Travail Pratique 2 - High Sea Tower

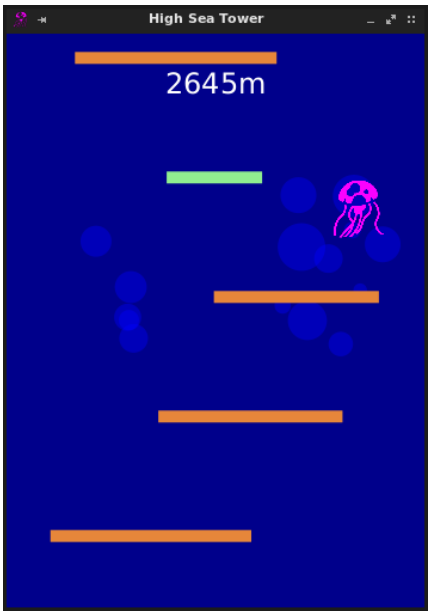
IFT1025 - Programmation 2

Concepts appliqués – Programmation orientée objet, interfaces graphiques, animation, développement d'application complète

---

1 Contexte

Le second TP consiste à programmer un jeu en interface graphique avec la librairie *JavaFX*.



1.1 Description du jeu

Vous incarnez une méduse qui tente de remonter le plus haut possible dans l'océan en sautant de plateforme en plateforme.

L'écran monte graduellement automatiquement, le but est de ne pas tomber au fond de l'océan.

Il n'y a pas d'objectif autre que de monter le plus haut possible, jusqu'à ce que l'océan vous absorbe.

La méduse peut se déplacer de gauche à droite avec les flèches du clavier (gauche/droite) et peut sauter avec la barre espace ou la flèche du haut.

1.2 Plateformes

Les plateformes sont des rectangles de différentes couleurs, tous espacés de $100px$ verticalement. Leur hauteur est toujours de $10px$, leur largeur est choisie aléatoirement entre $80px$ et $175px$. Leur position horizontale (en x) est choisie au hasard de façon à rester dans les bornes de l'écran.

Pour simplifier les collisions, la méduse est affichée avec des images mais est en réalité modélisée par un carré de taille $50px$ par $50px$.

Il y a quatre types de plateformes. Les plateformes réagissent différemment à la collision avec la méduse. Chaque nouvelle plateforme ajoutée au niveau est choisie aléatoirement parmi les types possibles avec une probabilité différente.

1.2.1 Plateforme simple (probabilité de 65%, orange)

La plateforme simple est une plateforme qu'on peut traverser depuis le bas mais qui sert de plancher lorsqu'on tombe dessus.

Utilisez la couleur `Color.rgb(230, 134, 58)` pour afficher ces plateformes.

1.2.2 Plateforme rebondissante (probabilité de 20%, vert pâle)

La plateforme rebondissante a pour effet de faire faire un rebond à la méduse lorsqu'elle tombe dessus.

Lorsque la méduse entre en collision avec la plateforme (donc en tombant dessus depuis le haut), sa vitesse verticale v_y est inversée comme dans tous les rebonds, mais elle est également amplifiée par un facteur de $\times 1.5$.

Pour assurer un rebond minimalement intéressant, la vitesse après rebond est forcée à être *au moins* de 100px/s vers le haut.

Utilisez la couleur `Color.LIGHTGREEN` pour afficher ces plateformes.

1.2.3 Plateforme accélérante (probabilité de 10%, jaune foncé)

Lorsque la méduse se pose sur une plateforme accélérante, la vitesse à laquelle l'écran monte automatiquement est multipliée par 3. La méduse a donc intérêt à rester posée dessus le moins longtemps possible pour éviter de se retrouver dans le fond de l'eau.

Utilisez la couleur `Color.rgb(230, 221, 58)` pour afficher ces plateformes.

1.2.4 Plateforme solide (probabilité de 5%, rouge)

La plateforme solide *ne peut pas être traversée depuis le bas*. Il s'agit donc d'un rectangle solide avec des collisions normales.

Puisque cette plateforme ne peut pas être traversée depuis le bas, avoir plusieurs plateformes de ce type de suite rendrait le jeu trop difficile. Vous devrez donc empêcher que deux plateformes de suite soient des plateformes solides.

Dans le cas où une plateforme solide est ajoutée, la plateforme suivante doit être de l'un des trois autres types. Les probabilités à utiliser pour choisir le type de plateforme dans ce cas ne sont pas définies, à vous de choisir ce que vous faites.

Utilisez la couleur `Color.rgb(184, 15, 36)` pour afficher ces plateformes.

1.3 Méduse

La méduse est modélisée par un carré de taille $50px$ par $50px$ et est animée avec les images `jellyfish1.png` à `jellyfish6.png`, avec un framerate de 8 images par seconde.

Ces images montrent la méduse qui regarde vers la droite. Si la méduse regarde à gauche, on devrait plutôt utiliser les images `jellyfish1g.png` à `jellyfish6g.png` dans l'animation.

De base, l'écran monte automatiquement à une vitesse initiale de $50px/s$ et accélère graduellement à une vitesse de $2px/s^2$. Plus le jeu avance, plus l'écran monte vite de lui-même, ce qui ajoute de la difficulté à mesure qu'on monte.

Dès que la méduse dépasse 75% de la hauteur de l'écran (à partir du bas) en sautant, cela fait monter l'écran de façon à ce que la coordonnée la plus haute de la méduse ne soit jamais affichée plus haut que ça.

Si la coordonnée la plus haute de la méduse se retrouve plus bas que le bas de l'écran, la méduse est absorbée par l'océan, la partie est perdue et le jeu est remis à zéro.

La "gravité" du jeu est de $1200px/s^2$ vers le bas. Chaque saut avec la barre espace/la flèche du haut donne instantanément une vitesse de $600px/s$ vers le haut.

Lorsqu'on appuie sur les flèches gauche et droite, cela a pour effet de donner instantanément à la méduse une *accélération en x* de $1200px/s^2$ vers la gauche ou vers la droite (respectivement).

La méduse ne peut pas sortir de l'écran par les côtés : lorsque la méduse touche la gauche ou la droite de l'écran, elle rebondit dans l'autre direction.

1.4 Début de la partie

Lorsque la partie commence, la méduse est sur le sol, tout en bas de l'écran. On attend alors que l'utilisateur appuie sur une touche (gauche, droite ou saut) pour commencer à faire avancer l'écran automatiquement. Cela laisse le temps aux utilisateurs de se préparer avant que la partie ne commence.

Lorsque la méduse tombe plus bas que l'écran, la partie est perdue, une nouvelle partie commence, et on attend à nouveau que l'utilisateur appuie sur une touche pour recommencer.

1.5 Bulles

Pour donner un peu de vie à l'océan, l'arrière-plan va afficher des bulles qui remontent à la surface de temps en temps.

Des bulles apparaissent dans l'arrière-plan à toutes les 3 secondes :

- Les bulles sont des cercles d'un rayon aléatoire entre 10 et 40px
- Elles sont affichées en bleu avec une opacité de 40%, ce qui peut être obtenu avec : `Color.rgb(0, 0, 255, 0.4)`
- Chaque bulle a une vitesse aléatoire entre 350 et 450 px/s vers le haut
- Les bulles font partie du décor, la méduse ne peut pas entrer en collision avec elles
- Pour donner un effet plus réaliste, les bulles sont générées en 3 petits groupes de 5 bulles :
  - Trois coordonnées $base_x$ sont générées aléatoirement entre 0 et la largeur de l'écran
  - Pour chacune de ces coordonnées, cinq bulles sont générées avec comme position x initiale la valeur $base_x \pm 20$
  - La position y initiale des bulles doit être à l'extérieur de l'écran (tout en bas). Les bulles montent jusqu'à ce qu'elles dépassent le haut de l'écran

2 Interface

- La fenêtre doit avoir une largeur de `350px` et une hauteur de `480px`
- Le canvas (la fenêtre de jeu) dans l'application doit avoir la même taille et occuper tout l'espace
- 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 porter le titre "High Sea Tower"
- La fenêtre doit avoir une des images de la méduse en guise d'icône dans la barre de tâches
- Le jeu doit se fermer lorsqu'on appuie sur la touche `Escape`. Utilisez `Platform.exit();` pour mettre fin au programme.

En tout temps, on devrait voir le score actuel en nombre de "mètres" montés. Ce nombre commence à zéro et correspond en réalité au nombre de pixels dont l'écran a monté depuis le début de la partie.

2.1 Mode `Debug`

Pour faciliter les tests, vous devez inclure un mode `debug` à votre jeu, activable/désactivable en appuyant sur la touche `t` à n'importe quel moment.

Le mode `debug` permet de tester les plateformes et l'avancement dans le jeu : lorsqu'il est activé, la fenêtre arrête de monter automatiquement et monte seulement lorsque la méduse dépasse les 75% de hauteur d'elle-même.

Ce mode aide à valider les collisions :

- Un carré rouge est dessiné dans la boîte englobante de la méduse. Cela correspond à toute la zone 50 par 50 qui compte dans une détection de collision entre la méduse et une plateforme
- Lorsqu'une plateforme est en collision avec la méduse, on change la couleur de cette plateforme et on la dessine plutôt en jaune (`Color.YELLOW`)

De plus, des informations doivent être affichées en haut à gauche de l'écran. On doit afficher :

- La position (x, y) de la méduse *dans les coordonnées du niveau*. Le point $(0, 0)$ en termes de coordonnées dans le niveau **se trouve tout en bas à gauche, sur la ligne où la méduse commence**. Vous devrez donc gérer à la fois :
  - Les coordonnées dans le canvas JavaFX, toujours entre $(0, 0)$ et $(350, 480)$
  - Les coordonnées dans le niveau lui-même, avec un x entre 0 et 350 et un $y > 0$, qui pourrait théoriquement monter à l'infini à mesure qu'on avance dans le jeu
- Affichez juste en dessous de ça la vitesse de la méduse
- Son accélération
- Si oui ou non la méduse touche présentement le sol (soit parce qu'elle est sur une plateforme, soit parce qu'elle est sur la ligne où le niveau commence)

3 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 à ce qui constitue votre Modèle pour ce jeu, à comment définir une vue correctement et à comment faire usage du Contrôleur pour isoler complètement les classes de la Vue des classes du Modèle.

La classe principale du programme doit se nommer `HighSeaTower` et être dans le package par défaut (aka, pas de ligne `package ...` au début du fichier).

Vous pouvez créer des packages pour vos autres classes si vous jugez que c'est nécessaire, mais gardez en tête que le programme ne devrait pas être particulièrement gros et n'en a pas absolument besoin.

4 Note sur la mémoire utilisée

Si votre code crée des instances d'objets pour représenter des éléments qui peuvent sortir de l'écran (ex.: bulles, plateformes, ...), 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 une référence à une plateforme qui ne sera plus jamais affichée de tout le reste du jeu.

5 Éléments fournis

Aucun code n'est fourni, mais vous pouvez vous inspirer des exemples de code donnés dans le chapitre sur les GUIs et (surtout) dans le chapitre sur l'infographie 2D.

Les images nécessaires à l'animation de la méduse sont fournis sur StudiUM et sont basées sur une image par le graphiste Lorc (image originale ici : https://game-icons.net/1x1/lorc/jellyfish.html).

6 Bonus

6.1 Améliorer le jeu (jusqu'à 5%)

Ajoutez des fonctionnalités de votre choix au programme pour en faire un jeu plus intéressant. Soyez créatifs, rendez le jeu amusant, amusez-vous!

Le pourcent bonus sera donné sur l'originalité (autrement dit, si vous faites le strict minimum simplement pour avoir votre point bonus, ça ne sera pas compté) et sur la complexité de ce que vous avez fait. Quelque chose de relativement simple à ajouter ne vaudra pas 5%.

Si vous faites ce bonus, ajoutez un fichier `BONUS.txt` à votre remise contenant la liste de ce que vous avez ajouté et expliquant ce que vous avez dû faire pour y arriver.

6.2 Version mobile (10%)

Portez le jeu sur Android. Faites vos recherches vous-même, si votre découpage MVC est bien conçu, vous ne devriez pas avoir trop de misère à adapter le code.

Vous *devez* utiliser exactement les mêmes classes dans votre projet JavaFX et dans votre projet Android.

Notez que vous ne pouvez *pas* utiliser un outil qui compile du JavaFX directement sur Android, vous devez utiliser la librairie standard d'Android pour définir l'affichage.

Si vous choisissez de faire ce bonus, envoyer un courriel à `nicolas.hurtubise@umontreal.ca` pour le faire évaluer.

7 Barème

- **50%** ~ Exécution (programme conforme à ce qui est demandé)
- **30%** ~ Découpage en classes et utilisation judicieuse de l'héritage
- **10%** ~ Découpage du programme en suivant l'architecture MVC
- **10%** ~ Qualité du code
  - Code bien commenté (*JavaDoc*, mais pas besoin d'en mettre pour les accesseurs/mutateurs)
  - Code bien indenté, bon découpage en fonctions, encapsulation, noms de variables bien choisis, performance du code raisonnable, camelCase, pas trop compact, pas trop espacé... bref, du code clair et lisible en suivant les principes que vous connaissez
  - Limitez vos lignes de code à 120 caractères de large

8 Indications supplémentaires

- La date de remise est le *14 avril 2020 à 23h59*. Un travail remis le lendemain avant 23h59 aura une pénalité de *-33%*. Un travail remis passé le lendemain se verra attribué la note de zéro.
- Vous **devez** faire le travail par groupes de 2 personnes. Indiquez vos noms clairement dans les commentaires au début de votre code, dans le fichier principal (`HighSeaTower.java`).
- Une seule personne par équipe doit remettre le travail sur StudiUM
- Un travail fait seul engendrera une pénalité. Les équipes de plus de deux ne sont pas acceptées.
- Remettez tous les fichiers nécessaires à l'exécution de votre code dans une archive `.zip` (pas de `.rar` autorisés)
- De plus :
  - La performance de votre code doit être raisonnable
  - Chaque fonction devrait être documentée en suivant le standard `JavaDoc`
  - Il devrait y avoir des lignes blanches pour que le code ne soit pas trop dense (utilisez votre bon sens pour arriver à un code facile à lire)
  - Les identificateurs doivent être bien choisis pour être compréhensibles (évitez les noms à une lettre, à l'exception de i, j, ... pour les variables d'itérations des boucles for)
  - Vous devez respecter le standard de code pour ce projet, soit les noms de variables et de méthodes en camelCase