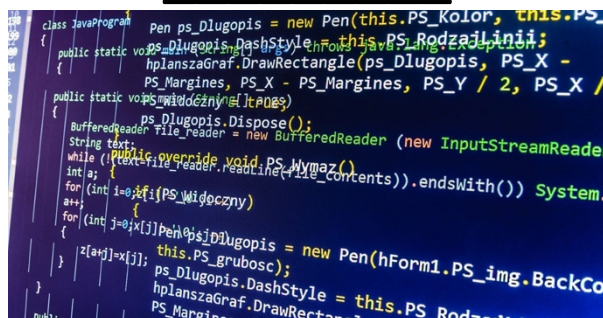

Rapport de synthèse

Simulation d'évacuation d'une foule



```
class JavaProgram {
    Pen ps_Dlugopis = new Pen(this.PS_Kolor, this.PS_RodzajLinii);
    public static void main(String[] args) {
        hplanszaGraf.DrawRectangle(ps_Dlugopis, PS_X -
        PS_Margines, PS_X + PS_Margines, PS_Y / 2, PS_X /
        PS_Widoczny);
        ps_Dlugopis.Dispose();
        BufferedReader file_reader = new BufferedReader (new InputStreamReader
        String text;
        while ((text = file_reader.readLine(file_contents)).endsWith()) System.
        int a;
        for (int i=0; i<PS_Widoczny; i++)
        {
            for (int j=0; j<PS_Wysokosc; j++)
            {
                ps_Dlugopis = new Pen(hForm1.PS_img.BackCo
                ps_Dlugopis.DashStyle = this.PS_RodzajLinii;
                hplanszaGraf.DrawRectangle(ps_Dlugopis, PS_X -
                PS_Margines, PS_X + PS_Margines, PS_Y / 2, PS_X /
                PS_Widoczny);
            }
        }
    }
}
```

Professeurs référents : Malika GRIM-YEFSAH, Marie-Dominique et VAN DAMME Mehdi Zrhal

Chef de projet : Antoine RAINAUD

Équipe de projet : Charles LAVERDURE, Jules PIERRAT et Antoine RAINAUD

- Introduction -

Ce rapport de synthèse vise à rassembler les processus de réflexion ainsi que la chronologie de notre projet de codage en java. Il retrace notre parcours pour arriver jusqu'à un livrable et relève les aspects positifs et négatifs de notre avancée. Il se divise en une partie déroulement général du projet qui présente les défis structurels et d'une partie axée sur la programmation pure.

1 - DÉROULEMENT GÉNÉRAL DU PROJET

Nous avons commencé notre projet java sans diagramme de classe ce qui a été notre première erreur. Nous avons alors créé seulement trois classes : une classe Cellule, une classe Salle et enfin une classe Modele. Dans la classe Cellule nous définissions un type, une position et une taille pour notre cellule. La classe Salle était alors seulement un tableau de Cellule. C'était une erreur d'utiliser un tableau car nous ne pouvions pas profiter des avantages des collections, notamment la longueur variable de la collection. La dernière classe était Modele et tous les calculs et affichage s'y déroulaient, ce type de code n'était pas propre et pas construit.

Nous avons alors repris de zéro notre programme et commencé par créer un diagramme de classe et un diagramme UML. Le premier changement était de séparer nos classes et de créer les bons objets : des individus, des étages, des obstacles... C'est dans ces classes que nous tirons pleinement partie du côté orienté objet du langage java.

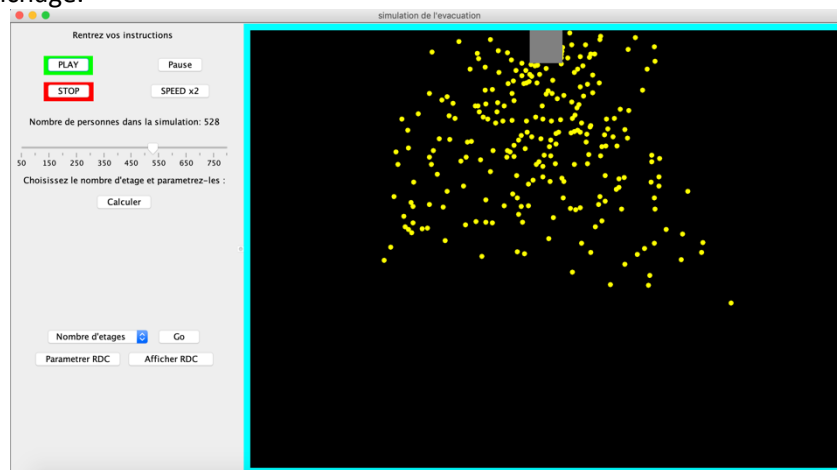
Ensuite, nous créons une classe Modele synthétisant toutes nos classes précédentes : dans cette classe nous générons en fait le bâtiment. Nous créons les étages, les escaliers, les issues ... et affectons des individus aux étages de ce bâtiment.

Enfin, nous créons une classe Simulation où les calculs se déroulent : nous récupérons toutes les données de notre modèle et calculons les positions au fur et à mesure de l'évacuation de nos individus. C'est ici que se trouve toute la partie programmation pure : nous utilisons des Deque, modifions les coordonnées etc...

Nous avons aussi affiché les vecteurs vitesse et accélération des individus pendant leur déplacement enfin de suivre de manière plus complète leurs déplacements.

Pour la partie purement interface graphique avec l'utilisateur, la construction des fenêtres et options de paramétrage se sont révélés plus compliqués que prévu. Après une longue acclimatation aux librairies java.swing et java.awt, nous sommes partis sur l'idée d'une fenêtre principale « splitée » [voir screenshot] en deux verticalement avec une partie gauche pour les contrôles et une partie droite pour l'affichage de la simulation, les dimensions des deux parties étant ajustables. Toutes les informations entrées par l'utilisateur sont acquises sur le panneau de gauche comme des interactions simples telles qu'avec les boutons de contrôle de la simulation mais aussi un paramétrage complet des spécificités des étages. La récupération des informations entrées par

l'utilisateur et son implémentation dans la partie simulation étaient la partie la plus complexe des classes d'affichage.



Écran « splité » paramètres / affichage

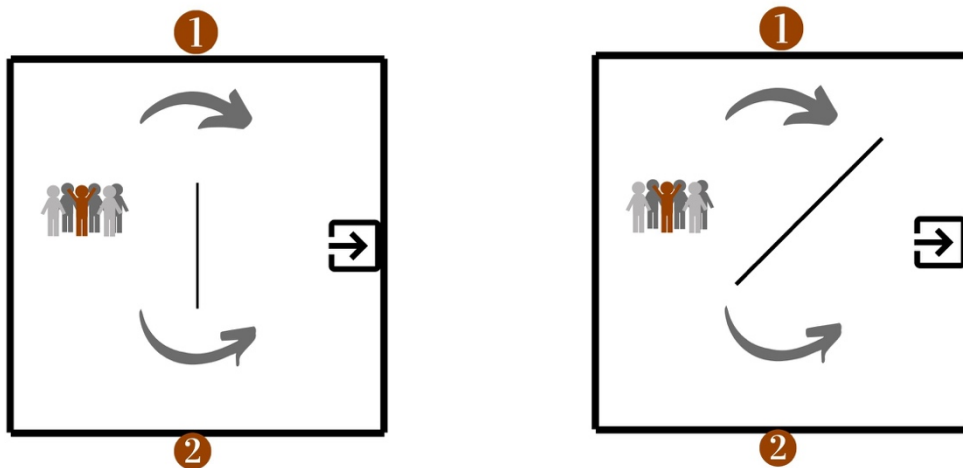
Mais, dans un souci d'amélioration nous avons préféré nous orienter vers des menus en utilisant JDialog, des ActionListener... Ce changement de plan a été très compliqué et a nécessité énormément de temps mais donne un rendu qualitatif.

Avec ce projet nous avons découvert la plateforme GitHub qui permet de déposer et télécharger du code. Cet outil s'est avéré être essentiel pour la mise en commun de travaux à distance.

2 - DÉROULEMENT DU PROJET : ASPECT PROGRAMMATION

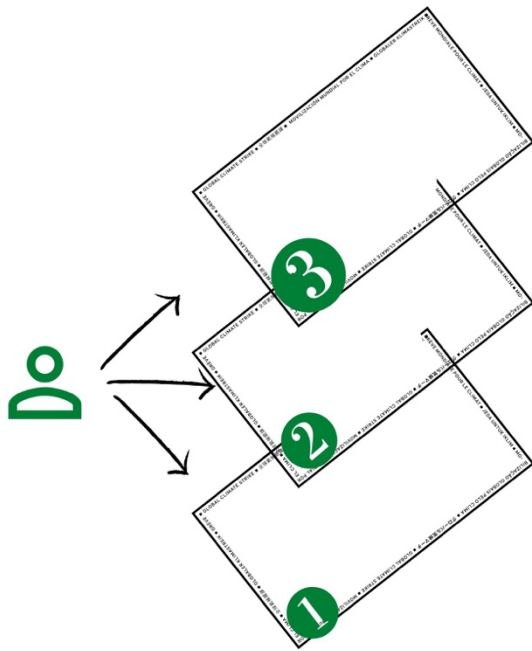
Les méthodes `joinSalles` et `repulsionObstacle` ont soulevé un même problème qu'il a fallu résoudre pendant la programmation de ces dernières : le positionnement relatif des objets en 2D.

Concentrons-nous d'abord sur la méthode `repulsionObstacle`. L'idée est de donner un déplacement différent à un individu si la projection de sa vitesse donne une position correspondant à un obstacle. Mais l'individu ne doit pas seulement éviter l'obstacle, il doit le contourner pour pouvoir atteindre la sortie.



Dans le cas ci-dessus, l'individu a deux choix : suivre la flèche 1 ou 2. Il est clair que dans une optique de chemin le plus court, la flèche 1 est la plus adaptée. Dans le cadre de la méthode `joinSalle` il fallait aussi réussir à bien positionner les salles les unes en fonction des autres. Cette méthode a finalement été abandonnée car l'objet salle n'était plus d'actualité.

L'affectation d'un individu à une salle est un sujet qui a aussi posé problème. Pour répartir un nombre de personnes dans différents étages nous avons décidé de créer une liste de répartition des personnes par salle générée de manière aléatoire. Mais le `Math.random()` de java renvoie un double et les personnes étant entières nous avons dû utiliser transformer un double en entier. Alors, lorsque nous demandons à notre programme de faire apparaître 10000 personnes il n'y en avait que 9998. Nous avons alors compris que la transformation en entier faisait perdre de l'information en arrondissant et ce qui donnait ainsi quelques individus en moins.



La question s'est alors posée pour l'insertion de quelques individus dans les différents étages. Nous demandons de créer 3 personnes pour 3 étages et une seule personne est créée. Cette erreur due aux arrondis présente certes le même nombre d'unité différents (il manque deux personnes) mais dans le premier cas ces 2 individus manquant représentent 0,2% d'erreur et dans le second cas 66,6% d'erreur. Il était alors impératif de changer notre manière de coder cette partie. Nous décidons donc d'affecter un étage à un individu de manière aléatoire. Comme sur l'image ci-contre, un individu est traité de manière unique plutôt que de manière groupée. Chaque individu a une chance sur n où n est le nombre d'étages d'aller à l'étage i , i variant de 0 à n .

La plupart des collections utilisées sont des Deque : Double Ended Queue, ce sont des objets itérables sans limite de taille. Nous pouvons accéder seulement au premier et dernier élément de ces collections. La raison de ce choix est le gros avantage du Deque sur une liste classique : la complexité temporelle de toutes les opérations sur un Deque est $O(1)$ alors que pour une liste classique (List) la complexité temporelle est en $O(n)$.

L'utilisation des Deque a considérablement réduit le temps d'exécution de notre algorithme.

Pour programmer notre application et proposer le meilleur rendu possible nous avons dans un premier temps pensé qu'il fallait calculer et afficher en parallèle. Cependant, calculer les positions de nos individus pendant que nous les affichions provoquait certaines erreurs et donnait parfois lieu à des ralentissements de l'image. Nous avons alors décidé de faire dans un premier temps tous les calculs nécessaires pour l'affichage et ensuite récupérer uniquement ce qu'il fallait pour afficher. Enfin, nous affichons dans notre interface le rendu.

L'interface a été réalisée dans le but de permettre à l'utilisateur de faire varier des paramètres et visualiser l'impact de ces changements. Elle est découpée en différents menus qui s'ouvrent et se ferment en fonction des choix de l'utilisateur.

Pour la demande d'information à l'utilisateur, nous avons d'abord pensé à des cases de saisie de texte sous la forme de JOptionPane simple. Cependant, face aux nombreuses possibilités d'erreur du type d'insertion d'information, nous sommes partis sur une interface plus directive et restrictive pour l'utilisateur. C'est ainsi que toutes nos données numériques apparaissent sous la forme de JSlider qui permettent non seulement de retourner uniquement des entiers mais aussi de fixer des limites hautes, basses mais surtout une valeur « défaut » à notre variable.

La création d'un JPanel cohérent s'est aussi révélée difficile. Après récupération du ContentPane général de notre JFrame, nous avons dû séparer l'espace pour y faire rentrer tous nos composants harmonieusement et sans conflit. Nous avons utilisé plusieurs Layouts pour arranger les JPanels entre eux.

L'empilement des différents JPanels créés dans des classes différentes pour simplifier le programme est devenue un désavantage au moment de faire passer les informations recueillies jusqu'à la classe de simulation.

Les outils tels que JDialogs ou JOptionPane ont rapidement été abandonnés au profit d'éléments plus simples telles que des JFrames avec des récupérations en continu des indicateurs dans des listes puis choix d'une variable définitive en fin de liste. Cette méthode n'est pas la plus optimisée mais était la plus simple à mettre en œuvre dans un temps restreint.

Nous avons rapidement appris à utiliser des ActionListeners dès la mise en place rudimentaire de notre affichage, cette fonctionnalité ne fut pas un problème majeur.

3 – CONCLUSION ET PERSPECTIVE

Ce projet a été très motivant pour toute l'équipe et appliquer les enseignements reçus s'est avéré être très plaisant. Il y a eu beaucoup de détermination de la part de toute l'équipe pour résoudre les problèmes rencontrés et nous sommes fiers du rendu.

Notre application est fonctionnelle, fidèle à la réalité, intuitive et respecte les consignes fixées.

Pour la suite nous pouvons utiliser d'autres méthodes pour l'esquive d'obstacles et pour acheminer les individus vers les issues de secours. Notamment en utilisant un algorithme du plus court chemin, optimisé et adapté pour la situation. L'algorithme de Dijkstra nous semblait trop lourd à mettre en place dès le début mais pourrait être intéressant.

L'affichage peut lui aussi être amélioré en proposant par exemple l'affichage en perspective cavalière du bâtiment et l'affichage en parallèle des différents étages. Cette tâche est assez complexe c'est pourquoi nous ne l'avons pas mis en place directement.

Enfin, à l'aide de données sur des évacuations nous pourrions améliorer notre simulation en essayant d'adapter les différents coefficients définissant notre modèle. Une sorte de méthode type Machine Learning pourrait beaucoup apporter à notre simulateur mais est très complexe.

- Sources -

<https://fr.wikipedia.org/wiki/Bousculade> Consulté le 07/05/2020 et la dernière modification a été faite le 6 mai 2020.

<https://www.youtube.com/watch?v=pOEcVIPchYY> Consulté le 20/04/2020 et publié le 9 décembre 2018.

<https://www.pnas.org/content/108/17/6884/> Article associé à la vidéo, consulté le 20/04/2020 et publié le 26 avril 2011. Les auteurs sont : Mehdi Moussaïd, Dirk Helbing et Guy Theraulaz. L'article est publié sur PNAS

<https://link.springer.com/article/10.1140/epjds7> Article sur les mouvements de foule, consulté le 20/04/2020 et publié en 2012. Publié sur SpringerLink. Les auteurs sont Drik Helbing et Pratik Mukerji.