Université de Paris 2020/2021

« Corona Bounce »

Projet de

Aliche Zahra, Bragina Natalia, Golikova Anna, Sahi Kenza, Raveneau Emilien



[Introduction 3](#_Toc71992801)

[Problématique 3](#_Toc71992802)

[Cahier des charges 3](#_Toc71992803)

[Objets principaux et fonctions basiques 3](#_Toc71992804)

[Réglages possibles 3](#_Toc71992805)

[Coté GUI 4](#_Toc71992806)

[Diagrammes des classes 5](#_Toc71992807)

[Méthodologie et déroulement du projet 7](#_Toc71992808)

[Réalisation de contamination 7](#_Toc71992809)

[Réalisation de scenarios 7](#_Toc71992810)

[Réalisation de Model–View–Controller 8](#_Toc71992811)

[Gestion des threads 8](#_Toc71992812)

[Réalisation de la partie graphique, JavaFx 9](#_Toc71992813)

[Maven 10](#_Toc71992814)

[JavaDoc 10](#_Toc71992815)

[Tests 10](#_Toc71992816)

[Intégration 11](#_Toc71992817)

[Difficultés rencontrées 13](#_Toc71992818)

[Problèmes connus 15](#_Toc71992819)

[Pistes d’extensions 15](#_Toc71992820)

[Gestion du projet et répartition des taches 15](#_Toc71992821)

[Conclusion 16](#_Toc71992822)

# Introduction

Pour notre UE de projet de notre deuxième année de Licence en informatique à l’université de Paris Diderot, nous avons travaillé sur un projet de Corona Bounce. Au cours de ces trois mois de durs labeurs, nous avons utilisé nos connaissances déjà acquises au cours de nos parcours respectifs, ainsi que de nouvelles apprises au cours de ce délai, pour offrir un simulateur d’épidémie. Ce projet est réalisé en groupe, dans lequel chacun de nous a apporté sa pierre à l’édifice. Il est constitué de cinq étudiants de parcours informatique/math info :

- ALICHE ZAHRA

- BRAGINA NATALIA

- GOLIKOVA ANNA

- SAHI KENZA

- RAVENEAU ÉMILIEN

Le projet réalisé est proposé par nos encadrants : Mr Yan Jurski, et sous la direction de Mr Aldric Degorre, Responsable des projets du département informatique de l’Université.

# Problématique

Développer un simulateur d’épidémie qui permet l’observation de sa propagation dans des populations virtuelles en variant les mesures et les paramètres. Ainsi qu’en analysant les courbes de contamination et de guérison, de pouvoir faire un point sur l’efficacité des stratégies adoptées et de déduire les facteurs inhibiteurs et stimulateurs de la diffusion de la maladie.

# Cahier des charges

Créer le moteur de la simulation capable de gérer les déplacements et les contaminations.

Réaliser une application qui offre la possibilité de lancer l’animation avec des paramètres de base. Pouvoir changer les paramètres (à partir de GUI).

## Objets principaux et fonctions basiques

Les objets principaux sont des points qui représentes les individus, qui bougent et qui changent son couleur selon son état (en bonne santé, contaminés, malades et guéris).

Les graphes affichent en temps réel le développement de l’épidémie en affichant le nombre de personnes malades (contagieuses ou porteur sain), saines et guéries.

En appuyant sur les boutons du menu principal on peut :

1. Lancer le mouvement (« Start »)
2. Arrêter le mouvement et le reprendre (« Pause/Resume »)
3. Changer les réglages (« Settings »)
4. Recharger les populations à nouveau tout en gardant les réglages ainsi que scenarios (« Reset »)
5. Montrer la légende pour chaque élément de la scène (« ? »).

## Réglages possibles

Le programme permet à l’utilisateur d’effectuer des changements de paramètre ou de scénarios.

Les manettes de réglages à partir de « Settings » permettent de faire varier les valeurs de paramètres suivantes :

1. Maladie

* Durée de l’incubation ;
* Durée de recouvrement ;
* Durée de l’immunité ;
* Rayon de contamination ;

1. Population

* Nombre de personnes ;

1. Espace

* Nombre de murs/frontières (qui restreignent le mouvement des points) ;
* Vitesse de fermeture les frontières.

Outre cela nous proposons de choisir parmi les scenarios suivants :

* « Soft Lockdown » (déplacement s’effectue dans une zone de quelque rayon par rapport de position initial d’individu).
* « Boundaries » (déplacement aléatoire mais limité par les frontières).
* « Strict Lockdown » (une grande partie de population est confinée, qu’une petite partie de population se déplace librement – médecines, pompiers etc.).
* « Soft Lockdown + Boundaries » (application deux scenarios à la fois).
* « No scenario » : déplacement aléatoire et non limité.

## Coté GUI

Les fenêtres et ses éléments s’adaptent aux dimensions de l’écran.

Il y a une légende expliquant les objets et les fonctionnalités impliqués.

Les fenêtres sont redimensionnables. Toutes les parties des fenêtres ainsi que les objets de la simulation s’adaptent aux changements de dimensions dynamiquement.

# Diagrammes des classes

Diagramme des classes du modèle :



Diagramme des classes de vue et contrôleur :



Diagramme de main classes : 

# Méthodologie et déroulement du projet

Nous avons réalisé tout ce que nous avions défini initialement dans le cahier des charges.

## Réalisation de contamination

La **Zone** représente l’environnement de vie d’une population. Elle est caractérisée par une hauteur et une largeur. Nous avons une population ayant un certain nombre d’individus, ils circulent dans un environnement Zone. Les individus sont représentés par des points de la Zone d’étude.

Les individus représentés par la classe **Individual**, peuvent prendre 4 états : sain, porteur sain, porteur contagieux et guéri. Il y a 4 classes représentant ces 4 états de santé possible de chaque individu : **Healthy**, **Incubating, Sick** et **Recovered**.

La capacité de l’Individu à se déplacer est réalisée par la classe **CoquilleBille**, qui encapsule un individu. CoquilleBille possède une position et une vitesse découpé en un vecteur x et en un vecteur y.

**ConfinedBille** est un type spécial de CoquilleBille encapsulant les individus d’une population soumise à un confinement (SoftLockdown). Ils ont une vitesse bornée par une valeur maximale et circulent selon le scenario de confinement.

Les objets **Wall** sont des murs restreignant les déplacements des individus. Chaque mur a une position fixe X et une position Y variable ainsi qu’une largeur. Dans l’implémentation, les murs peuvent se déplacer vers le bas uniquement. Ses objets peuvent être vue comme des frontières que l’on ferme pour enrayer la propagation de l’épidémie.

Au début de la simulation, la population est dispersée aléatoirement sur la surface de la zone (de façon à ce qu’aucun point de soit créer sur un autre tant que c’est possible). Par la suite, les points circulent avec une certaine vitesse aléatoire bornée par une valeur maximale (que l’on peut changer). La contamination se produit si un point sain est dans le rayon de contamination d’un point malade ou contagieux. Il rentre dans une phase d’incubation (la durée d’incubation est 3s[[1]](#footnote-1)). Il devient guéri après une période de temps (dure 8s1) et acquiert une immunité temporaire (dure 5s1) qui le protège contre une nouvelle infection.

## Réalisation de scenarios

**No scenario :** déplacement aléatoire et non limité sans protocoles sanitaires.

**SoftLockDown :** Dans ce cas de figure, un type spécifique de CoquilleBille(s) est créé : ConfinedBille(s) caractérisé par une vitesse et un rayon de déplacement restreints. Les individus ne se déplacent que dans un voisinage de leur position initiale et avec une vitesse réduite. Le principe de ce scénario est de réduire la vitesse de déplacement des individus et les forcer à rester dans le voisinage de leur domicile, pour cela on calcule la distance entre la position courante et la position de départ d’un individu si elle dépasse le rayon défini alors il faut forcer l’individu à revenir à sa zone de départ sinon on le laisse circuler librement mais avec une fréquence de déplacement réduite. L’exemple concret de cette politique est celui que la France a appliqué en mars 2020 en autorisant les sorties mais pas loin de quelques kilomètres des domiciles des citoyens.

**StrictLockdown :** Une grande partie de la population est forcée de rester chez elle, aucun déplacement n’est toléré. Une minorité de gens peuvent circuler, cela concerne notamment ceux qui exercent des métiers indispensables au bon fonctionnement et à la sécurité de la population, policiers, docteurs, pompiers, livreurs, urgences etc. Alors 70-80% des points ont la vitesse nulle, et 20-30% se déplacent aléatoirement sans aucune limitation de vitesse.

**Boundaries :** Des frontières sont fermés progressivement et divisent la zone en sous-zones pour limiter les déplacements (déplacements régionaux uniquement).

**Boundaries + Softlockdown :** Une combinaison entre frontières et confinement cités plus haut.

Les choix de scenarios sont réalisés grâce à trois variables booléennes de l’état de population suivantes : l’une pour donner la restriction de mouvement de plupart des individus, l’autre pour mettre les murs, la troisième pour constituer la population de ConfinedBilles.

## Réalisation de Model–View–Controller

Notre projet est organisé selon MVC design pattern. Pour effectuer la connexion entre Model, Vue et Controller nous utilisons deux interfaces :

* **Displayable** qui sert les fonctions nécessaires à Model pour bien être affiché dans Vue : les points et les murs, dessiner les graphiques et gérer les timers.  
  Alors, Displayable contient les getters de liste de points, de nombres de personnes en bonne santé, malades, contagieux, guéris, ainsi que les getters de murs, ses positions et sa hauteur, les fonctions servent à sauvegarder la statistique en historique et fermer les tâches de timer.
* **Controllable** qui contient les fonctions nécessaires à transmettre au modèle les paramètres pris à partir de GUI. Donc ce sont les getters et les setters de la zone, de population, de maladie, des murs et de scenarios, ainsi que les fonctions pour changer l’état de threads.

Nous avons créé la classe **Controller**, son rôle consiste à être auxiliaire au Main Controller et Settings Controller.

Controller contient les valeurs initiales des paramètres pour lancer le modèle. En outre, en tant qu’il implémente l’interface Controllable, il offre les fonctions nécessaires pour changer ces valeurs initiales à partir de l’UI (user interface) pendant l’usage de l’application.

## Gestion des threads

Dans notre projet nous avons implémenté le multi-thread architecture.

On dispose 5 threads en tout : l’un qui est principale (celui de App.java) et deux pour chaque une des populations (un pour gérer le déplacement des points, l’autre pour gérer les cycles de maladie et sauvegarder l’historique afin de la représenter en graphiques).

A part de main thread, les threads se créent dans deux classes : Population et Zone.

Le thread de *timer crée dans la Population* gère les classes suivantes :

* Classe Incubating – pour gérer la durée d’incubation de maladie, temps écoulé - Incubating devient Sick. Durée = durée d’incubation.
* Classe Sick – pour gérer la durée de maladie après le contact – dès que le temps s’écoule l’individu Sick change la classe pour Recovered. Durée = durée de maladie.
* Classe Recovered – pour gérer la durée de non contamination, temps écoulé - recovered devient healthy. Durée = durée d’immunité.
* Classe Population – pour sauvegarder la statistique afin de la représenter en graph.
* Classe Wall – pour fermer les frontières.

Le thread de *timer crée dans la Zone* sert à effectuer le déplacement des individus.

L’animation dans GUI est attachée au *thread principal*.

Afin d’assurer la synchronisation des processus, on met à pause tous les timers à partir de création des instances. Pour cela on utilise une variable enum (« Paused » au début) qui est passée par l’interface Cotrollable dans toutes les classes concernées (Zone, Population, Sick, Incubating, Recovered, Wall). Ce n’est qu’appuyant sur le bouton « Start » que cette variable change sa valeur pour « Working » et tous les processus commencent à tourner. Le même principe nous utilisons pour mettre model en pause (bouton « Pause / Resume »).

On assure la synchronisation des données grâce à la classe Reentrantlock.

Elle est utilisée :

* Pour changer la valeur de l’enum de l’état de threads (interface Controllable et classe Controller)
* Dans le cycle de dessin de graphiques afin de synchroniser les processus de lecture et d’écriture de statistiques (classe Data).

Reentrantlock offre une fonctionnalité transparente et plus souple que la synchronisation implicite offerte par synchronized. Ainsi Reentrantlock nous permet-il de séparer les processus de lock en unlock afin de les implémenter dans les fonctions à part. Grâce au Reentrantlock les plusieurs objets de différentes classes partagent un seul lock ce qui serait impossible avec synchronized. Cela nous permet d’assurer l’accès unique à partir de thread principal à l’objet de Data pendant le dessin les graphiques et de procéder à la synchronisation après. La lecture et l’écriture de enum state, qui s’effectue à partir de plusieurs méthodes et plusieurs threads, sont aussi parfaitement contrôlées avec Reentrantlock.

## Réalisation de la partie graphique, JavaFx

Dans le cahier des charges, il a été prévu de réaliser une interface graphique pour représenter les simulations et des moyens de choisir divers scénarios et paramètres et de les appliquer. Une possibilité de redimensionnement a été aussi envisagée.

Nous avons réussi à implémenter ces fonctionnalités. En particulier, pour le redimensionnement nous avons choisi de fixer les largeur et hauteur minimales des fenêtres (pour toujours pouvoir voir les simulations et lire les légendes) et de permettre d’agrandir les fenêtres. Aussi, les dimensions initiales prennent en compte les paramètres de l’écran de l’utilisateur.

Pour la réalisation de la partie graphique du projet nous avons choisi d’utiliser **JavaFx** comme UI framework. Dans notre choix nous sommes basés sur les critères de performance et de facilité d’utilisation grâce aux CSS et fxml.

La partie graphique est réalisée à l’aide du format fxml. Deux fichiers fxml, corona bounce.fxml et settings.fxml correspondent aux deux fenêtres du programme, c’est-à-dire la fenêtre principale où la simulation se passe et la fenêtre des paramètres.

Le format fxml nous a permis de regrouper les éléments essentiels de la partie graphique. Pour ce faire nous avons utilisé l’application **SceneBuilder** et nous avons aussi modifié le code directement.

La création des éléments graphiques est possible aussi bien dans fxml que dans les parties .java ; nous nous sommes servis de ce fait pour dessiner les graphes. En effet, on a déplacé la création des graphiques en java, car cela nous permet de les appliquer les styles à l’aide de .css plus facilement.

La création des graphes se fait dans la classe Graph qui est utilisé par MainController. Le dessin des frontières et des points/individus s’effectue par la classe Renderer.

Comme JavaFX ne supporte pas « Stacked Area 100% chart », nous avons imité ce type de chart avec « Stacked Area chart » en prenant :

* Le nombre total des individus comme le nombre des individus sains (la couche la plus profonde) ;
* La somme de nombre des individus malades, contagieux et guéris comme le nombre des individus malades (la couche du milieu) ;
* Le nombre des individus guéris comme le nombre des individus guéris (la couche superficiel) ;

Ses 3 couches représentent la répartition des sains, malades/contagieux et guéris sur 100%.

Pour faciliter l’usage de l’application, un système de légendes a été implémenté à l’aide de tooltips (définis aussi bien dans MainController).

On définit certains styles dans style.css ; mais une partie des caractéristiques de style se fait dans les fichiers. fxml directement (surtout dans les cas où les éléments sont difficilement accessibles via css).

Pour la création des fenêtres, on utilise les attributs essentiels d’une application JavaFX, à savoir Scene et Stage que nous définissons dans App.java.

## Maven

Nous avons utilisé Maven pour faciliter la compilation et l’exécution du projet pour l’ensemble du groupe. Maven nous a paru plus facile d’utilisation que Gradle, moins contraignant (sur l’utilisation de fichier sauvegarder en local par exemple) et mieux connu (par nous) que ses équivalents.

## JavaDoc

La JavaDoc est disponible dans le projet et contient toutes les descriptions de fonctions, classes et variables que nous avons jugé nécessaire d’ajouter. Nous avons préféré utiliser l’anglais dans la JavaDoc (comme dans le reste du projet) pour rendre notre projet plus proche de ce qui se fait en entreprise.

Nous aurions aimé pouvoir afficher la Javadoc en un clic en l’hébergeant sur un site statique telle que Gitlab Pages, malheureusement cette fonctionnalité ne semble pas prise en compte dans notre version de Gitlab. La Javadoc peut-être lancer en ouvrant « code/docs/javadoc/apidocs/index.html » dans un navigateur.

## Tests

Des tests unitaires ont été effectués pour des fonctions simple en testant d’abord des fonctions très basique puis des fonctions plus globales qui réutilisent les petites fonctions.

Les tests sont dans leur propre branche pour ne pas être inclus dans le .jar finale.

Les classes de test sont toujours nommées de la même façon, ClassNameTest, les fonctions sont nommées testFunctionName (+ un numéro à la fin s’il y a plusieurs tests pour 1 seule fonction).

Les tests ne sont pas exhaustifs, mais ils nous ont quand même permis d’identifier plusieurs soucis, parfois rare et donc difficile à détecter autrement. Ils permettent en outre la non régression sur de petite briques de code.

Le GUI n’étais pas testé en mode automatique. Par contre, les tests de certaines fonctions dans GUI a été fait directement dans le code. De telle manière nous testaient :

* *Le temps d’exécution* des fonctionsd’affichage de points, de murs, de graphs (tous séparément/indépendamment).
* *Fonctionnement de buttons et de menubars*.
* *La synchronisation des processus* dans le model et dans la vue (par exemple propagation de maladie).

La plupart de ces tests a été supprimées après la vérification, les restes sont commentées dans le code.

## Intégration

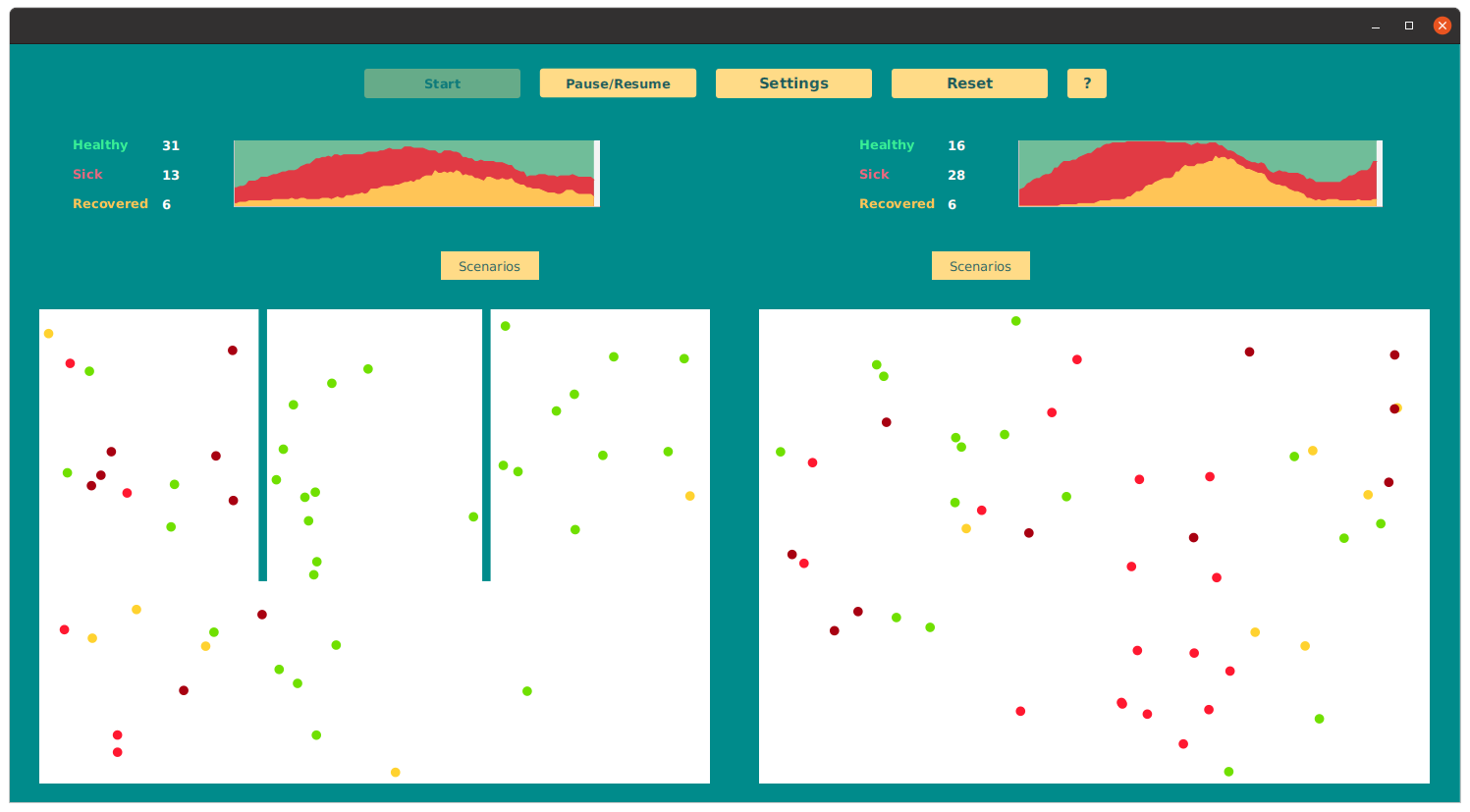
Le projet est utilisable sous la forme d’un .jar incluant les dépendances utilisées et les ressources pour une plus grandes facilité d’utilisation.

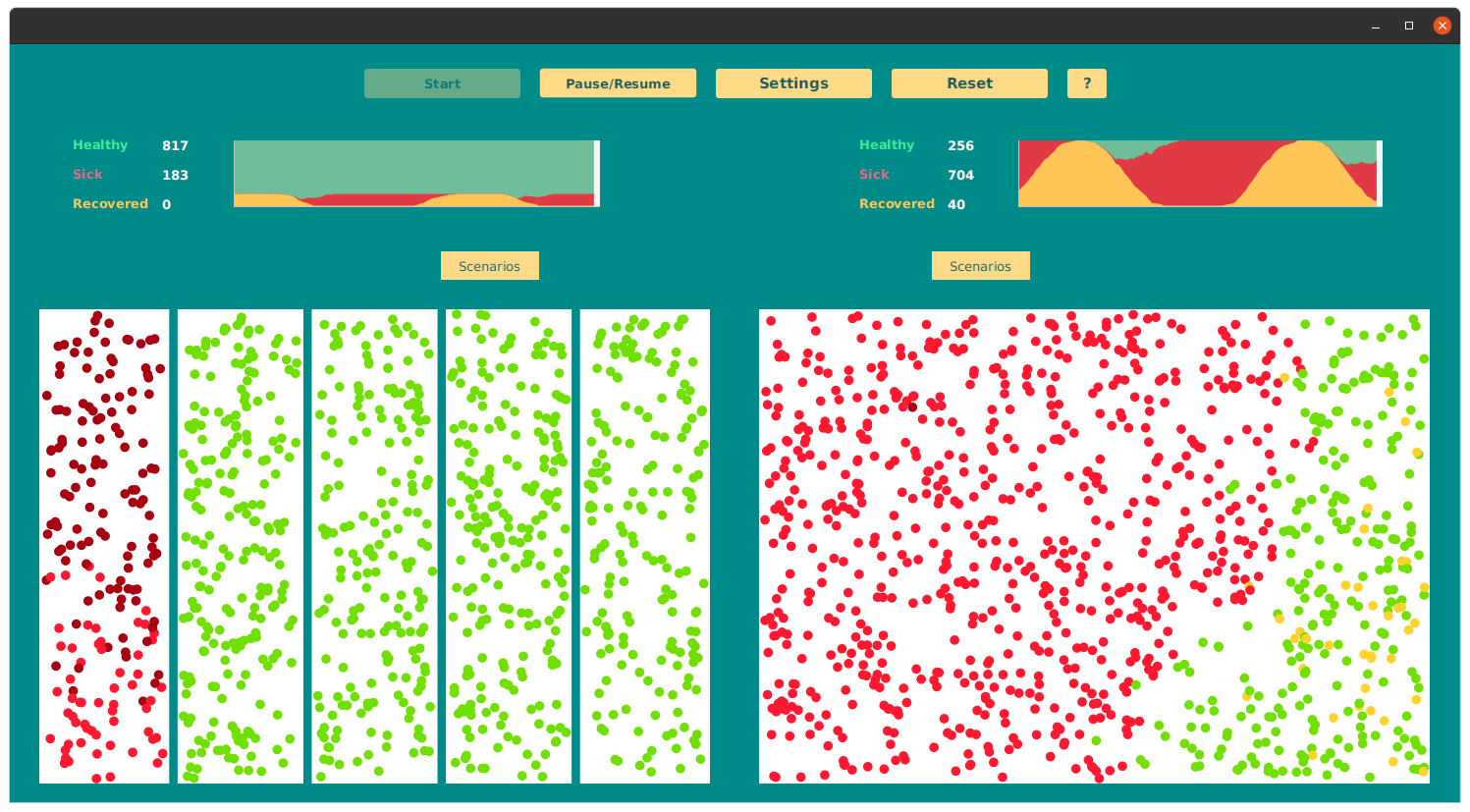
La seule chose qu’il est nécessaire d’installer pour lancer le .jar, c’est la dernière version de Java.











# Difficultés rencontrées

Quelque part on a découvert *la charge importante de CPU* par le programme. Après les tests on a trouvé que cette surcharge est provoquée par le fonctionnement des plusieurs populations – deux qui sont actives et certaines en background. En effet après le changement de controller et la création des nouvelles populations, les points des anciennes populations n’étaient pas détruits par le Garbage collector, car les timers et ses taches n’étaient pas clôturés. On a résolu ce souci en appliquant la fonction de clôture correcte de TimerTask et de Timer avant de la création des nouvelles populations.

*Déphasage de l’épidémie avec le mouvement.* Dès qu’on avait fait fonctionner le bouton « Start » qui lance la fonction moving() sur les Zone(s), nous nous sommes rencontrés le fait que l’épidémie s’évolue de manière cachée : si on n’appuient pas « Start » immédiatement, au moment de lancement de mouvement les individus étaient guéris. A partir de cela on a commencé à étudier quels timers où et quand on crée dans notre programme. Plus tard on a découvert comment gérer l‘état de tous les timers par une seule variable. Cela nous a permis de contrôler et donc de synchroniser le début du mouvement, des contaminations et des durées de maladie.

La gestion de threads avait nous posé beaucoup de problèmes, car on a un certain nombre de timers qui créent chacun son thread et ces threads partagent les mêmes données.

Avec la création des graphiques on se rencontré avec le problème de *présentation des données pour l’axe d’abscisse* (pour celui d’ordonnée on avait les données statistiques) et de *mis à jour les données de manière dynamique*. L’implémentation de FIFO a nous permis de résoudre ce problème. On a choisi le vecteur comme la structure de données pour FIFO, car il est thread-saved.

Pour éviter les lags suite aux surcharges *de CPU par dessin des graphiques de populations étendues,* nous adaptons la fréquence de mise à jour les graphiques relativement à la taille de population (10 fois/sec pour les populations au plus de 500 perçons et 5 fois/sec sinon).

Il a été très difficile *d’empêcher les points de passer à travers les murs en x et en y*, en prenant en compte l’épaisseur des murs et des CoquilleBille ainsi que les déplacements des murs.

|  |  |  |
| --- | --- | --- |
| L’idée d’origine était d’essayer de savoir si le haut d’un point allait passer dans le mur pour savoir si on devait faire rebondir la coquilleBille. | Mais cela ne fonctionnait pas toujours, dans le cas où le centre d’un point ne rentrait jamais dans le mur, mais ou le bord passait à travers. | L’adaptation qui fonctionna fus de vérifier les 2 coins en haut à droite et en haut à gauche pour éviter que des coquilleBilles puisse traverser un mur. |
|  |  |  |

Les soucis de billes dans des murs suite à un déplacement de murs ont été géré autrement, si la bille se situe dans la zone où le mur va aller, on la décale vers le bas d’autant que nécessaire. (Ou vers les côtés si on ne peut plus pousser les billes vers le bas sans les faire sortir de la zone).

En ce qui *concerne les fichiers fxml*, il a été difficile d’utiliser dans le travail en même temps Scene Builder et parfois devoir refaire les mêmes choses manuellement (par exemple pour la disposition des éléments pour avoir le bon redimensionnement) car cela entraînait des changements dans certains paramètres.

Ensuite, toujours côté graphique, comme nous avons choisi de représenter les fenêtres à l’aide de stage et scene de JavaFX, nous n’avons pas trouvé ensuite le moyen de passer de dimensions de fenêtres établis une fois au début pour avoir une fenêtre de Settings plus petite. Nous avons remédié en quelque sorte à ce problème en mettant les éléments de Settings au milieu pour équilibrer ainsi leur disposition.

Et le dernier souci de l’interface graphique est encore lié à SceneBuilder : il n’y avait pas dedans de fonctionnalité pour ajouter un graphique, alors on l’a implémenté directement dans le code java ; ainsi, l’interface graphique combine deux type d’éléments : ceux provenant du code fxml et ceux du code java.

# Problèmes connus

* + - 1. Au-dessus d’un certain nombre d’individu (500 à 3000 Individu en fonction des CPU) l’interface graphique ne suit pas le modèle et freeze. Cela est apparu en même temps que l’ajout de fonctionnalité qui nécessite un parcours de l’ensemble des points (comme les rebonds entre les points). La solution envisagée : séparation de la Zone par secteurs et comparaisons des points aux secteurs les plus proches. Cela nécessiterait de découper la liste des points en sous listes avant d’interagir avec des points proche (complexité en Θ(n)). Cela permettrait de n’avoir qu’à parcourir qu’une toute petite liste de voisin pour réaliser les interactions d’1 points. Si l’on suppose qu’on peut découper l’espace en secteur de façon à ce qu’il y a 10 points en moyenne par secteur, la complexité moyenne pour faire interagir tous les points serait donc Θ(n)\*10 = Θ(n), bien mieux que la complexité actuelle en Θ(n2).
      2. Quelques points s’attachent sans respect ses gabarits et ainsi continus à se déplacer ensemble. Parfois il y a un rebouclage qui se produit entre eux.

# Pistes d’extensions

* + - 1. Frontières initialement fermés, puis l’ouverture s’effectue avec le lancement du scenario « Open bouderies ».
      2. Ouverture partielle de frontière (juste une petite fenêtre où que quelques individus peuvent entrer).
      3. Réglage de la vitesse des points à partir de Settings.
      4. Ajout la mortalité.
      5. De plus il est possible de réaliser ce projet de manière moins abstrait et donc plus concret, en implémentant des classes économiques et sociales en dedans de population : des salaries, des écoliers, des retraités, les professionnels du secteur d’urgence et de sécurité publique etc. Dans ce cas les scénarios seraient plus concrets.
      6. Ajouter un scénario vaccination pour visualiser l’évolution de la pandémie après la mise en place du vaccin.
      7. Mise en place de la politique du troupeau qui est basée sur le principe de permettre aux gens de se déplacer librement en autorisant la propagation de l’infection et de cette façon les personnes infectées après le rétablissement elles vont gagner automatiquement l’immunité contre ce virus et donc la prochaine infection ne prendra pas beaucoup de temps pour le rétablissement (donc il faut jouer sur la durée de guérison).
      8. Mettre en place d’un épicentre de pandémie (ex : Chine) pour détecter les facteurs qui augmente la propagation du virus et essayer de les éliminer.

# Gestion du projet et répartition des taches

Dans notre travail nous sommes basés sur le cahier des charges que nous avons élaboré tout au début de notre travail. Pendant les périodes de modélisation et de développement extensif la planification commune était organisée hebdomadairement en forme de réunions en distanciel. En plus, les réunions entre les différents membres d’équipe se faisaient selon le nécessité. Pendant les périodes de « travaux de finition » les réunions étaient plus rare.

Nous avons utilisé GitLab comme le système de gestion des versions.

En termes d’organisation à l'intérieur de l’équipe nous avons adopté le principe standard : division sur back-end et front-end.

*L’équipe de back-end* était constituée par Aliche Zahra, Sahi Kenza at Raveneau Emilien. *L’équipe de front-end* était constituée par Bragina Natalia et Golikova Anna, mais Natalia s’est également occupée de certaines fonctionnalités du back-end.

Plus précisément les taches étaient reparties de manière suivante :

- Séparation entre la partie dynamique et statique des points en mettant la classe CoquilleBille, classe position et Individual \_kenza.

* Ajout de Maven au projet – Kenza
* Design et implémentation des interfaces de MVC – Natalia
* Design et implémentation de fonctionnalités de Population – Kenza, Emilien, Zahra
* Design et implémentation de cycle de contamination-maladie-guérison (par les classes Individual, Healthy, Sick, Recovered, Incubated etc.) – Kenza, Emilien, Zahra
* Design et implémentation de déplacement aléatoire des points – Kenza
* Implémentation des méthodes d’accès aux données du modèle par l’interface graphique – Kenza
* Design et implémentation de fonctionnalités de Wall – Zahra, Kenza
* Design et implémentation de fonctionnalités de Zone – Emilien, Kenza
* Gestion de statistiques – Kenza, Natalia
* Implémentation en modèle de scenario « Strict Lockdown » – Zahra
* Implémentation en modèle de scenario « Soft Lockdown », classe CoquilleBille – Kenza
* Rebonds contre les murs – Emilien
* Rebonds entre les points – Kenza, Natalia
* Design et implémentation de fonctionnalités de SettingsController, sauvegarde des paramètres des Settings – Anna
* Design et implémentation de fonctionnalités de MainController – Natalia, Anna
* Design et implémentation d’App classe – Anna
* Intégration de Controller dans le cycle de recréation et de reloads des fenêtres – Anna
* Redimensionnement et pris des bonnes dimensions par les fenêtres – Anna
* Design, maquettes de fenêtres, fxml-s – Anna, Natalia
* Redisposition manuelle des éléments de Settings et MainController - Anna
* Gestion de threads – Natalia
* Visualisation des points, des murs et des graphiques – Natalia
* Tests – Emilien
* JavaDoc – toute l’équipe

# Conclusion

Nous avons fait preuve d’une patience et d’une camaraderie qui ont été des atouts dans l’élaboration de notre projet. La répartition du travail et la présence de chacun à toutes les réunions ont permis un développement dans de très bonnes conditions.

1. Paramètre qui peut être changé à partir de settings. [↑](#footnote-ref-1)