Introduction à la Programmation

—

Examen final sur machine

—

G1: pascal.monasse(at)enpc.fr
G2: thomas.luka(at)enpc.fr
G3: thomas.belos(at)enpc.fr
G4: laurent.bulteau(at)u-pem.fr
G5: abderahmane.bedouhene(at)enpc.fr
G6: pierre-alain.langlois*(at)*enpc.fr
G7: marcela.carvalho(at)onera.fr

—

10/01/20

—

1 Enoncé

1.1 Bubble Shooter

Nous allons programmer ce jeu qui consiste à lancer des bulles de couleur dans une piscine. Quand la collision d'une bulle crée un ensemble de 3 ou plus bulles connectées de même couleur, ces bulles sont détruites. Si cela déconnecte des bulles du fond de la piscine (ligne du haut), quelle que soit leur couleur, elles sont aussi détruites. Chaque fois qu'un tir ne détruit pas de boules, il y a une chance que les bulles se déplacent d'un rang vers le bas et qu'une rangée de bulles apparaisse au fond de la piscine. Le jeu se termine quand la piscine déborde (une bulle atteint le bord inférieur). Le score est le nombre de bulles détruites. La bulle lancée rebondit sur les bords latéraux ; lors de la collision, elle se range dans la case la plus proche sur une grille hexagonale (chaque case a 6 voisins). Créez un projet Imagine++ avec un fichier contenant le `main`. Prenez le `CMakeLists.txt` d'un projet existant et adaptez-le. Les différentes parties de l'examen (sauf la dernière) sont relativement indépendantes.

Il est plus important de livrer un code clair (commenté et indenté) et qui compile sans warning, même s'il ne répond pas à toutes les questions. Pour cela, vérifiez à chaque étape que votre programme compile et se lance correctement. À la fin de l'examen, nettoyez votre code (indentation...) et vérifiez qu'il compile. Créez alors une archive portant votre nom et numéro de groupe.

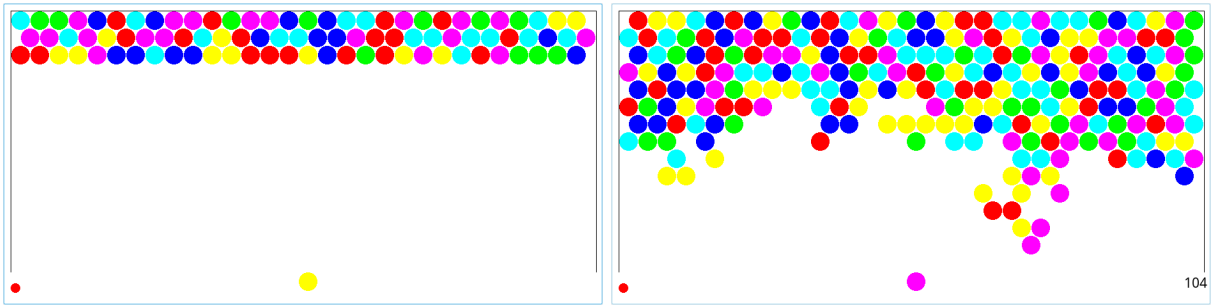

FIGURE 1 – Les bulles sont disposées en quinconce, les cases ayant 6 voisins. Les bords et le fond de la piscine sont matérialisés par des traits noirs, la bulle à lancer est en bas au centre, en bas à gauche la prochaine couleur de bulle est visible, en bas à droite le score. À gauche, configuration initiale de la piscine. À droite, on s'approche du débordement.

1.2 Bases du jeu

1. La fenêtre réserve `marge`= 20 pixels de chaque côté et `margeScore`= 50 pixels en bas. Chaque bulle a un `rayon`=15 pixels, donc elles sont séparées de 2∗`rayon` en largeur et $\sqrt{3}/2$ fois cette taille en hauteur. Votre fonction `main` ouvre une fenêtre de taille adaptée à ces constantes (attention à la disposition en quinconce) pour `width`= 30 par `height`= 15 cases et dessine le cadre de la piscine.
2. Dans des fichiers à part, créer la classe `Piscine` qui stocke les couleurs de bulles, par un entier entre 0 compris et `nbcoul`= 6 exclu, dans un tableau. La piscine prendra ses dimensions en variables, potentiellement on pourrait avoir un jeu avec plusieurs piscines de dimensions différentes. On aura une table de couleurs `coul` associant à un indice sa couleur, et `coul[nbcoul]` sera `WHITE` pour marquer l'absence de bulle.
3. Écrire les constructeur et destructeur de `Piscine`. Le constructeur prend ses dimensions (en cases), et le nombre de rangées remplies de bulles avec chacune une couleur aléatoire.
4. Écrire la méthode `insere_ligne` qui décale les rangées de bulles vers le bas et insère une nouvelle ligne de bulles au fond (haut). Si la ligne du bord (bas) a une bulle, elle renvoie false. On gardera en mémoire une variable `parite` qui vaudra 0 ou 1 suivant que la première ligne est collée à gauche ou à droite. Utiliser cette méthode dans le constructeur pour remplir le nombre choisi par le programmeur de rangées initiales.
5. Écrire une méthode `pos_case` convertissant un indice (x, y) de case en coordonnées pixel (x_p, y_p) de son centre. C'est une transformation affine, attention x_p dépend de y et de `parite` alors que y_p dépend seulement de y.
6. Définir une méthode `dessine` de la piscine. Tracer en fait des cercles de `rayon`−1 pour éviter des chevauchements.
7. Écrire une fonction `test_affiche` qui montre une piscine initialement vide et insère une ligne à chaque clic jusqu'à remplissage. [1]

1.3 La bulle mobile

8. Définir une classe `Bulle` comprenant un indice de couleur, des coordonnées (x, y) (en pixels) et une vitesse (dx, dy), ces valeurs en float pour éviter des arrondis. Ses méthodes `dessine` et `efface` fonctionnent comme les bulles fixes dans la piscine.
9. Écrire une méthode `Piscine::init_bulle` qui initialise une variable interne de type `Bulle` avec une couleur aléatoire et position initiale fixe (x_0, y_0) en bas au centre. Pour la vitesse, elle attend un clic de l'utilisateur en un point (x, y) avec $y < y_0$ (vitesse vers le haut) et prend le vecteur directeur de norme 2 de (x_0, y_0) à (x, y).
10. La méthode `Piscine::bouge_bulle` efface la position courante de la bulle, avance d'un pas de vitesse avec rebond eventuel sur les bords latéraux et affiche la nouvelle position. Elle renvoie false si bulle atteint le fond du bassin.
11. Démontrer le bon fonctionnement par une fonction `test_bulle` qui construit une piscine vide et fait bouger la balle dans la piscine jusqu'à atteindre le fond.
12. La méthode `Piscine::trouve_case_bulle` implémente la fonction réciproque de `pos_case`, elle convertit les coordonnées pixel de la balle et retourne la case la plus proche.
13. Cette méthode est utilisée pour "figer" la bulle mobile dans le tableau interne de la piscine quand la bulle atteint le fond.

1.4 Destructions de bulles

14. Implémenter une méthode `Piscine::voisin` renvoyant la case voisine numéro i $(0 \leq i < 6)$ de son paramètre (x, y). Outre les voisins sur la même ligne, on a aussi les voisins au-dessus et en-dessous (qui dépendent de la parité). Ne faîtes pas du code spaghetti avec des if / else, utilisez des tables de vecteurs (dx, dy) dépendant de la parité (max 10 lignes de code).
15. Pour détecter une collision de la bulle mobile avec les bulles de la piscine, `bouge_bulle` regarde la case courante de la bulle mobile et teste si le centre d'une case voisine est à distance moins de 2∗`rayon`. On fige alors la bulle dans sa case et la méthode renvoie `false`.

1. Une fois qu'elle fonctionne, vous pouvez mettre l'appel en commentaire pour ne pas perdre de temps en testant la suite, mais n'oubliez pas de rétablir l'appel à la fin.

16. Les fonctions récursives suivantes font des explorations et nécessitent un tableau de `bool vu` pour se souvenir des cases déjà parcourues. Le créer/détruire dans le constructeur/destructeur et écrire une méthode `init_vue` mettant tout à false.
17. La méthode `Piscine::voisins_non_vus` est récursive et compte les voisins non explorés de même couleur et leurs voisins, etc. Le but est de compter le nombre de cases de la composante connexe iso-couleur de la bulle qui se fige. Si cette taille est au moins 3, on va détruire la composante connexe dans la fonction suivante :
18. `Piscine::detruis` fais disparaître la composante connexe iso-couleur de la bulle qu'on vient de figer. Quand une bulle disparaît, on peut la colorer en noir un court instant avant de l'effacer.
19. La méthode `Piscine::detruis_residus` va enlever les bulles non connectées au fond : pour chaque bulle du fond, on explore la composante connexe (indépendamment de la couleur) de bulles par une fonction récursive `explore` (5-10 lignes de code). On parcourt finalement le tableau des cases pour détruire celles non explorées par cette procédure.
20. Intégrer ces fonctions lors de la collision dans `bouge_balle`.

1.5 Le jeu final

21. Intégrer le comptage du score dans `Piscine` et son affichage.
22. La couleur de la prochaine boule est tirée au hasard et affichée dans `init_bulle`.
23. Le jeu fait des tirs successifs de bulles. À chaque tir sans destruction (comparer le score avant et après), on détermine s'il faut insérer une ligne de bulles dans la piscine ou non. On part d'une probabilité à 0.2 et elle est multipliée par 1.2 à chaque fois que finalement on n'insère pas de ligne. Par contre, elle retourne à 0.2 une fois la nouvelle ligne insérée.