()` qui déplace les colonnes qui contiennent encore des boules vers la droite de telle sorte que les colonnes vides soient le plus à gauche possible.

6 TD 10 et 11 – Le Serpent

Le jeu SERPENT se compose d'un serpent qui se déplace sur un quadrillage. Le serpent peut donc aller à droite, à gauche, en bas ou en haut. Sur le quadrillage on retrouve des éléments de couleurs et de formes différentes. Lorsque le serpent passe sur un élément, il rajoute un anneau à son corps avec cet élément ou il meurt.

6.1 Initialisation

Le plateau de jeu est composé d'une grille de 40×60 cases. Chaque case est de taille de 10×10. Une case du quadrillage peut contenir un élément ou être vide. Les éléments peuvent avoir les formes suivantes : un cercle, un carré ou une croix et les couleurs suivantes : Rouge, Jaune, Vert ou Bleu. Le serpent est représenté par une tête (cercle blanc) et une suite d'éléments (on stockera la longueur de cette suite).

Le fichier `Serpent.c` suivant (et disponible aussi sur e-campus) vous donne la structure minimale de votre programme.

1. Compiler et exécuter le fichier `Serpent.c`. Vous devez obtenir une fenêtre de 400×600 complétement noire.
2. Ecrire le corps de la fonction `void init_jeu(struct elem J[40][60])` qui initialise le plateau de jeu en mettant des éléments au hasard sur le plateau.
3. Ecrire le corps de la fonction `void affiche_jeu(J)` qui affiche le plateau de jeu.
4. Compiler et exécuter votre fichier pour tester les deux fonctions précédentes.
5. Ecrire le corps de la fonction `struct serpent init_serpent()` qui initialise le serpent. Son corps est vide. Il n'a qu'une tête, positionnée aléatoirement sur le plateau de jeu.
6. Ecrire le corps de la fonction `void affiche_serpent(struct serpent S)` qui affiche le serpent.
7. Compiler et exécuter votre fichier pour tester les deux fonctions précédentes.

6.2 Déplacements

Maintenant, il faut gérer les déplacements du serpent. Le temps est discrétisé. A chaque top, le serpent avance d'une case dans sa direction courante. Quand le joueur appuie sur une des quatre flèches, la tête du serpent va dans la direction de la flèche appuyée. Les éléments du corps passent exactement par où est passée la tête. Pour cela vous :

1. modifierez la structure `serpent` pour qu'elle contienne la direction courante du serpent ;
2. écrirez une fonction `struct serpent deplace_serpent (struct serpent S, POINT p);` qui en fonction de l'appui sur les touches de flèches stocké dans `p`, déplace le serpent et réaffiche le plateau de jeu et le serpent.

La fonction `main` devient alors :

```
int main()
{

  struct elem J[40][60];
  struct serpent S;
  POINT p;

  init_graphics(400,600);
  init_jeu(J);
  S = init_serpent();
  while (1)
  {
     p = get_arrow();
     S = deplace_serpent(S,p);
     affiche_jeu(J);
     affiche_serpent(S);
  }
  wait_escape();
  exit(0);
}
```

6.3 Manger les éléments

Voici quelques règles à programmer et à tester (dans cet ordre) :

— Quand le serpent heurte les bords de la fenêtre la partie se termine.
— Quand la tête du serpent passe sur un élément, cet élément est rajouté au début du corps.
— Quand la tête du serpent passe sur un cercle rouge, un carré bleu ou une croix jaune, le serpent meurt et le jeu s'arrête.
— Quand trois éléments de la même forme et de la même couleur se suivent dans le corps du serpent, ils disparaissent et la taille du serpent diminue donc de trois.
— A chaque fois que le serpent mange un morceau de corps le score augmente d'un point, quand trois morceaux identiques sont mangés successivement (et donc qui disparaissent) le score augmente de 100 points. Il faut afficher le score sur le plateau de jeu.
— Toutes les 10 secondes, un nouvel élément apparaît aléatoirement sur le plateau de jeu.

7 Bonus – Simulation de propagation d'incendie

Le but de cette partie est de simuler la propagation d'un incendie. Du point de vu de la programmation le but est de renforcer l'habitude du découpage en fonction et de parfaire la connaissance de la manipulation des tableaux.

7.1 Stockage des données

On considère un terrain représenté par $L \times C$ parcelles.

Exercice 25 *Constantes*
Rappeler comment on déclare des constantes et déclarer L et C comme des constantes. On prendra $L = 40$ et $C = 60$.

Rappeler la différence entre constantes et variables.

Le terrain est divisé en $L \times C$ parcelles, chaque parcelle pouvant être de différents types : Terre, Arbre, Eau, Feu, Cendres tièdes et Cendres éteintes.

Au niveau de la représentation graphique, chaque parcelle sera représentée par une couleur correspondant au type de la parcelle :

Type	**Couleur**	**Durée de l'état**
Terre	Marron	$+\infty$
Arbre	Vert	Dépend des voisins
Eau	Bleu	$+\infty$
Feu	Rouge	constante `DUREE_FEU`
Cendres tièdes	Gris	constante `DUREE_CENDRE`
Cendres éteintes	Noir	$+\infty$

`DUREE_FEU` et `DUREE_CENDRE` sont deux constantes à définir dont les valeurs sont laissées à votre appréciation.

Exercice 26 *Structure de données*
Proposer une structure de données (utiliser un `struct`) permettant de stocker une parcelle. Appeler ce type `PARCELLE`.

Déclarer une variable globale `T1` qui permet de stocker les $L \times C$ parcelles de terrain.

7.2 Initialisation des données

Il faut pouvoir remplir les parcelles et comme il y en a beaucoup, on va les remplir au hasard. L'objectif des deux exercices suivants est de créer deux fonctions, une qui remplit une parcelle au hasard, l'autre qui

remplit le tableau. Pour cela, vous avez à votre disposition les fonctions suivantes :

— `float alea_float()`
qui renvoie un nombre flottant (réel) aléatoire dans l'intervalle $[0; 1[$.

— `int alea_int(int N)`
qui renvoie un nombre entier aléatoire dans l'intervalle $[1..N[$.

Exercice 27 *Couleur aléatoire*
Écrire une fonction :
`COULEUR couleur_terrain_aleatoire()`
qui renvoie une couleur au hasard dans l'ensemble des couleurs possibles du terrain, chaque couleur ayant la même probabilité d'apparaitre.

Les couleurs rouge, bleu, vert, noir, gris et marron sont définies et existent.

Exercice 28 *Remplir `T1`*
Écrire une fonction :
`void terrain_aleatoire()`
qui remplit aléatoirement toutes les parcelles du terrain.

7.3 Affichage des données

Maintenant que le terrain est rempli, on veut pouvoir l'afficher. Pour cela vous allez créer deux fonctions, une pour afficher une parcelle, l'autre pour afficher tout le terrain.

Exercice 29 *Affichage*
Écrire une fonction :
`void affiche_parcelle (POINT bg, COULEUR c)`
qui affiche un carré rempli de couleur `c`, de coté 10 et dont le point en bas à gauche est `bg`.

Écrire une fonction :
`void affiche_terrain ()`
qui affiche toutes les cases du tableau `T1`.

N'oubliez pas d'appeler les fonctions : `void affiche_auto_off()` et `void affiche_all()` pour rendre l'affichage plus joli.

Donnez le programme principal qui permet d'initialiser le tableau et de l'afficher.

7.4 Modification des données

Le but étant de simuler la propagation d'un incendie, on va modifier les données. Le principe consiste à avoir deux tableaux `T1` et `T2`. On initialise `T1` puis on l'affiche. On calcule `T2` à partir de `T1` on recopie `T2` dans `T1` puis on affiche `T1` et on recommence : calcul de `T2` à partir de `T1`, etc.

Il faut donc donner deux choses :

— Les règles de calcul de `T2` à partir de `T1`

— Jusqu'à quand continue-t-on cette répétition de calcul et d'affichage ?

Le calcul de `T2` se fait case par case. Pour chaque case de `T1` de coordonnées (i, j) $i^{\text{ème}}$ligne et $j^{\text{ème}}$colonne on regarde les neuf cases centrées autour de la case (i, j). Plusieurs cas sont possibles :

1. Si la parcelle (i, j) est en feu dans `T1` elle reste en feu ou devient en cendres tièdes dans `T2` si elle est restée en feu une durée `DUREE_FEU`.

2. Si la parcelle (i, j) est en cendres tièdes dans `T1` reste en cendres tièdes ou elle devient en cendres éteintes dans `T2` si elle est restée en cendres tièdes une durée `DUREE_CENDRE`.
3. Si la parcelle (i, j) est de type Arbre et qu'une des 8 parcelles autour est en feu la parcelle devient en feu dans `T2`.
4. Une parcelle Eau reste toujours une parcelle Eau et une parcelle Terrain reste toujours une parcelle Terrain.

Pour les cases au ¡¡bord¿¿, il n'y a pas neuf cases voivines, adapter donc l'algo ci-dessous pour les bords.

Exercice 30 *Modification*
Écrire une fonction :
`COULEUR modif_parcelle(int i, int j)`
qui prend en argument les coordonnées d'une case de `T1` et qui calcule la valeur de la case de `T2` de même coordonnées en respectant les rêgles ci-dessus

Écrire une fonction :
`void modif_terrain()`
qui calcule les valeurs des cases de `T2` en fonction des valeurs de `T1`.

Écrire une fonction :
`void recopie_terrain()`
qui recopie `T2` dans `T1`

Écrire le programme principal qui itère sur les modifications de `T1` et `T2`

Une condition d'arrêt évidente est que `T1` n'est plus modifié entre deux itérations.

Exercice 31 *Condition d'arrêt*
Donner l'algorithme qui implémente la condition d'arrêt décrite ci-dessus

Écrire une fonction :
`int continuer()`
qui renvoie 1 si on doit continuer à itérer et 0 sinon.

Optimiser la fonction.

7.5 Amélioration 1 : Déclenchement d'incendie

Afin de rendre la simulation plus réaliste, on va modifier le programme afin qu'au départ il n'y ait pas de parcelle en feu, cendres tièdes ou cendres éteintes.

Exercice 32 *Initialisation terrain : modif. 1*
Modifier la fonction :
`void terrain_aleatoire()` afin de n'avoir que des parcelles Terre, Arbre ou Eau.

Agin de pouvoir déclencher les incendies où on veut, l'utilisateur doit pouvoir après l'initialisation et avant de déclencher les itérations cliquer où il veut sur le terrain et quand il clique, cela met la parcelle cliquée au type Feu. On rappelle qu'il y a la fonction :
`POINT wait_clic()`
qui attend que l'utilisateur clique et renvoie le point où l'utilisateur a cliqué.

Exercice 33 *Initialisation Feu*
Écrire une fonction :
`void init_feu()`
qui attend que l'utilisateur clique. Si le clic est dans le terrain alors la parcelle correspondante est mise à Feu et la fonction attend un nouveau clic. Si le clic est en dehors du terrain, la fonction se termine.

7.6 Amélioration 2 : Arrivée des pompiers

Modifier votre programme pour qu'un clic sur une case du terrain la transforme en Eau pendant une durée de `DUREE_EAU`, après elle redevient un terrain normal. La valeur de la constante `DUREE_EAU` est laissée à votre appréciation.

7.7 Amélioration 3 : Ajouter du vent

Modifier votre programme pour que deux clics successifs sur le terrain donnent la direction et la force du vent. Soitent P_1 et P_2 les deux points cliqués dans cet ordre :

— La direction est donnée par le vecteur $\overrightarrow{P_1P_2}$

— La vitesse du vent est donnée par la distance entre P_1 et P_2.

7.8 Amélioration 4 : Définir la carte avec la souris

Au départ donner à l'utiluisateur de définir les différents types de terrain par des rectangles Afficher en bas de l'écran des carrés correspondant aux différents types de terrains et un carré indiquant : blanc.

— L'utilisateur clique sur un des carrés en bas pour choisir le terrain qu'il veut mettre

— L'utilisateur clique 2 fois sur le terrain, ce qui définit un rectangle qui sera du terrain de la zone précedemment sélectionnée.

— Le clic sur le carré blanc commence la simulation.

8 Les tris

Le but de cette section est de visualiser différents algorithmes de tri.

Vous allez tout d'abord définir un ensemble de fonctions permettant cette visualisation.

8.1 Fonctions préliminaires

La constante `N` qui définit la taille des tableaux à gérer.

```
#define N 100
```

`void init_croissant(int T[]);` qui initialise `T` avec les valeurs de 1 à 100 stockées en ordre croissant.

`void init_decroissant(T[]);` qui initialise `T` avec les valeurs de 1 à 100 stockées en ordre décroissant.

`void init_alea(T[]);` qui initialise `T` avec les valeur de 1 à 100 stockées en ordre aléatoire.

`void chrono_start();` qui met à zéro et démarre le chronomètre.

`void affiche(int T[],int v)` qui affiche le tableau sous forme de barres verticales et affiche un petit cercle jaune sous la barre d'indice `v` passé en argument. Si `v` ≤ 0 aucun cercle jaune n'est affiché. Cette fonction affiche également la valeur courante du chronomètre.

8.2 Ce qu'il faut faire

Il faut programmes les fonctions de :

— tri à bulle ;

— tri par sélection ;

— tri par insertion ;

Vous devrez appeler la fonction d'affichage au cœur de votre programme pour que l'on puisse visuellement suivre chaque étape de l'algorithme.

8.3 Rappel des algorithmes

Tri à bulle

```
fonction tri_bulle(tableau T, entier n)
     répéter
         aucun_échange <- vrai
         pour j de 0 à n - 2
             si T[j] > T[j + 1], alors
                 échanger T[j] et T[j + 1]
                 aucun_échange <- faux
     tant que aucun_échange = faux
```

Tri par sélection

```
fonction tri_selection(tableau t, entier n)
      pour i de 1 à n - 1
          min <- i
          pour j de i + 1 à n
              si t[j] < t[min], alors min <- j
          échanger t[i] et t[min]
```

Tri par insertion

```
fonction tri_insertion(tableau T, entier n)
      pour i de 1 à n - 1
          x <- T[i]
          j <- i
          tant que j > 0 et T[j - 1] > x
              T[j] <- T[j - 1]
              j <_ j - 1
          T[j] = x
```

A Annexe : types, variables constantes et fonctions disponibles

A.1 Types, Variables, Constantes

Types

```
typedef struct point int x,y; POINT;
typedef Uint32 COULEUR;
typedef int BOOL;
```

Variables

La largeur et la hauteur de la fenêtre graphique :

```
int WIDTH;
int HEIGHT;
```

Ces deux variables sont initialisées lors de l'appel à `init_graphics()`.

Constantes

Déplacement minimal lorsque l'on utilise les flèches :

```
#define MINDEP 1
```

Les constantes de couleur :

```
#define noir    0x000000
#define gris    0x777777
#define blanc   0xffffff
#define rouge   0xff0000
#define vert    0x00ff00
#define bleu    0x0000ff
#define jaune   0x00ffff
#define cyan    0xffff00
#define magenta 0xff00ff
```

Les constantes booléennes :

```
#define TRUE 1
#define True 1
#define true 1
#define FALSE 0
#define False 0
#define false 0
```

A.2 Affichage

Initialisation de la fenêtre graphique

```
void init_graphics(int W, int H);
```

Affichage automatique ou manuel

Sur les ordinateurs lent, il vaut mieux ne pas afficher les objets un par un, mais les afficher quand c'est nécessaire. c'est aussi utilme pour avoir un affichage plus fluide quand on fait des animations.

On a donc deux modes d'affichage, automatique ou non. Quand l'affichage automatique est activé, chaque dessin d'objet l'affiche automatiquement. Quand il n'est pas automatique c'est l'appel à la fonction `affiche_all();` qui affiche les objets.

Pour basculer de l'affichage automatique au non automatique, il y a deux fonctions :

```
void affiche_auto_on();
void affiche_auto_off();
```

Quand on est en mode non automatique l'affichage se fait lorsque l'on appelle la fonction :
`void affiche_all();`
Par défaut on est en mode automatique.

Création de couleur

`COULEUR couleur_RGB(int r, int g, int b);` prend en argument les 3 composantes rouge (`r`), vert (`g`) et bleue (`b`) qui doivent être comprises dans l'intervalle : [0..255].

A.3 Gestion d'événements clavier ou souris

Gestion des flêches

`POINT get_arrow();`
Si depuis le dernier appel à `get_arrow()`, il y a eu G appuis sur la flèche gauche, D appuis sur la flèche droite H appuis sur la flèche haut et B appuis sur la flèche bas.
Le point renvoyé vaudra en `x` $D - B$ et en `y` $H - B$.
Cette instruction est non bloquante, c'est à dire que si aucune flèche n'a été appuyée les champs x et y vaudront 0.

Gestion des déplacements la souris

`POINT get_mouse();` renvoie le déplacement de souris avec la même sémantique que `get_arrow()`. Cette instruction est non bloquante : si la souris n'a pas bougé les champs x et y vaudront 0.

Gestion des clics de la souris

`POINT wait_clic();` attend que l'utilisateur clique avec le bouton gauche de la souris et renvoie les coordonnées du point cliqué. Cette instruction est bloquante.
`POINT wait_clic_GMD(char *button);` attend que l'utilisateur clique et renvoie dans `button` le bouton cliqué :
— `*button` vaut 'G' (pour Gauche) après un clic sur le boutton gauche,
— `*button` vaut 'M' (pour milieu) après un clic sur le boutton du milieu,
— `*button` vaut 'D' (pour Droit) après un clic sur le boutton droit.
Cette instruction est bloquante.

Fin de programme

`POINT wait_escape();` attend que l'on tape Echap et termine le programme.

A.4 Dessin d'objets

`void fill_screen(COULEUR color);` remplit tout l'écran.
`void draw_pixel(POINT p, COULEUR color);` dessine un pixel.
`void draw_line(POINT p1, POINT p2, COULEUR color);` dessine un segment.
`void draw_rectangle(POINT p1, POINT p2, COULEUR color);` dessine un rectangle non rempli. Les deux points sont deux sommets quelconques non adjacents du rectangle
`void draw_fill_rectangle(POINT p1, POINT p2, COULEUR color);` dessine un rectangle rempli Les deux points sont deux sommets quelconques non adjacents du rectangle
`void draw_circle(POINT centre, int rayon, COULEUR color);` dessine un cercle non rempli.
`void draw_fill_circle(POINT centre, int rayon, COULEUR color);` dessine un cercle rempli.

```
void draw_circle_HD(POINT centre, int rayon, COULEUR color);
void draw_circle_BD(POINT centre, int rayon, COULEUR color);
void draw_circle_HG(POINT centre, int rayon, COULEUR color);
void draw_circle_BG(POINT centre, int rayon, COULEUR color);
```
dessinent des quarts de cercle.

`void draw_fill_ellipse(POINT F1, POINT F2, int r, COULEUR color);` dessine une ellipse remplie.

`void draw_triangle(POINT p1, POINT p2, POINT p3, COULEUR color);` dessine un triangle non rempli.

`void draw_fill_triangle(POINT p1, POINT p2, POINT p3, COULEUR color);` dessine un triangle rempli.

A.5 Écriture de texte

L'affichage de texte n'est pas en standard dans SDL. Il faut donc que la librairie `SDL_ttf` soit installée.

La configuration fournie teste (grâce au `Makefile`) si `SDL_ttf` est installée ou pas. Si elle est installée, l'affichage se fait dans la fenêtre graphique sinon il se fait dans la fenêtre shell.

`void aff_pol(char *a_ecrire, int taille, POINT p, COULEUR C);` affiche du texte avec

- le texte est passé dans l'argument `a_ecrire`
- la police est celle définie par la constante `POLICE_NAME` dans `graphics.c`
- la taille est passée en argument
- l'argument p de type POINT est le point en haut à gauche à partir duquel le texte s'affiche
- la COULEUR C passée en argument est la couleur d'affichage

`void aff_int(int n, int taille, POINT p, COULEUR C);` affiche un entier. Même sémantique que `aff_pol()`.

Les fonctions suivantes affichent dans la fenêtre graphique comme dans une fenêtre shell. Commence en haut et se termine en bas :

`void write_text(char *a_ecrire);` écrit la chaîne de caractère passée en argument.

`void write_int(int n);` écrit l'entier passé en argument.

`void write_bool(BOOL b);` écrit le booléen passé en argument.

`void writeln();` renvoie à la ligne.

A.6 Lecture d'entier

`int lire_entier_clavier();` renvoie l'entier tapé au clavier. Cette fonction est bloquante

A.7 Gestion du temps

Chronomètre élémentaire

Ce chronomètre est précis à la microseconde.

`void chrono_start();` déclenche le chrono. Le remet à zéro s'il était déjà lancé.

`float chrono_val();` renvoie la valeur du chrono et ne l'arrête pas.

Attendre

`void attendre(int millisecondes);` attend le nombre de millisecondes passé en argument.

L'heure

`int heure();` envoie l'heure de l'heure courante.
`int minute();` renvoie le nombre de minutes de l'heure courante
`int seconde();` renvoie le nombre de secondes de l'heure courante.

A.8 Valeur aléatoires

`float alea_float();` renvoie un `float` dans l'intervalle $[0; 1[$.
`int alea_int(int N);` renvoie un `int` dans l'intervalle $[0..N[$ soit N valeurs différentes de 0 à $N-1$.

A.9 Divers

`int distance(POINT P1, POINT P2);` renvoie la distance entre deux points.