Projet de méthodologie de conception et de programmation : « Programmons un jeu vidéo »

Level 4

Avant-propos : Si ce n'est déjà fait, penser à restructurer votre code afin que celui-ci soit modulaire : utilisation de structures et de différents fichiers !

Objectifs du niveau : Dans ce niveau, nous nous acheminons dans la joie et la sérénité vers la fin de la version de base du projet (dans le level 5, il ne restera "plus qu'à" s'occuper de la connexité des bulles). En particulier, nous allons :

- lancer des bulles de couleurs différentes, choisies aléatoirement,
- gérer les collisions entre bulles et permettre leur disposition en quinconce,
- réinitialiser le plateau de jeu et afficher "game over" (dans la console linux) lorsqu'une bulle atteint la ligne du bas du plateau.

Lors de ce niveau, vous êtes fortement encouragés à écrire des petites fonctions auxilliaires pour les calculs de position, les tests de collision, etc.

Bulles de couleurs différentes. Jusqu'ici nous avons considéré une seule bulle de couleur bleue, représentée par une `SDL_Surface` que nous appellerons ici `*bub`. Nous allons à présent autoriser le lancement de bulles de différentes couleurs, choisies aléatoirement (pour le moment nous continuons à afficher en bleu les bulles collées à la paroi supérieure). Pour cela, vous devez :

- remplacer la variable `*bub` par un tableau de `SDL_Surface` (`*bub[NUM_COLORS]`, où `NUM_COLORS=8`),
- remplir ce tableau en utilisant les huit *sprites* fournis (`bub_black.bmp`, `bub_blue.bmp`, ...),
- utiliser une variable, par exemple `current_col`, pour conserver le numéro de couleur de la bulle courante (c'est-à-dire la bulle posée sur le lanceur ou en mouvement). `current_col` est un entier compris entre 1 et `NUM_COLORS`, choisi aléatoirment à chaque fois qu'on (re)positionne une bulle sur le lanceur. Utilisez la fonction `rand()` et le modulo (`%`) pour générer cet entier ;
- à chaque itération de la boucle principale, afficher la SDL_SURFACE correspondant à la couleur courante (`*bub[current_col-1]`).

Disposition des bulles en quinconce. Comme nous l'avons vu dans le level 3, les bulles ne peuvent occuper que certaines positions sur le plateau de jeu. Nous avons introduit au level 3 le tableau `bub_array`, qui permet d'indiquer la présence de bulle à ces positions, 0 s'il n'y a pas de bulles, 1 s'il y en a une. Alors que nous ne nous sommes intéressés qu'à la première ligne du tableau, nous voyons à présent comment traiter le tableau tout entier.

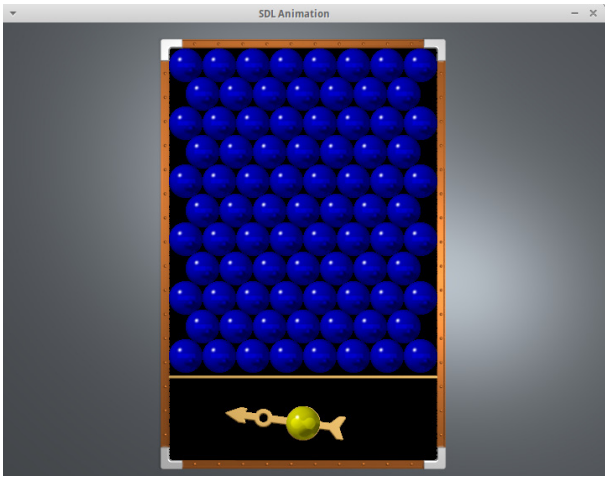
Nous aurons besoin de convertir fréquemment des indices du tableau en coordonnées écran (centres des bulles pour permettre des calculs de distance au centre). Pour permettre de ne faire cette conversion qu'une seule fois, nous proposons d'utiliser un second tableau :

```
int bub_array_centers[BUB_NY][BUB_NX][2],
```

qui contiendra les coordonnées des centres des bulles correspondant aux indices du tableau. Remplissez ce tableau avant d'entrer dans la boucle principale. Attention : les lignes impaires n'ont que `BUB_NX-1` positions à définir.

Lors du calcul de ces positions, on souhaite que deux lignes consécutives soient collées, ce qui implique un petit calcul de trigonométrie. Tous calculs faits, on obtient que l'écart entre deux lignes doit être de $\sqrt{3}/2$ fois la taille de la bulle. Avec `BUB_SIZE=40`, on obtient un écart que l'on arrondit à 35 pixels.

Testez vos positions en affichant une bulle à chaque position de `bub_array_centers` (cf. illustration page suivante). Dans la suite, on suppose que le tableau `bub_array` est rempli de 0 quand il n'y a pas de bulles, ou du numéro de couleur de la bulle quand il y en a une (d'où l'intérêt de commencer la numérotation des bulles à 1).



Gestion des collisions entre bulles. Cette gestion se fera en deux trois temps :

1. détection d'une éventuelle collision,
2. si une collision est détectée, détection de la case vide la plus proche de la bulle lancée,
3. remplacement de 0 par `current_col` dans la case vide détectée.

Pour détecter l'éventuelle collision, on considère les coordonnées des centres des bulles présentes sur le plateau (en utilisant les tableaux `bub_array` et `bub_array_centers`). On calcule les distances dx et dy (distances en x et en y) du centre de la bulle courante à chacune de ces coordonnées. Si le cas $\sqrt{dx^2 + dy^2} < \alpha * \texttt{BUB\_SIZE}$ se produit, une collision est détectée. α est un coefficient ≤ 1 qui permet aux bulles de se "faufiler" un peu mieux entre les bulles présentes sur le plateau de jeu. Essayez avec $\alpha = 1$ pour vous rendre compte du problème et ajutez ce paramètre de manière empirique (par exemple, $\alpha = 0.87$ ne nous a pas déçu !).

Pour trouver la case vide la plus proche en cas de collision, on parcourt les coordonnées des centres des cases ne contenant pas de bulles, et la case telle que $\sqrt{dx^2 + dy^2} \leq \texttt{BUB\_SIZE}/2$ est la case recherchée.

Détection d'une fin de partie. Supprimez toutes les bulles du plateau (un jeu d'enfant grace au tableau `bub_array`) quand la bulle courante atteint la ligne du bas du plateau (attention : c'est un peu subtil car la bulle est déjà sous cette ligne au départ !).

N'oubliez pas de faire un commit (pour l'un des membres du binôme) et un update (pour l'autre membre) de votre code avant de passer au niveau 5.

Achievement : level 4 finished !