On cherche à développer un jeu d'adresse, dans l'esprit des jeux en bois traditionnels dont le but est de faire passer une bille dans un labyrinthe tout en évitant les trous qui parsèment le plateau.

1 Présentation du jeu et de ses éléments

On présente ici la version de base du jeu, tel qu'il sera à la fin de la Section 3 ; la Section 4 propose des pistes pour complexifier et améliorer le jeu et son interface.

Pour cette version de base, l'écran apparaît vide au démarrage de l'application. La joueuse peut ensuite taper plusieurs fois sur l'écran afin de placer les différents éléments du jeu :

— son premier toucher fera apparaître l'`Objectif`,

— son deuxième toucher fera apparaître la `Bille`,

— chacun de ses touchers après le deuxième fera apparaître un `Obstacle`.

Cette mise en place des éléments de jeu est traitée dans la Section 2. La Figure 1 illustre la situation après un premier toucher en haut à gauche, un deuxième en bas à droite, et une vingtaine de touchers entre les deux.

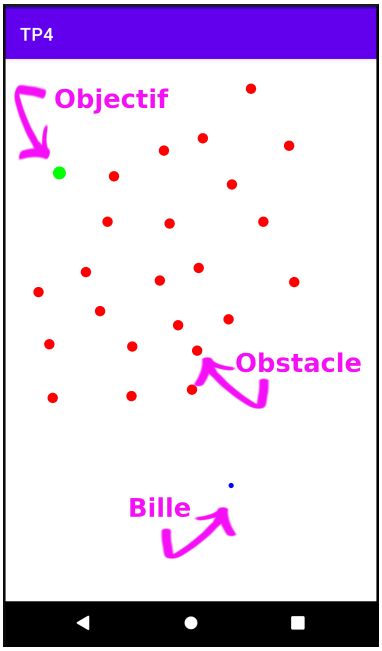

FIGURE 1 – Aperçu du résultat attendu en fin de Section 2.

À partir du moment où la `Bille` est créée, le jeu détectera tout mouvement de la joueuse, et la `Bille` roulera en suivant la gravité.

Si la `Bille` touche un `Obstacle`, la partie est perdue. Si la `Bille` touche l'`Objectif`, la partie est gagnée. Dans chacun des deux cas, un message de fin de partie s'affiche et le jeu est réinitialisé.

La gestion du mouvement de la `Bille` et les conditions de victoire et de défaite font l'objet de la Section 3.

2 Mise en place du `Terrain`

Puisque les classes `Bille`, `Obejectif` et `Obstacle` partagent beaucoup de caractéristiques, on va les faire hériter d'une classe mère : `Element`.

Question 1 Créez une classe `Element`, dont les attributs sont
- un `float[]` de taille 2, contenant les coordonnées du centre de l'`Element`,
- un `float` indiquant son rayon,
- un `Paint`, indiquant comment il doit être dessiné.

On dotera cette classe des constructeurs et accesseurs ~~idoines~~ appropriés.

Question 2 Étendez cette classe en trois classes filles : `Bille`, `Objectif` et `Obstacle`. Attribuez une couleur par défaut et un rayon par défaut à chacune de ces classes, que vous pourrez utiliser dans des constructeurs partiels.

Maintenant que les éléments de jeu sont traités, on va se pencher sur la question du plateau : il sera représenté par un `Terrain`.

Question 3 Créez une classe `Terrain` qui hérite de `View`, et ayant comme nouveaux attributs
- une `Bille`,
- un `Objectif`,
- une `ArrayList` d'`Obstacle`.

À la création, leurs valeurs seront respectivement `null`, `null` et une `ArrayList` vide.

Du côté du layout, on créera un `Terrain` qui sera le seul enfant du `ConstraintLayout` racine.

Question 4 Réimplémentez la méthode `View::onDraw()` pour qu'en se dessinant, un `Terrain` dessine chacun de ses `Element` existants.

Question 5 Réimplémentez la méthode `View::onTouchEvent()` de `Terrain` pour que chaque toucher du joueur crée un `Element` selon les règles détaillées en Section 1. Au premier toucher, il faudra donc qu'on crée un `Objectif` à l'endroit où le doit a été posé. Au second toucher, ce sera au tour de la `Bille` d'être crée et placée, puis au tour des `Obstacle` après ça.

On fera en sorte qu'un `Element` placé apparaisse immédiatement.

À ce stade, on devrait obtenir un résultat similaire à celui présenté en Figure 1. On va maintenant passer à l'implémentation de la mécanique de jeu.

3 Mouvement de la `Bille`

Pour éviter des rotations d'écran incessantes, dans la mesure où l'écran sera le plus souvent horizontal, on ajoutera la ligne suivante dans le Manifest, comme attribut de `MainActivity` :

```
android:screenOrientation=''portrait''
```

Question 6 Dans `MainActivity::onCreate()`, vérifiez la présence d'un accéléromètre. Dans le cas contraire, on pourra se contenter de fermer poliment l'application.

Question 7 Écrivez une méthode `Terrain::roll()`

```
public boolean roll(float[] deplacement);
```

chargée de modifier la position de la `Bille`. L'argument est un tableau contenant deux `float` : la première case contient le déplacement horizontal (positif en allant vers la droite), et la deuxième le déplacement vertical (positif en allant vers le bas).

Cette méthode devra vérifier que ce déplacement ne fait pas sortir la `Bille` de l'écran. Dans le cas contraire, on arrêtera la `Bille` quand son bord touche le bord du `Terrain`.

Pour l'instant, on se contentera de renvoyer `true`. La valeur de retour permettra, plus tard, de faire savoir quand la `Bille` a rencontré un `Obstacle` ou l'`Objectif`.

On fera en sorte que le `Terrain` se redessine après le déplacement de la `Bille`.

Question 8 Dans `MainActivity`, créez un `SensorEventListener` qui écoute l'accéléromètre et appelle `Terrain::roll()` avec les bonnes valeurs.

On fera attention à faire appel à `registerListener()` et `unregisterListener()` aux moments propices du cycle de vie de l'activité. Pour éviter un monitorage trop saccadé, on pourra utiliser la constante `SensorManager.SENSOR_DELAY_GAME` comme troisième argument à `registerListener()`.

Testez votre application, et vérifiez en particulier que le comportement de la `Bille` contre les bords du `Terrain` est cohérent. On se rendra peut-être compte que la `Bille` roule trop lentement pour que le jeu soit intéressant. On pourra alors multiplier le vecteur de déplacement par une constante `float` (par exemple, `3f` ou `4f`), qu'on appelera `difficulte`.

Il ne nous reste maintenant plus qu'à intégrer à l'application la condition de victoire (la `Bille` touche l'`Objectif`) ainsi que la condition de défaite (la `Bille` touche un `Obstacle`).

Question 9 Écrivez une méthode `Element::chevauche()`

```
public boolean chevauche(Element element);
```

indiquant si les deux `Element` se chevauchent. On se souviendra de ses cours de géométrie :

$$\text{distance}((x,y),(x',y')) = \sqrt{(x-x')^2+(y-y')^2}\,.$$

Question 10 Écrivez une méthode `Terrain::checkPosition()` qui renvoie un `boolean` indiquant si la `Bille` chevauche un `Obstacle` ou l'`Objectif`. Le cas échéant, cette méthode devra afficher un `Toast` indiquant respectivement "Game Over" ou "Victoire !".

Modifiez la méthode `roll()` pour qu'elle fasse appel `checkPosition()`. En cas de chevauchement, faites en sorte que le jeu reparte de zéro (avec un écran vide).

Vérifiez que votre application se comporte correctement en cas de collision entre la `Bille` et un autre `Element`.

4 Pour aller plus loin

Maintenant que notre application est fonctionnelle, on cherche à en améliorer certains aspects.

Question 11 Lors de la mise en place, on veut que la joueuse puisse choisir la taille des `Obstacle` à ajouter, en appuyant plus ou moins longtemps sur l'écran.

En faisant appel à `MotionEvent::getEventTime()`, qui retourne un `long` indiquant la date, en millisecondes, de déclenchement de ce `MotionEvent`, modifiez `Terrain::onTouchEvent()` pour que la taille de l'`Obstacle` soit fonction affine de la durée du toucher. On choisira des paramètres raisonnables.

Question 12 De manière similaire, faites en sorte que la `difficulte` de la Question 8 soit déterminée par la durée du toucher lors de la création de la `Bille`.

Question 13 (☠) On veut maintenant travailler sur la physique de la `Bille`, et notamment gérer son accélération. Modifiez la classe `Bille` pour que les données de l'accéléromètre agissent sur l'accélération de la `Bille`, et non plus sur sa vitesse (comme c'était le cas jusqu'à présent).

On devrait ainsi observer un effet d'*emballement* de la `Bille` en laissant l'appareil dans une inclinaison fixe.

Enfin, les plus courageux et courageuses implémenteront un effet de rebond quand la `Bille` tape sur un des bords du `Terrain`.