

# IFT1025 - Travail Pratique 2

## Colors Witch



*Concepts appliqués : Programmation orientée objet, interfaces graphiques, hiérarchies de classes, développement d'application de taille plus importante*

### Contexte

Pour ce deuxième travail pratique, vous devrez améliorer un jeu inspiré du jeu mobile addictif *Color Switch*.

À l'heure actuelle, il n'est plus possible de télécharger le jeu, mais vous pouvez avoir une idée de ce dont ça avait l'air en regardant cette vidéo : <https://www.youtube.com/watch?v=ivtzSFdH2k8>

### *Colors Witch*

Le jeu que vous aurez à modifier est *Colors Witch*, un jeu dans lequel vous incarnez une sorcière  à la recherche de champignons .

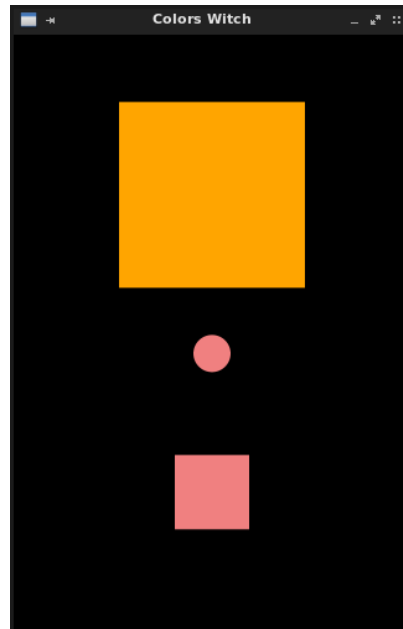



Figure 1: Capture d'écran du jeu

Pour y arriver, la sorcière s'est matérialisée sous forme de boule d'énergie colorée et doit franchir des obstacles. Pour une raison magique quelconque, la sorcière peut seulement passer à travers les obstacles lorsqu'elle est de la même couleur que ceux-ci.

De temps en temps, la sorcière trouve des potions magiques  qui lui permettent de changer de couleur.

## Structure du code

La logique de base du jeu vous est fournie, vous aurez la tâche d'ajouter des obstacles, niveaux, items et d'apporter diverses améliorations.

Afin de faciliter la maintenance du jeu, les classes sont séparées suivant une architecture *MVC*.

## Modèle

La hiérarchie de classes est représentée dans la *Figure 2*. Lisez les commentaires dans le code pour avoir plus d'informations sur les classes, méthodes et attributs déjà présents.

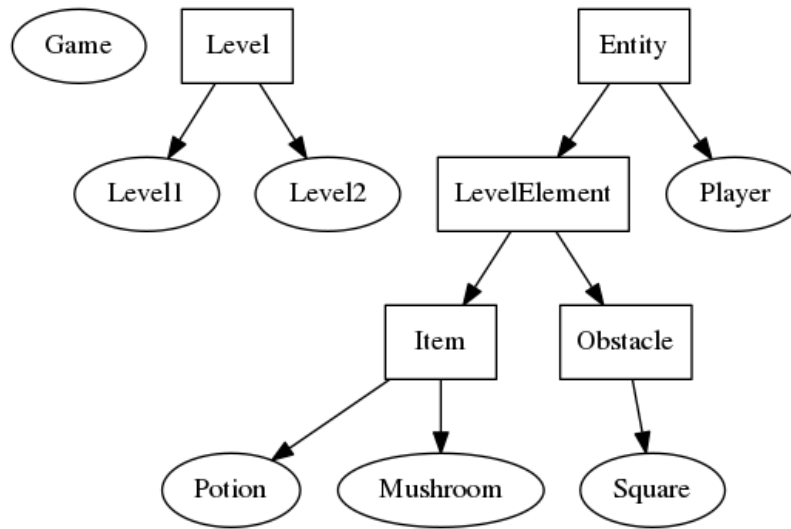


Figure 2: Hiérarchie d'objets constituant le modèle du jeu : carrés = classes abstraites, cercles = classes concrètes

La classe abstraite de base qui représente les entités dans les niveaux est la classe **Entity**.

Une **Entity** possède une position  $(x, y)$  définie par rapport au niveau ( $y=0$  correspond au début du niveau, tout en bas) et définit une fonction `tick(double dt)` qui est appelée à chaque frame du jeu.

## Vue

La classe principale, dans `ColorsWitch.java`, définit l'interface graphique, les événements et l'`AnimationTimer` qui permet à la logique du jeu de s'exécuter à chaque frame.

Afin de séparer le modèle de l’affichage graphique, chaque **Entity** est associée à un **Render**, une classe seulement chargée de faire le rendu d’une entité sur le canvas.

Les classes **PlayerRenderer**, **SquareRenderer** et **ImageRenderer** permettent de faire le rendu de la sorcière, d’un obstacle **Square** et d’une **Entity** quelconque utilisant une image.

### Coordonnées de l’écran vs Coordonnées dans le niveau

Pour faciliter certains aspects du développement, les coordonnées des entités sont données par rapport au début du niveau. Puisque les niveaux commencent en bas de la fenêtre et s’étendent vers le haut, les coordonnées du niveau sont données en utilisant comme référence le coin en bas à gauche.

JavaFX utilise cependant toujours une référence qui part d’en haut à droite, faites attention de ne pas mêler les deux lors de l’affichage.

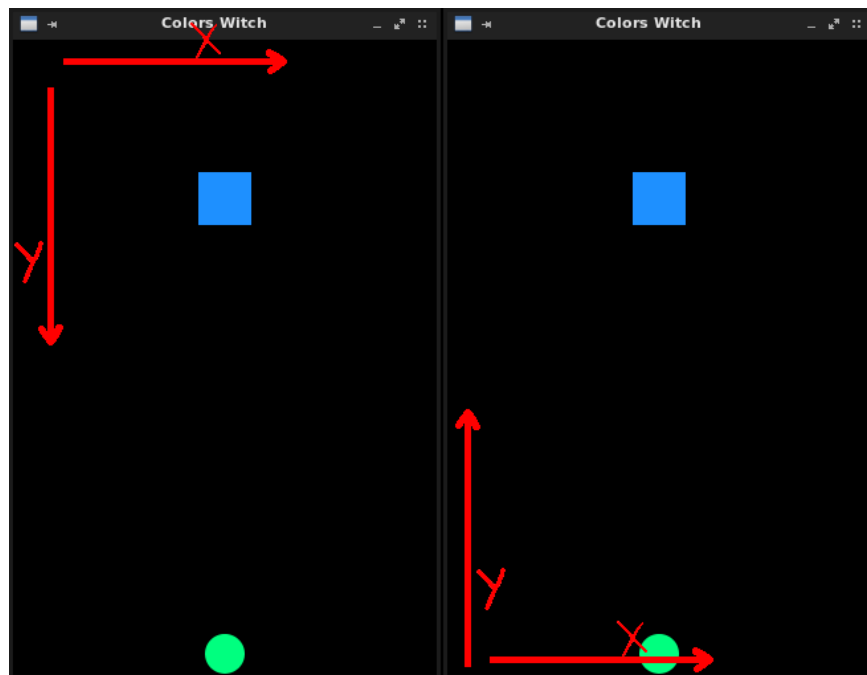


Figure 3: Système de coordonnées de l’écran (à gauche) vs Système de coordonnées du niveau (à droite)

### Contrôleur

Le contrôleur (voir `Controller.java`) se charge simplement de faire le pont entre la vue et le modèle.

Vous pouvez lui ajouter du code au besoin.

## Choses à faire

Vous avez une petite liste de tâches pour améliorer le jeu, avec la pondération de points associée à chacune :

### 1. (25%) Créer trois nouveaux types d'Obstacles

La classe `Obstacle` est une sous-classe d'`Entity` qui représente un objet du niveau. Lorsque la sorcière entre en collision avec un `Obstacle` qui n'a pas la même couleur qu'elle, le niveau est perdu.

Par exemple, la classe `Square` représente un obstacle carré qui change de couleur avec le temps.

Vous devrez créer trois obstacles :

1. Un obstacle nommé `VerticalBar`. Il s'agit simplement d'une barre verticale dont la couleur ne change pas et qui se déplace de gauche à droite en rebondissant sur les murs du niveau
2. Un obstacle nommé `GrowingCircle`. Il s'agit d'un cercle qui grossit et rétrécit avec le temps et dont la couleur change avec le temps (à la même vitesse que le `Square` donné en exemple)
3. Un obstacle nommé `RotatingCircle`. Il s'agit d'un cercle de taille fixe qui tourne autour d'un point central et dont la couleur change avec le temps (à la même vitesse que le `Square` donné en exemple)

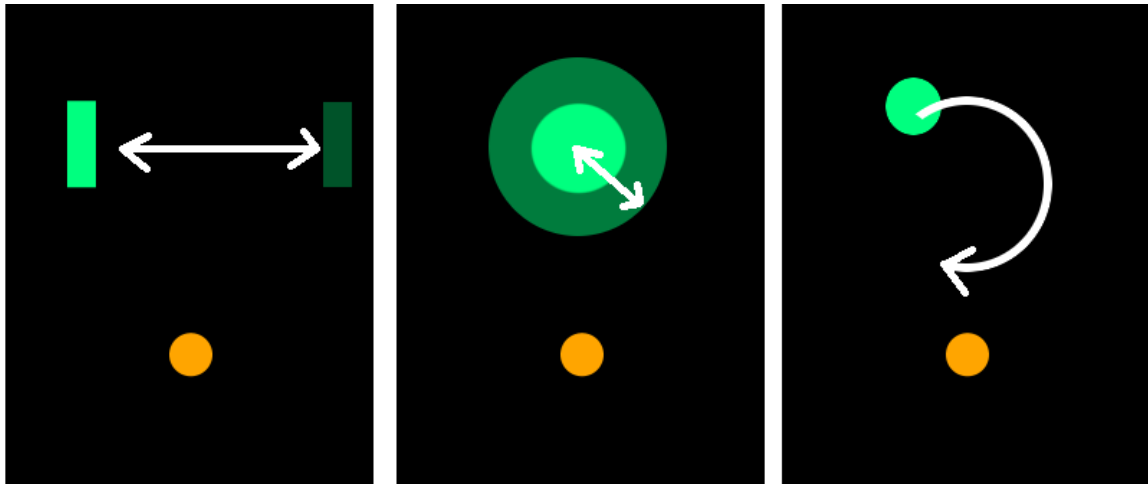


Figure 4: Obstacles (de gauche à droite) : `VerticalBar`, `GrowingCircle`, `RotatingCircle`


Notez : les fonctions `Math.sin` et `Math.cos` vous seront utiles pour l'animation du `RotatingCircle`

### 2. (15%) Créer un nouvel Item

La classe `Item` est une sous-classe d'`Entity` qui représente un objet quelconque du niveau. Lorsque la sorcière entre en collision avec un `Item`, la méthode `handleCollision(Player player, Game game)` est appelée sur l'item. N'importe quoi peut alors se produire.

Un exemple d'item est la `Potion`, qui change la couleur de la sorcière lorsqu'il y a collision.

Vous devrez créer une sous-classe `Shield` de la classe `Item`, qui représente un item qui rend la sorcière invincible pendant 3 secondes.

- L’affichage d’un `Shield` devrait utiliser l’image `shield.png`  dans le dossier `src/`
- La détection d’intersection du `Shield` avec le `Player` devrait tenir compte du fait qu’il s’agit d’un cercle d’un rayon de 32 pixels

### 3. (20%) Créer 4 niveaux

Utilisez les nouveaux items et obstacles que vous avez créés pour créer 4 nouveaux niveaux du jeu.

Vous pouvez remplacer les classes `Level1` et `Level2` par vos propres niveaux, ces classes sont seulement là à titre d’exemples.

### 4. (10%) Corriger les collisions entre le cercle (`Player`) et le reste

Dans la base de code fournie, toutes les détections de collisions se basent uniquement sur le point central du cercle qui représente la sorcière, ce qui n’est pas correct. Vous devrez corriger ça.

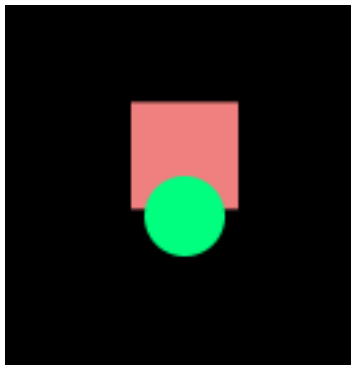


Figure 5: La collision entre le `Player` et les autres `Entity` n’est pas détectée tant que le centre ne touche pas l’objet... À corriger

*Notez :* vous pouvez assumer que les potions et les champignons sont des carrés qui contiennent une image. Dès que la sorcière touche le carré, il y a collision. Pas besoin de faire une collision précise au pixel près pour ces entités.

### 5. (15%) Animer le champignon

Remplacez le `Renderer` des champignons (classe `Mushroom`) par une nouvelle classe qui fait plutôt une animation avec les frames `mushroom_animation1.png` à `mushroom_animation26.png`.

Créez une classe `AnimationRenderer` qui prend en paramètre de son constructeur :

- `String prefix` : le chemin de base de l’image (sans le numéro de frame)
- `int number` : le nombre de frames totales dans la série d’images
- `double framerate` : le nombre de fois que l’image doit être mise à jour à chaque seconde

- **Entity entity** : l'entité de jeu associée au rendu

Inspirez-vous de la classe **ImageRender** et des notes de cours et exemples donnés sur l'animation basée sur des images.

## 6. (15%) Afficher un message lorsqu'on gagne ou perd dans un niveau

Présentement, lorsqu'un niveau est complété, on passe directement au niveau suivant, sans félicitations, sans indications de quoi que ce soit.

De la même façon, si on perd dans un niveau, on devrait avoir une indication textuelle.

Trouvez une façon d'afficher un message lorsqu'on gagne ou on perd, soit en ajoutant un élément **Text** dans la scène, soit en dessinant du texte sur le canvas déjà présent avec la méthode **drawText** quelque part.

## Bonus

## 7. (5%) Animer la mort de la sorcière

Lorsque la sorcière touche un obstacle de la mauvaise couleur, on voudrait qu'elle "explose" en 100 petites balles qui partent dans tous les sens et qui rebondissent sur les murs, un peu comme dans le jeu original (voir <https://www.youtube.com/watch?v=ivtzSFdH2k8>).

## 8. (5%) Créer plus de variété

Ajoutez :

- 2 autres types d'obstacles animés, dont les animations sont différentes de celles des autres obstacles
- 2 autres types d'items, un qui aide la sorcière dans sa quête, l'autre qui lui nuit (par exemple, en modifiant sa vitesse, en ajoutant des obstacles au niveau, ou quoi que ce soit d'autre...)
- 6 niveaux, pour un total de 10 niveaux

Inspirez-vous d'images tirées du site <http://game-icons.net> au besoin pour les items.

Soyez créatifs, vous n'aurez pas tous vos points si vous ne faites que des choses simples et faciles pour finir le plus vite possible... Le but est de rendre le jeu réellement intéressant !

## 9. (5%) Ajouter un menu qui permet de choisir le niveau dans lequel on joue

Dans le code fourni, lorsqu'on démarre le jeu, on commence directement dans le niveau 1. On devrait plutôt pouvoir choisir le niveau qui nous intéresse.

Ajoutez un menu qui permet de choisir le niveau dans lequel on joue.

De plus, pendant un niveau, on devrait pouvoir appuyer sur **Escape** pour quitter le niveau et revenir au menu.

## Barème

- **65%** ~ Le jeu marche comme demandé
  - Respect de la spécification : toutes les fonctionnalités demandées sont présentes
  - Pas de bugs, ni problèmes d’affichage graphique, ni erreurs de logique
- **35%** ~ Qualité du code
  - Découpage des classes en conservant l’architecture *Modèle-Vue-Contrôleur*
  - Code bien commenté, bien indenté
  - Performance du code raisonnable

## Indications supplémentaires

- La date de remise est le *27 avril 2018 à 23h55*. Il y a une pénalité de 25% pour chaque jour de retard.
- 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 (**ColorsWitch.java**).
- Une seule personne par équipe doit remettre le travail sur StudiUM, l’autre doit remettre un fichier nommé **equipe.txt** contenant uniquement le code d’identification de l’autre personne (p1234..., soit ce que vous utilisez pour vous connecter sur StudiUM)
- Un travail fait seul engendrera une pénalité. Les équipes de plus de deux ne sont pas acceptées.
- Basez-vous sur le code fourni et remettez tous les fichiers
- 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