LMI 1 - Semestre 2 Mars-Avril 2016

Université d'Artois
Faculté des Sciences Jean Perrin

ALGO2 – Algorithmique et Programmation 2

Projet : Roundballs

Le projet

Il s'agit de réaliser un jeu de la famille *Candy Crush* : on permute des objets dans une grille, lorsqu'au moins trois objets identiques sont alignés ils disparaissent (et font augmenter un score), les objets des lignes supérieures tombent dans les trous créés et de nouveaux objets apparaissent en haut de la grille. Le jeu que nous allons réécrire est inspiré du jeu libre *Roundballs* auquel vous pouvez jouer à l'adresse `http://framagames.org/roundball/roundball.html` (jeu classique).

Votre travail

Pour cela, vous disposez d'un module `gereDessins.py` qui propose une classe `GereDessins` qui prend en charge le dessin des balles et les animations. Votre travail sera :

— de créer le modèle pour ce jeu : une classe `Roundball` qui permet de jouer en mode texte ;
— de créer une interface graphique simple qui permettra d'afficher sous forme de disques colorés les éléments du jeu ;
— de créer les contrôleurs qui feront le lien, au niveau de l'interface graphique, entre le modèle et la classe qui gère les dessins et les animations.

Vous serez guidés, avec un travail à rendre régulièrement, jusqu'à la réalisation finale du projet qui s'étale sur 5 semaines. **Ce projet est à faire seul ou en binôme.** À cet effet, vous devez vous inscrire et donner le nom de votre binôme dans le questionnaire prévu à cet effet sur Moodle pour le **23 mars 2016 à 18h** dernier délai. J'insiste sur le fait que chaque étudiant doit faire s'inscrire et déclarer son binôme : Kévin doit dire qu'il est en binôme avec Anissa et Anissa doit dire qu'elle est en binôme avec Kévin.

Vous serez évalués en fonction de la qualité de votre conception, de votre implémentation, le bon fonctionnement de l'application et la documentation fournie. Vous veillerez en particulier à spécifier et à commenter vos classes, méthodes et fonctions.

Vous devrez fournir également un document descriptif des différentes fonctions utilisées. Ce document sera au format `pdf` et sera également déposé sur Moodle.

Attention : à la fin de chaque étape, vous devrez déposer à l'emplacement réservé (”Dépôt Étape 1”, ”Dépôt Étape 2”, etc.) sur Moodle l'état d'avancement de votre application. Vous déposerez également un document texte qui décrira votre application. Dans un binôme, ce sera toujours la même personne qui dépose les fichiers (pour éviter les erreurs).

Calendrier des dépôts :

— **Étape 1** date limite de dépôt le mercredi 30 mars à 22h
— **Étape 2** date limite de dépôt le mercredi 13 avril à 22h
— **Étape 3** date limite de dépôt le vendredi 22 avril à 22h
— **Étape 4** date limite de dépôt le vendredi 29 avril à 22h

Chaque étape sera notée. Si toutes les fonctionnalités demandées sont rendues à l'heure avec un fonctionnement correct, un bonus de **1 point** par étape sera accordé. Si le travail à faire pour une étape est finalisé seulement pendant le délai donné pour l'étape suivante, aucun bonus ni malus ne sera appliqué.

Au-delà d'une étape de retard, un malus de 1 point sera appliqué. Pour que votre projet soit évalué, vous devez déposé le travail que vous avez effectué à chaque étape. Rien ne sera accepté au-delà du vendredi 29 avril à 22h.

Un contrôle individuel sera effectué lors de la dernière séance de TP.

Le projet Roundball

Le but de ce projet est de réaliser le jeu *Roundball*, dont la version finale pourrait ressembler à la figure 1 :

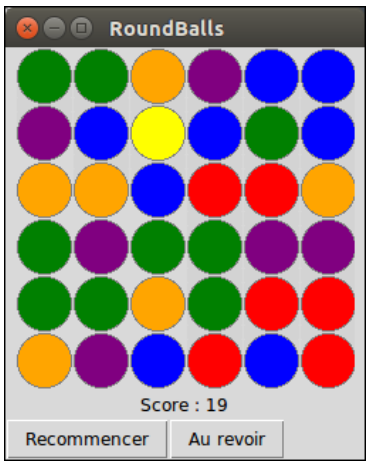

FIGURE 1 – Roundball : interface graphique finale du projet

Ce projet a été découpé en quatre étapes, afin de vous accompagner dans sa réalisation. Chaque semaine, vous rendrez la nouvelle version de votre projet incluant les fonctionnalités demandées à chaque étape. Vous devez donc respecter l'ordre de réalisation pour chaque semaine. Une grande part de votre travail sera de respecter la bonne décomposition du projet : la séparation du moteur du jeu (le modèle, qui fait les calculs) de la partie qui gère l'interface graphique, . . .Vous serez guidés dans cette décomposition, qui vous permettra d'avancer facilement d'étape en étape.

Planning du projet

Étape 1 - Le modèle (1)

La première semaine sera consacrée à réaliser une version en mode texte du jeu. Pour cela, basiquement, on retrouve la structure de grille sur laquelle nous avons travaillé (pour le taquin, pour le démineur, . . .). Vous allez donc pouvoir récupérer du code déjà écrit, pour avoir une classe qui manipule une matrice d'entiers, qui s'affiche, . . .Je vous rappelle que vous disposez d'un fichier `demineur.py` qui contient toutes les méthodes d'affichage nécessaires. Dans le cas de notre jeu, on va pouvoir se contenter d'une seule classe `Roundball` qui va gérer une matrice d'entiers. Les valeurs de la matrice seront comprises entre -1 et *nombre de couleurs - 1*. Au départ, la matrice est initialisée avec des valeurs ≥ 0 (les valeurs positives symbolisent les couleurs, `-1` représente une case vide). La classe `Roundball` contient également un attribut `__score` pour donner le nombre de points gagnés par le joueur.

Les principales nouvelles actions à avoir sont :

— détecter les alignements (horizontalement ou verticalement) de trois valeurs identiques - c'est la méthode `recherche_trios(self)` qui retourne la liste des balles à supprimer (sans les supprimer)

— faire les changements de valeurs dans la grille qui en découlent : supprimer les balles qui doivent l'être, faire tomber dans les colonnes celles qui doivent glisser, créer de nouvelles balles aux endroits libres - c'est la méthode `change_valeurs(self,balles_a_effacer)` qui modifie la matrice.

Voici un exemple d'interaction (des commentaires ont été ajoutés) :

```
anne@gavotte:~/enseign/algo2/15-16/roundballs$ python3
Python 3.4.3+ (default, Oct 14 2015, 16:03:50)
[GCC 5.2.1 20151010] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from modeleRoundball import *
>>> r = Roundball()  #### par défaut, 6 lignes, 6 colonnes et 8 couleurs
>>> r.affiche_plateau()
   0 1 2 3 4 5
  +-+-+-+-+-+-+
0 |2|0|0|0|4|5|   #### on remarque ici que par hasard
  +-+-+-+-+-+-+   #### trois zéros ont été alignés en
1 |5|3|5|1|0|1|   #### (0,1), (0,2), (0,3)
  +-+-+-+-+-+-+
2 |2|1|4|3|4|4|
  +-+-+-+-+-+-+
3 |5|1|3|2|2|0|
  +-+-+-+-+-+-+
4 |2|3|4|2|1|2|
  +-+-+-+-+-+-+
5 |4|1|0|1|4|0|
  +-+-+-+-+-+-+
>>> l = r.recherche_trios()
>>> l
[(0, 1), (0, 2), (0, 3)]  #### recherche_trios les a trouvés
>>> r.change_valeurs(l)   #### on fait les mouvements de plateau
>>> r.affiche_plateau()
   0 1 2 3 4 5
  +-+-+-+-+-+-+
0 |2|5|3|3|4|5|   #### trois nouvelles valeurs sont arrivées
  +-+-+-+-+-+-+
1 |5|3|5|1|0|1|
  +-+-+-+-+-+-+
2 |2|1|4|3|4|4|
  +-+-+-+-+-+-+
3 |5|1|3|2|2|0|
  +-+-+-+-+-+-+
4 |2|3|4|2|1|2|
  +-+-+-+-+-+-+
5 |4|1|0|1|4|0|
  +-+-+-+-+-+-+
>>> l = r.permute((0,1),(1,1))  #### on permute les valeurs
                                #### des cases (0,1) et (1,1)
                                #### et on fait un appel à recherche_trios()
>>> l
```

```
[(0, 1), (0, 2), (0, 3), (1, 0), (1, 1), (1, 2)]
                            #### il y a trois 3 alignés en ligne 0
                            #### et trois 5 alignés en ligne 1
                            #### qui devront être effacés
#### remarque : permute et recherche_trios ne modifient pas la matrice !
>>> r.change_valeurs(l)     #### on applique les modifications
>>> r.affiche_plateau()     #### - effacer les cases à effacer
   0 1 2 3 4 5              #### - faire tomber les valeurs au-dessus
  +-+-+-+-+-+-+             #### des trous crées
0 |5|0|4|2|4|5|             #### - tirer aléatoirement de nouvelles
  +-+-+-+-+-+-+             #### valeurs aux endroits restés vides
1 |2|5|3|1|0|1|
  +-+-+-+-+-+-+
2 |2|1|4|3|4|4|
  +-+-+-+-+-+-+
3 |5|1|3|2|2|0|
  +-+-+-+-+-+-+
4 |2|3|4|2|1|2|
  +-+-+-+-+-+-+
5 |4|1|0|1|4|0|
  +-+-+-+-+-+-+
>>> r.score()
9
```

Vous devez donc implémenter la classe `Roundball` qui contient :

— un constructeur, qui prend en paramètre un nombre de lignes, un nombre de colonnes, et un nombre de couleurs. Par défaut, le nombre de lignes, de colonnes et de couleurs sont fixés à 6. Le nombre de couleurs **ne peut pas** dépasser 8. La matrice du jeu est initialisée avec des valeurs aléatoirement tirées entre `0` et (*nombre de couleurs* - 1). La classe `Roundball` définit également un attribut `__score` initialisé à zéro.

— une méthode pour afficher la matrice du jeu

— une méthode `lig(self)` qui retourne le nombre de lignes

— une méthode `col(self)` qui retourne le nombre de colonnes

— une méthode `valeur(self,i,j)` qui retourne la valeur d'un élément donné par ces coordonnées

— une méthode `recherche_trios(self)` qui retourne la liste des balles à supprimer (mais cette méthode ne supprime rien !).

Le principe est simple : la méthode parcourt d'abord chaque ligne, et sur chaque ligne, on mémorise les coordonnées de tous les éléments successifs de même valeur. Si le nombre de ces éléments est d'au moins 3, alors ces coordonnées sont mémorisées dans la liste qui sera retournée. Puis on fait la même chose en parcourant chaque colonne. Un élément de la matrice peut donc servir à deux alignements. Exemple :

```
>>> r = Roundball()
>>> r.affiche_plateau()
   0 1 2 3 4 5
  +-+-+-+-+-+-+
0 |2|0|2|7|3|5|
  +-+-+-+-+-+-+
1 |4|4|5|7|5|2|
  +-+-+-+-+-+-+
```

```
2 |4|5|0|7|7|7|
  +-+-+-+-+-+-+
3 |0|7|3|4|1|3|
  +-+-+-+-+-+-+
4 |3|4|5|5|0|2|
  +-+-+-+-+-+-+
5 |4|7|7|5|3|5|
  +-+-+-+-+-+-+
>>> l = r.recherche_trios()
[(2, 3), (2, 4), (2, 5), (0, 3), (1, 3), (2, 3)]
```

— une méthode `permute(self,coords_case1,coords_case2)` qui opère la permutation entre deux cases si elles sont voisines, lance un appel à `recherche_trios` et retourne la liste des balles à supprimer

— une méthode `tombe(self,lig,col)` qui fait tomber une balle dans la case aux coordonnées `(lig,col)` :

  — s'il y a une couleur (une valeur différente de `-1`) dans la matrice en `(lig,col)`, alors il n'y a rien à faire (la case est déjà occupée, tout va bien)

  — sinon, il faut remonter dans les lignes supérieures de la même colonne pour trouver la première case contenant une couleur pour faire *tomber* la valeur en `(lig,col)`. Si une telle case n'existe pas (toute la colonne `col` au-dessus de la ligne `lig` est vide), alors il faut choisir une nouvelle couleur aléatoirement

— une méthode `supprime_trios(self,les_elts_a_supp)` qui prend en paramètre les coordonnées des éléments à supprimer (la liste calculée par la méthode `recherche_trios`). Cette méthode commence par mettre `-1` sur les éléments à effacer. Le `__score` est augmenté de `1` pour chaque élément supprimé. Un élément qui apparaitraît deux fois dans la liste retournée par `recherche_trios` sera pris en compte deux fois dans le score. Puis elle doit calculer les mouvements glissants vers le bas et les créations de nouvelles valeurs pour remplir la matrice.
Une fois les éléments effacés, si on reprend l'exemple précédent, on pourrait trouver dans la matrice les valeurs suivantes :

```
   0 1 2  3  4  5
  +-+-+-+--+--+--+     #### une nouvelle valeur à créer en colonne 3
0 |2|0|2|-1| 3| 5|     #### une nouvelle valeur à créer en colonne 4
  +-+-+-+--+--+--+     #### une nouvelle valeur à créer en colonne 5

  +-+-+-+--+--+--+     #### 5 devra tomber (colonne 4),
1 |4|4|5|-1| 5| 2|     #### 5 devra tomber (colonne 5),
  +-+-+-+--+--+--+     #### une nouvelle valeur à créer en colonne 3

  +-+-+-+--+--+--+     #### 5 devra tomber (colonne 4),
2 |4|5|0|-1|-1|-1|     #### 2 devra tomber (colonne 5),
  +-+-+-+--+--+--+     #### une nouvelle valeur à créer en colonne 3

  +-+-+-+--+--+--+
3 |0|7|3| 4| 1| 3|     #### rien à changer sur cette ligne
  +-+-+-+--+--+--+
4 |3|4|5| 5| 0| 2|     #### rien à changer sur cette ligne
  +-+-+-+--+--+--+
5 |4|7|7| 5| 3| 5|     #### rien à changer sur cette ligne
  +-+-+-+--+--+--+
```

Il suffit donc de regarder la matrice ligne par ligne en partant du bas, et d'appeler `tombe` sur

chaque cellule de la ligne.
- une méthode `change_valeurs(self,les_elts_a_supp)` qui prend en paramètre les coordonnées des éléments à supprimer (la liste calculée par la méthode `recherche_trios`). Cette méthode permet de répéter les étapes `recherche_trios`, `supprime_trios` tant qu'un changement doit s'effectuer. Lorsqu'on appelle cette méthode, `recherche_trios` a déjà été appelée une fois (c'est pour cela qu'une liste des éléments à supprimer est donnée en paramètre).
- une méthode `re_init(self)` qui réinitialise la matrice avec de nouvelles valeurs, et effectue autant que de besoin des appels à `recherche_trios` et `supprime_trios`.

Dans cette première étape, vous aurez réalisé presque toutes les fonctions fondamentales du jeu. Vous devez pouvoir jouer à travers l'interprète Python, comme dans l'exemple ci-dessus, en faisant des appels directs sur les méthodes de la classe `Roundball`.

Les contraintes qui vous sont données sur les méthodes à écrire sont impératives, et vous garantiront une modélisation correcte et facilement maintenable de votre application lorsque vous aborderez la partie graphique.

Étape 2 - La vue

Dans cette deuxième semaine, nous allons créer une première interface graphique, sans animation. Cette première étape vous permettra de vous familiariser avec la classe `GereDessins`, et de mettre en place les contrôleurs dont vous aurez besoin.

L'interface graphique est constituée d'une fenêtre incipale, sur laquelle est posée un composant de type `tkinter.Canvas`, et, en-dessous, de deux zones de texte (une pour le score, l'autre pour le message `Partie finie !`) et de deux boutons (un pour quitter, un pour recommencer une partie. Un `Canvas` est un composant graphique sur lequel on peut dessiner. C'est sur ce composant que sera représentée la partie.

La classe `VueRoundball` est construite avec une instance de la classe `Roundball` en paramètre. Elle conserve comme attributs :
- le modèle (l'instance de `Roundball`)
- son nombre de lignes et de colonnes
- la `__taille` d'une cellule qui sera initialisée à 40 pixels
- la fenêtre (l'instance de `Tk()`)
- la zone de texte pour le message `Partie finie !` (c.f. figure 2)
- la zone de texte pour le score
- le canevas (l'instance de `Canvas`
- un couple de coordonnées `__prems` initialisé à `(-1,-1)`
- une instance de `GereDessins`, (que nous nommerons `__gd` dans le sujet)

La classe `GereDessins`, qui vous est fournie dans le fichier `gereDessins.py`, permet de dessiner des disques de couleur sur un canevas, d'effacer des disques, de marquer une case sélectionnée par le joueur et également de faire des animations (mais ce ne sera pas pour cette semaine). Le constructeur de `GereDessins` prend en paramètre :
- la vue
- la fenêtre
- le canevas
- la taille d'une case
- un temps de délai pour les animations : vous n'avez pas à vous en préoccuper, il est très bien initialisé par défaut à 40 millisecondes.

Votre travail, cette semaine consiste à implémenter la classe `VueRoundball` :

1. créez le constructeur de votre interface graphique ;
   - un `Canvas` est un composant graphique qui se construit simplement :
     `self.__can = tkinter.Canvas(self.__fenetre,width=largeur,height=hauteur)`

où `hauteur` et `largeur` correspondent au nombre de pixels nécessaires pour le canevas, soit, respectivement, *nbre de lignes* × `__taille` et *nbre de colonnes* × `__taille` ;
— quand vous aurez créé votre instance de `GereDessins`, vous pourrez lui demander de dessiner des disques de la couleur définie par le modèle en utilisant la méthode
`dessine_disque(self,i,j,val)`
où `i` est le numéro de ligne, `j` le numéro de colonne, et `val` une valeur positive ;

2. associez le bouton *Quitter* à l'action de quitter l'application
3. associez le bouton *Recommencer* à l'action de réinitialiser la partie (méthode `reinit(self)` à écrire, voir plus loin) ;
4. associez un click avec le bouton gauche (événement `"<1>"`) sur la canevas à la méthode `click(self,event)` que vous allez écrire ;
5. écrivez deux méthodes qui ne font rien pour le moment, juste ainsi :

```
def disable_canvas(self):
    // à écrire plus tard (semaine 4)

def enable_canvas(self):
    // à écrire plus tard (semaine 4)
```

6. écrivez ensuite les contrôleurs :
   (a) écrivez la méthode `reinit(self)` qui réinitialise le modèle puis redessine toute la grille en fonction des valeurs du modèle ;
   (b) écrivez la méthode `click(self,event)` :
      i. dans un premier temps, vous pouvez récupérer les coordonnées de l'endroit où le click de souris a eu lieu par `event.x` (abscisse) et `event.y` (ordonnée). Dans un `Canvas`, le point origine est en haut à gauche. Déduisez-en le numéro de ligne et le numéro de la colonne de la case cliquée par le joueur (rappel : vous connaissez la `__taille` d'une case). Affichez temporairement (à l'aide de `print`) ces coordonnées pour vérifier vos calculs ;
      ii. maintenant, affectez ces coordonnées à `__prems` et demandez systématiquement à `__gd` de marquer la case par la méthode
      `pose_marque(self,lig,col)`
      de la classe `GereDessins`. Chaque fois qu'une nouvelle case est choisie, la marque doit être retirée de l'ancienne (simplement en redessinant un disque de la bonne couleur aux anciennes coordonnées de `__prems` à l'aide de `dessine_disque`) ;
      iii. votre application est maintenant prête pour la dernière étape de cette semaine. Il s'agit de donner un contenu correct à la méthode `click(self,event)` :
         — s'il s'agit de la première case sélectionnée ou, si la case sélectionnée n'est pas une voisine immédiate de la case précédemment sélectionnée, alors on ne change rien au comportement précédent ;
         — sinon, il faut envoyer l'information au modèle que le joueur veut permuter la première case sélectionnée et la deuxième. Si la liste des éléments à supprimer retournée par le modèle est vide, alors la permutation n'est pas effective. Sinon, il faut alors demander au modèle de se mettre à jour, puis réafficher toute la grille en fonction des nouvelles valeurs dans la matrice du modèle.

Le comportement de votre jeu doit donc être le même que la semaine dernière, mais avec un affichage graphique.

Étape 3 - Le modèle (2) - Prévoir les animations

L'élément qui manque maintenant est la gestion des animations : dans ce jeu, il est important de pouvoir visualiser comment les modifications se font. Mais l'interface graphique ne peut pas, toute seule, deviner quelles sont les animations à mettre en oeuvre. C'est le modèle qui va donner les éléments d'information, et également l'ordre dans lequel on doit voir les animations :

— d'abord effectuer la permutation
— puis effacer les éléments qui doivent disparaître
— puis faire tomber les balles dans les lignes les plus basses, en remontant de ligne en ligne
— et on recommence avec de nouveaux éléments qui pourraient avoir à disparaître etc etc

Toutes ces informations sont en fait déjà calculées par le modèle, dans la méthode `change_valeurs`. Il ne vous reste (presque) plus qu'à les transmettre à la vue, qui les transmettra à l'objet `__gd` pour qu'il exécute la séquence d'animations grâce à sa méthode `execute_animations(self,anims)`. Cette méthode prend en paramètre une liste d'animations à effectuer dans l'ordre de la liste. Les animations sont données sous l'une de ces forme :

— un tuple de trois entiers `(lig,col,couleur)` : indique que la case `lig,col` doit être redessinée avec une balle de couleur `coul`
— un tuple de deux entiers `(lig,col)` : indique que la case `lig,col` doit être effacée
— un tuple de quatre entiers `(lig_dep,col,lig_arr,coul)` : indique qu'une balle de couleur `coul` doit tomber, dans la colonne `col`, de la ligne `lig_dep` à la ligne `lig_arr`.

Vous devez dans un premier temps modifier la méthode `change_valeurs` de votre modèle pour que cette méthode retourne la liste des animations à faire. Cette liste peut contenir simplement la suite des animations, où bien être sous une forme plus compliquée comme

```
[[liste_elts_à_effacer],[liste_balles_tombent],
 [liste_elts_suivants_à_effacer],[liste_balles_suivantes_tombent],...]
```

comme cela vous arrange. Vous pouvez aussi ne pas intégrer la première opération de permutation de contenus dans votre liste d'animations, et faire en sorte d'exécuter les permutations `avant` le reste des actions de dessin.

Dans un deuxième temps, vous devez modifier le contrôleur `click` de votre vue pour récupérer cette liste d'animations et la transmettre à l'objet `__gd` et appeler la méthode
`execute_animations(self,anims)`
de cet objet. Vous n'avez plus à réafficher complètement la matrice du jeu. Si vous n'avez pas intégré les permutations à la liste des animations, n'oubliez pas de les dessiner avant de lancer les animations.

Maintenant vous devez pouvoir jouer !

Étape 4 - Derniers ajustements - améliorations

C'est le moment d'effectuer les dernières fonctionnalités : la détection de la fin de partie dans le modèle (avec prise en compte dans la vue), et les méthodes `disable_canvas` et `enable_canvas`.

Une partie est finie lorsqu'il n'existe aucune permutation de deux cases voisines qui engendre la création de nouveaux "*trios*". Une autre manière de voir les choses, c'est que s'il existe une permutation de deux cases voisines (horizontalement, ou verticalement), qui fasse que `recherche_trios` détecte des cases à supprimer, alors la partie n'est pas finie. Écrivez la méthode `partie_finie` dans votre modèle, puis prenez la en compte dans votre vue pour prévenir le joueur.

Les méthodes `disable_canvas` et `enable_canvas` doivent permettre de supprimer les actions d'un click sur le canevas, ou de les ré-autoriser. C'est la méthode `unbind(self,event)` qui vous permettra de résoudre ce problème. Ces méthodes sont appelées par l'objet `__gd` pour supprimer les clicks perturbateurs pendant le dessin des animations. Elles peuvent vous être utiles pour une partie finie (et pour une réinitialisation de partie).

:

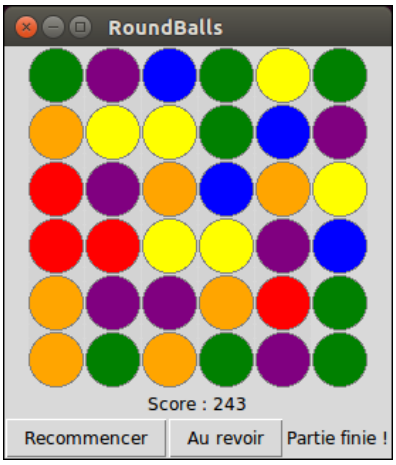

FIGURE 2 – Roundball : interface graphique finale du projet

Cette dernière semaine vous permettra également de terminer les fonctionnalités qui vous manquent, de peaufiner votre code (améliorer la conception en décomposant des fonctions complexes en plus petites fonctions, simplification de vos algorithmes), de soigner votre documentation...