COLLÈGE DE
BOIS-DE-BOULOGNE

TP2 – Squelette Espiègle

SIM – A22 – Développement de programmes dans un environnement graphique

Nicolas Hurtubise

Figure 1: Logo du jeu

Contexte

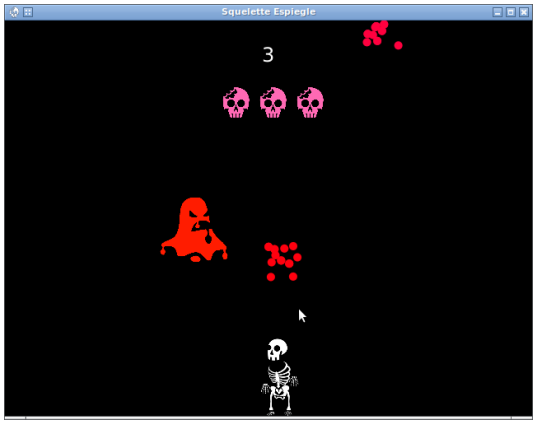
Le TP final consiste à programmer un autre jeu en interface graphique, toujours avec la librairie *JavaFX*.

Description du jeu

Vous incarnez un **squelette espiègle** qui chasse des monstres pour son souper.

Étant un squelette gourmand, vous ne pouvez pas vous permettre de laisser trop de monstres passer... À chaque monstre qui réussit à se sauver, on perd une vie. Au bout de 3 vies perdues, la partie est finie.

Le squelette se contrôle avec le clavier. Le jeu continue à l'infini, ou jusqu'à ce que la partie soit perdue (selon lequel arrive en premier).



Déroulement du jeu

Au début du jeu, le niveau est affiché pendant 3 secondes : **NIVEAU 1**.

Une fois les 3 secondes écoulées, le texte disparaît et le premier monstre apparaît à l'écran.

Afin de capturer les monstres, il faut **envoyer une boule de magie** en appuyant sur Espace. Chaque monstre capturé fait monter le score de +1. Cependant, les monstres se déplacent : si un monstre réussit à sortir de l'écran sans se faire capturer, il est considéré comme perdu, et on **perd une vie**. Au bout de 3 monstres ratés, le jeu se termine et la partie est perdue.

Quand la partie est perdue, on affiche le message **FIN DE PARTIE** en rouge pendant 3 secondes, puis on doit automatiquement revenir à l'accueil quand le temps est écoulé.

En tout temps, le score est affiché à l'écran et le nombre de **vies** restantes (initialement 3) est représenté par des têtes de squelettes en **rose** sous le score.

À tous les 5 monstres *capturés*, on augmente le niveau du jeu : les monstres arrêtent d'apparaître pendant 3 secondes et l'écran affiche le prochain numéro de niveau, ex.: **NIVEAU 2**.

Au début, il y a seulement des monstres simples, mais à mesure qu'on monte de niveau, des *monstres spéciaux* commencent à apparaître.

Squelette



Le squelette est modélisé par un rectangle de largeur $48px$ et de hauteur $96px$. Au repos, le Squelette est affiché avec l'image `stable.png` fournie. Lorsqu'il marche, on doit plutôt l'animer en alternant les images `marche1.png` et `marche2.png`. L'animation doit avoir un **framerate** de 10 images par seconde (*à lire là-dessus : notes de cours `JavaFX - Animations partie 1`*).

Ces images montrent le squelette qui regarde vers la droite. Si le squelette regarde plutôt à gauche, on devra **changer le sens des images** dans l'animation (regardez les fonctions disponibles dans la classe `ImageHelpers` fournie, il y a du code pour vous aider à faire ça).

La gravité à appliquer au squelette sera une accélération de $1200px/s^2$ vers le bas.

On peut **sauter** avec la flèche du haut : un saut donne instantanément une vitesse de $600px/s$ vers le haut. Notez qu'il est impossible de sauter quand on est déjà dans les airs.

Lorsqu'on appuie sur les flèches gauche et droite, cela a pour effet de donner instantanément au squelette une *accélération en x* de $1000px/s^2$ vers la gauche ou vers la droite (respectivement). Si aucune des deux touches n'est appuyée, on doit donner une accélération **en sens inverse** de pour faire en sorte que le squelette ralentisse automatiquement. Pour éviter une vitesse trop élevée, on devrait fixer une vitesse maximale à $300px/s$ vers la gauche ou vers la droite (*à lire là-dessus : notes de cours `JavaFX - Animations partie 5`*).

Le squelette ne peut pas sortir de l'écran par les côtés : lorsqu'il touche la gauche ou la droite de l'écran, il **rebondit** dans l'autre direction.

Monstres

À toutes les 3 secondes, des monstres apparaîssent soit depuis la gauche, soit depuis la droite de l'écran. Les monstres qui commencent à droite vont vers la gauche, les monstres qui commencent à gauche vont vers la droite. Leur position y initiale est tirée au hasard entre $\frac{1}{5}$ et $\frac{4}{5}$ de la hauteur du jeu.

Pour les dessins, utilisez les images `monstres/0.png` à `monstres/7.png` dans le dossier d'images fournies.

L'image des monstres doit être inversée au besoin pour les faire "regarder" dans la direction où ils se déplacent. Chaque nouveau monstre a également une couleur aléatoire. Pour manipuler les images fournies (attribuer une couleur aléatoire et inverser l'image), regardez les fonctions disponibles dans la classe `ImageHelpers` fournie.

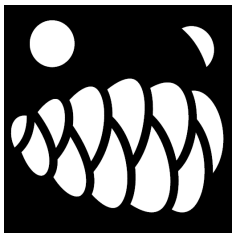
Pour simplifier les collisions, on va considérer que tous les monstres sont représentés par des cercles d'un rayon choisi au hasard entre $20px$ et $50px$ (donc d'un *diamètre* entre $40px$ et $100px$). Techniquement, l'image affichée pour un monstre pourrait déborder un peu à l'extérieur du cercle de collisions : c'est correct.

La gravité pour les monstres normaux doit donner une accélération de $100px/s^2$ vers le bas. La **vitesse verticale** initiale des monstres est tirée au hasard entre 100 et 200 px/s vers le haut de l'écran. La **vitesse horizontale** des monstres doit suivre la fonction :

$$vitesse(numeroNiveau) = 100 \times numeroNiveau^{0.33} + 200 \qquad px/s$$

Plus on avance dans les niveaux, plus les monstres iront vite.

Monstres spéciaux



À partir du niveau 2, un monstre spécial doit apparaître à toutes les 5 secondes en plus des monstres normaux. Un monstre spécial est soit un **Oeil**, soit une **Bouche** (50% de probabilité pour chacun).

Ces monstres spéciaux ont une façon différente des autres de se déplacer :

Oeil : l'Oeil est représenté par l'image `oeil.png`. Les Yeux vont à une vitesse de $1.3 \times vitesse(niveau)$ – voir la fonction définie plus haut – et vont en ligne droite (sans gravité, $acceleration_y = 0$).

Un Oeil avance et recule en alternant : il avance pendant 0.5s, puis recule pendant 0.25s, puis avance pendant 0.5s, ... jusqu'à avoir dépassé l'autre côté de l'écran.

Bouche : la Bouche est représentée par l'image `bouche.png`. Elle avance à la même vitesse que les monstres normaux, sans subir de gravité ($acceleration_y = 0$), mais oscille elle verticalement avec une amplitude de $50px/s$, **selon une fonction Sinus**. Inspirez-vous du code des flocons vu en classe pour faire ça.

Magie

Pour capturer les monstres, le squelette peut lancer des **boules de magie** (en appuyant sur Espace). Les boules de magie ne subissent pas d'accélération et vont en ligne droite vers le haut, à une vitesse de $300px/s$. Le squelette peut lancer maximum une boule à toutes les $0.6s$ (sinon on pourrait juste garder Espace enfoncé et gagner le jeu en tirant à l'infini!).

Au niveau des détections de collisions avec les monstres, on va considérer qu'une boule de magie est un cercle d'un **rayon de 35px** (*attention:* rayon=35 `=>` diamètre total = 70).

Au niveau de l'**animation** de la magie, on va se faire de quoi d'intéressant : une petite simulation de **particules chargées à l'intérieur du cercle**.

Plutôt que de réellement dessiner le cercle de rayon 35, on va le remplir de plusieurs particules de *rayon=5* (donc de diamètre total = 10) dont les mouvements seront expliqués plus bas.

Chaque boule de magie a sa propre couleur pour les particules, générée au hasard à chaque fois.

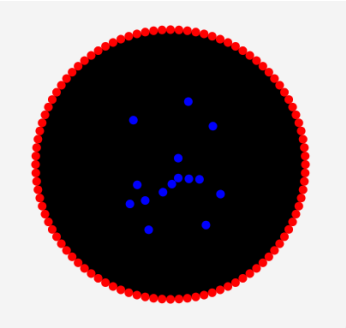
Formules de physique pour les particules chargées

La **loi de Coulomb** dit...

$$F_{électrique} = \frac{K * q_1 * q_2}{distance(particule_1, particule_2)^2}$$

où K est une constante, q_1 est la charge électrique de la première particule, q_2 est a charge de la deuxième particule, et la distance entre les deux est calculée avec la formule de pythagore.

À l'intérieur d'une boule de magie, on va simuler les particules suivantes :



On va avoir :

- **100 particules chargées invisibles, fixées et réparties tout autour de la circonférence du cercle**. Ces particules ne bougent pas et ne sont **pas dessinées** (elles sont en rouge dans le dessin d'exemple, mais on ne les voit pas dans le jeu). Elles servent uniquement à rendre intéressant le mouvement des 15 autres.
- **15 particules libres**, qui vont subir les forces de toutes les autres particules (elles sont en bleu dans le dessin en haut, mais elles changent de couleur au hasard dans le jeu)

On a donc un total de 115 particules. Les particules sont toutes chargées à `q = 10`, et on choisira une constante `K = 30`.

Puisque seules les 15 particules libres sont dessinées, on devrait voir un joli petit chaos de particules qui bougent dans tous les sens, ce qui donnera un aspect mystique à nos boules de magie.

De temps en temps, une particule libre va se sauver de la boule de magie : dans ces cas-là, on ramène la particule au centre de la boule et on remet sa vitesse à zéro pour lui faire reprendre sa simulation.

Pour réaliser cette simulation, à chaque `update()` d'une boule de magie, on devra :

1. **Recalculer les forces** pour chaque particule présente dans la boule
   - Pour chacune des 15 particules libres, on calcule la force électrique causée par chacune des 114 autres particules
   - On prend la somme de tout ça comme étant la force totale à appliquer sur la particule
2. **Mettre à jour l'accélération** de chaque particule
   - Puisqu'on n'a pas de masses (on va assumer que les masses sont toutes égales à 1), on a $F = ma = 1 * a = a$
   - La force est directement égale à l'accélération
   - Cette étape est plutôt facile `:^)`
3. **Mettre à jour la vitesse** selon l'accélération calculée à partir des forces, **puis la position** selon la vitesse, pour chaque particule
   - Même chose que dans nos exemples de balles, inspirez-vous du code vu en classe

Quelques précisions importantes

1. Assurez-vous de ne <u>PAS</u> tester une particule avec elle-même

On ne veut pas qu'une particule s'applique une force à elle-même...

2. Vitesses trop élevées

Si la distance entre deux particules devient trop petite, la vitesse va exploser (on divise par presque zéro...)

Pour éviter ça, **forcez une distance minimum** dans les calculs :

```
if (distance < 0.01)
    distance = 0.01;
```

Forcez également une vitesse toujours entre $-50px/s$ et $+50px/s$ en x et en y.

3. Plusieurs boules de magie

Notez que s'il y a plus d'une seule *boule de magie*, les particules des différentes boules ne s'entre-influancent pas... Le comportement de chaque boule est indépendant de celui des autres.

4. Gardez les positions des particules
PAR RAPPORT AU CENTRE DE LA BOULE DE MAGIE

Puisque le système de particules bouge au complet avec la boule de magie, gardez seulement le x/y des particules *par rapport au centre de la boule*. Une particule en $x = 0, y = 0$ voudra dire *"la particule est pile poil au centre de la boule de magie, peu importe où se trouve la boule de magie sur l'écran"*.

Au moment de dessiner les particules, plutôt que de faire :

```
Pour Chaque particule
    Dessiner Un Cercle à (particule_x; particule_y)
```

Vous allez plutôt faire :

```
Pour Chaque particule
    Dessiner Un Cercle à (bouleMagie_x + particule_x; bouleMagie_y + particule_y)
```

Ça va simplifier tous les calculs :

- La vitesse de la boule de magie va s'appliquer automatiquement sur les particules à l'intérieur
- On pourra vérifier si une particule est encore à l'intérieur du cercle avec la petite formule :

$$particule_x^2 + particule_y^2 < rayon_{bouleDeMagie}^2$$

Notez: les positions (x, y) des particules vont être négatives dans certains cas, quand une particule est à gauche du centre ou en haut du centre.

5. Comment faire ça en 2D

La loi de Coulomb donnée plus haut fonctionne avec une force *calculée en 1D*, le long de la ligne qui relie les deux particules.

Dans le contexte d'un jeu en 2D, cette force est la *magnitude* du vecteur force. On doit trouver les proportions x et y du vecteur unitaire qui représente la direction 2D de la force, ce qui nous donnerait la quantité de force en x et la quantité de force en y :

```
double distance = /* TODO: distance avec Pythagore */

// Différence de position entre les deux particules
double deltaX = particule1x - particule2x;
double deltaY = particule1y - particule2y;

// On normalise : ça revient à la vecteur unitaire
// dans la direction de la force
// TODO: toujours utiliser une distance d'au moins 0.01
double proportionX = deltaX / distance;
double proportionY = deltaY / distance;

double forceElectrique = /* TODO: loi de Coulomb */
// On calcule la proportion de la force en X vs en Y
double forceEnX = forceElectrique * proportionX;
double forceEnY = forceElectrique * proportionY;

// TODO: la force est égale à l'accélération avec nos simplifications
```

Débogage du jeu

Afin de simplifier le débogage du jeu, on devrait pouvoir :

- Appuyer sur la touche `H` pour faire monter le niveau de +1
- Appuyer sur la touche `J` pour faire monter le score de +1
- Appuyer sur la touche `K` pour faire regagner une vie
- Appuyer sur la touche `L` pour faire perdre la partie

Interface

La fenêtre du programme doit avoir une largeur de $640px$ et une hauteur de $480px$.

- La fenêtre ne doit **pas** être *resizable* : la taille de la fenêtre est fixée au début et elle ne peut pas être redimensionnée avec la souris
- La fenêtre doit avoir en guise de petit logo en haut à gauche l'image `squelette.png` fournie
- La fenêtre doit porter le titre "Squelette Espiègle"

Afin d'avoir un programme plus complet que pour le TP1, ce programme est constitué de 3 scènes.

Écran d'accueil

Lorsqu'on ouvre le programme, la scène affichée doit être l'écran d'acceuil. Cette scène montre :

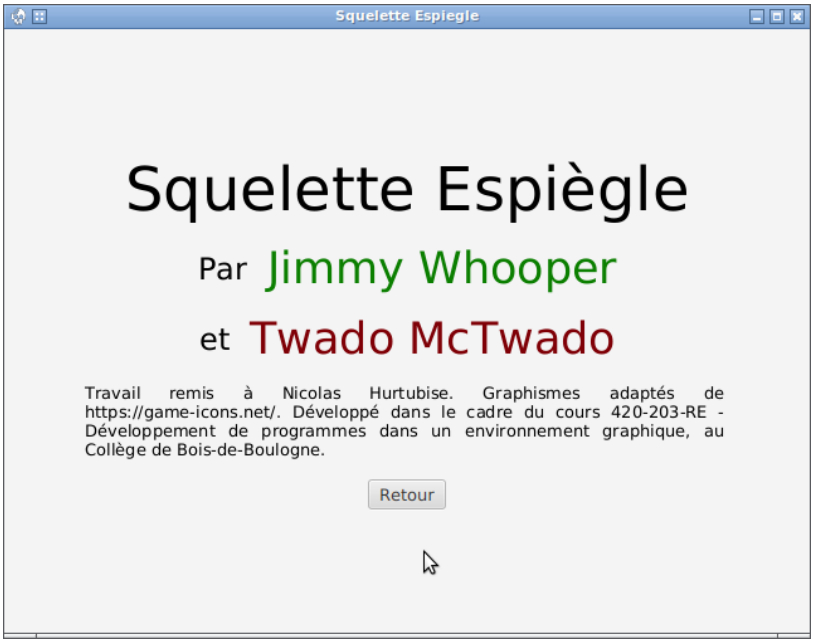
1. Le logo du jeu (`logo.png`)
2. Un bouton *Jouer!*
3. Un bouton *Infos*

Le bouton *Jouer!* passe à la scène du jeu et démarre une nouvelle partie.

Le bouton *Infos* permet de passer à la scène des *Informations* sur le programme.

=> Si on appuie sur Escape dans cette scène, on doit fermer le programme

Écran Infos



Cette scène doit afficher :

- Le titre du jeu en très gros
- Les noms des membres de l'équipe, affichés avec des *couleurs choisies au hasard à chaque fois*
- Un petit texte explicatif (fiez-vous à la capture d'écran)
- Un bouton pour retourner à l'accueil

=> Si on appuie sur Escape dans cette scène, on doit retourner à l'accueil

Scène de Jeu

La scène de jeu contient seulement un canvas (la fenêtre de jeu), qui doit occuper toute la fenêtre.

En tout temps, on devrait voir le score actuel (nombre de monstres capturés), ainsi que le nombre de vies restantes juste en dessous, sous la forme de petites têtes de squelettes (image: `squelette.png` fournie).

=> Si on appuie sur Escape dans cette scène, on doit retourner à l'accueil

Code & Design Orienté Objet

Ce sera à vous de choisir le découpage en classes optimal pour votre programme. Faites bon usage de l'orienté objet et de l'héritage lorsque nécessaire.

Vous **devez** utiliser une architecture du type *Modèle-Vue-Contrôleur* tel que vu en classe et vous serez évalués là-dessus.

Réfléchissez principalement à ce qui constitue votre **Modèle** pour ce jeu : la grosse majorité de vos classes devraient s'y retrouver.

Note sur la mémoire utilisée

Si votre code crée des instances d'objets pour représenter des éléments qui peuvent sortir de l'écran (ex.: les boules de magie, monstres, ...), assurez-vous ne de pas garder en mémoire des objets qui ne sont plus utilisés.

Autrement dit, assurez-vous de ne pas garder dans un tableau ou dans un ArrayList un Monstre ou un autre objet qui ne serait plus jamais affiché de tout le reste du jeu.

Éléments fournis

Une petite classe vous est fournie, pour vous aider à manipuler les images : `ImageHelpers.java`. Il s'agit d'une classe avec quelques méthodes statiques. Vous pouvez aller lire la JavaDoc pour comprendre ce qu'elle permet de faire.

Aucun autre code n'est fourni pour démarrer le projet, mais vous pouvez vous inspirer des exemples de code vus en classe lors des différents chapitres sur les Animations en JavaFX.

Les images nécessaires au programme sont fournies et sont basées sur différentes images du site https://game-icons.net/.

Si vous préférez utiliser d'autres images à la place, vous pouvez le faire `:-)`

Conseils pour bien commencer

1. Travaillez ensemble, surtout au début

Certaines parties du TP se séparent bien en équipes, d'autres, moins bien.

Au début du projet, je vous recommande de travailler en **programmation par binômes**, à deux sur le même ordinateur.

Si ça se passe très très bien, vous pouvez essayer de vous séparer les tâches (en gardant ça équitable entre les membres), mais sentez-vous libres de continuer le travail à deux si vous préférez ça.

2. Allez-y par étapes

Commencez par faire marcher une première étape relativement simple, par exemple : faire bouger le squelette au clavier, avec une seule image pour commencer.

Une fois que ça marche et que c'est bien testé, ajoutez un autre petit truc, par exemple : les monstres simples.

Une fois que ça marche et que c'est bien testé, ajoutez encore un autre petit truc, par exemple : le nombre de vies restantes affiché à l'écran, et qui diminue quand un monstre passe d'un côté à l'autre de l'écran.

Une fois que ça marche et que c'est bien testé, ajoutez encore autre chose, par exemple : une boule de magie simple, dessinée avec un gros cercle rouge en attendant de coder la simulation physique.

Une fois que ça marche et que c'est bien testé, ajoutez une nouvelle chose, par exemple : l'écran d'accueil.

etc

Allez-y avec une série de petites étapes, et vous verrez qu'un jeu relativement complexe n'est en réalité rien d'autre qu'un tas de petites étapes relativement simples.

Ne passez pas à la prochaine étape avant d'avoir bien testé celle d'avant.

3. Pensez en termes de Produit minimum viable

Gardez toujours à l'esprit la chose suivante : mieux vaut un jeu qui a moins de fonctionnalités, mais qui marche bien, plutôt qu'un jeu qui essaie de tout faire mais qui est brisé de tout bord, tout côté...

Truc de pro: Cette philosophie se vaut également lors d'entrevues techniques pour des emplois en informatique...

Enregistrement des équipes

➢ Le travail est à faire en **équipes de deux**.

Remplissez le formulaire ici pour enregistrer votre équipe :

https://forms.office.com/r/5V7gQgWFd0

Une seule remise par équipe est suffisante.

Remise

Vous devez remettre sur Léa votre projet IntelliJ (avec la configuration Gradle, le code, les ressources, etc.) **dans un fichier .zip**

La date de remise est spécifiée sur Léa.

Barème

- 60% : Fonctionnalités demandées implantées correctement
  - Jeu fonctionnel
    - (5%) Physique/contrôle/affichage du Squelette
    - (15%) Différents Monstres
    - (10%) Logique de jeu générale
      (gestion du score, des vies, du changements de niveaux, partie perdue)
    - (5%) Magie : logique de base, collisions
    - (5%) Magie : animation avec les particules chargées
  - (5%) Écran d'accueil
  - (5%) Écran "Infos"
  - (10%) **Pas d'erreurs!**
- 40% : Qualité du code
  - (10%) Structure MVC telle que vue en classe
  - (10%) Utilisation judicieuse de l'héritage
  - (20%) Qualité générale du code
    - Code bien commenté lorsque nécessaire
    - Respect des conventions : minusculeCamelCase pour les variables/méthodes, MajusculeCamelCase pour les noms de classes, noms de variables/méthodes clairs, respect de l'indentation, etc
    - Bon découpage en méthodes
    - Encapsulation : attributs `private` (ou `protected`), avec getters/setters au besoin
    - **Pas de variables globales!**
    - **Pas de copier-coller de code!**

Note sur le plagiat

Le travail est à faire **en équipes de 2**. Ne *partagez pas de code avec une autre équipe que la vôtre*, même pas "juste pour aider un ami", ça constituerait un **plagiat**, et **tous les membres des équipes concernées auraient la note de *zéro***.