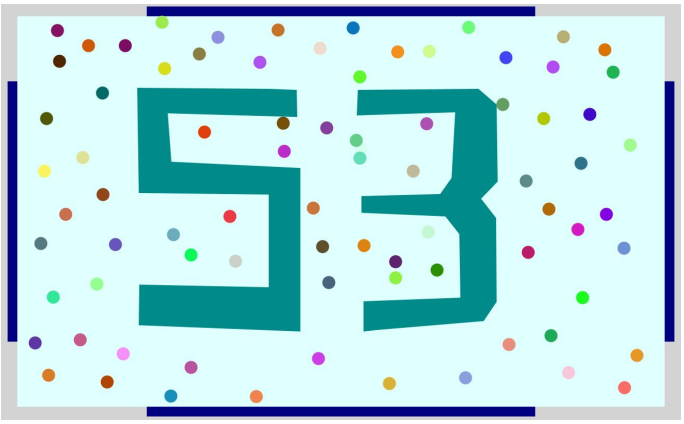
Rapport de projet

Simulateur de mouvements de foules



Réalisé par

CASTEL Florian
DA SILVA Joachim
PAYS Jordan
SARTORI Tom

Sous La direction de

BOUGERET Marin
POUPET Victor

Année Universitaire 2020-2021

Remerciements

Nous tenons, en premier lieu, à remercier l'équipe pédagogique de l'IUT Informatique Montpellier-Sète pour leurs enseignements.

Nous souhaitons particulièrement remercier monsieur M.Bougeret et monsieur V.Poupet, nos tuteurs, qui nous ont aiguillés et encouragés tout le long de ce projet. Nous les remercions également pour leur investissement dans ce dernier et pour le temps qu'ils y ont consacré.

Nous remercions aussi madame A.Messaoui pour les directives, la méthodologie, la relecture et plus généralement l'aide apportée afin de mener à bien la rédaction de ce rapport de stage.

Nous pensons également à remercier monsieur A.Marie-Jeanne pour son aide ponctuelle en mathématiques.

Sommaire

Table des figures

Glossaire

Checkbox :
Case à cocher, il s'agit d'un composant d'interfaces graphiques.

Slider :
Curseur de défilement, il s'agit d'un composant d'interfaces graphiques pour changer sélectionner une valeur à partir d'une échelle graduée définie.

IDE :
(Integrated Development Environment) Ensemble d'outils afin de concevoir un programme.

SDK :
(Software Development Kit) Kit de développement logiciel facilitant la programmation d'un logiciel en mettant à disponibilité divers outils.
Todo List : Liste de tâches à accomplir.

Introduction

Que cela concerne une évacuation, une manifestation, ou n'importe quel mouvement de foule, les infrastructures de notre société sont pensées, analysées et étudiées afin que ces déplacements collectifs engendrent un risque minimal. Pourtant, il est très difficile, voire impossible, d'expérimenter préalablement la réelle capacité d'un bâtiment à guider le plus efficacement possible les individus vers la sortie sans occasionner de risques majeurs. De plus, il semble très complexe de revoir le plan d'évacuation après la construction d'un bâtiment. Pour cela, lors de la conception d'une salle, il est primordial de savoir de quelle manière la population qu'elle va accueillir pourra s'en échapper. Afin de savoir si les agencements qu'elle contient sont envisageables, il est indispensable d'effectuer en amont, des simulations de différents scénarios d'évacuation.

De ce fait, il semble inévitable de se demander par quel procédé, un simulateur pourrait reproduire le plus naturellement possible le déplacement d'une foule dans un environnement donné.

Ce projet aura donc pour but de reproduire au plus proche de la réalité, les mouvements qu'un groupe d'individus auraient lors d'une évacuation. Pour illustrer cette simulation, une interface graphique minimaliste sera requise. De plus, l'utilisateur pourra interagir avec cette interface afin d'obtenir une représentation réaliste. A terme, il sera éventuellement possible de comprendre, prévoir et comparer des phénomènes naturels.

Dans un premier temps, nous mettrons au point l'ensemble des besoins fonctionnels et des spécifications dans le cahier des charges. Après cela, nous détaillerons le rapport technique présentant les différentes phases de la conception à la réalisation de ce projet. Par la suite, nous analyserons les résultats obtenus lors de l'utilisation de notre simulateur par le biais de différents tests unitaires. Pour parvenir à ces conclusions, nous avons dû mettre en place une méthode de gestion de projet efficace et adaptée, que nous justifierons.

1. Cahier des charges

1.1. Analyse du sujet et son contexte

Il existe des projets et des études similaires à notre projet tel que les articles [1] et [2]. Ces projets reprennent des concepts communs. On y retrouve une salle, des obstacles, une sortie et des personnes qui doivent sortir de la salle. Il y est pris en compte des collisions entre les personnes, les vitesses des personnes et des concepts géométriques au niveau des obstacles. Ces projets sont proches de notre projet aussi mathématiquement parlant, par exemple on utilise des vecteurs et des calculs géométriques pour les obstacles.

Il existe des études comme les articles [3] et [4], sur la même idée de notre sujet, des projets sur le thème des mouvements de foules. Ceux-ci proposent des fonctionnalités plus poussées. Le point de vue se situe plus autour des personnes/groupes de personnes dans un espace ouvert, comment vont ils se croiser, quelle trajectoire prendre au contact de deux groupes qui ont un objectif non commun, comment le champ de vision d'une personne impacte son déplacement.

.

Dans l'article [5] qui suit on parle de mouvement de foule dans le jeu assassin's creed qui prend place à paris durant la révolution française. Le concept de simulation de mouvement de foule est utilisé pour faire une foule virtuelle qui réagirait correctement aux actions du joueur ou de l'environnement autour de la foule.

Le logiciel proposé peut être utilisé dans une entreprise de construction de bâtiments/architecte pour définir une solution efficace et fiable sur la manière d'évacuer un bâtiment, un endroit par une ou plusieurs sorties.
Il pourrait éventuellement être aussi adapté pour être utilisé par une entreprise qui souhaite développer un jeu par exemple, les directions des personnes dans des salles peuvent servir à définir un chemin à suivre fiable et direct. De plus, la gestion de collision et la grande capacité de personnes dans la simulation peuvent permettre des projets ludiques plus ambitieux.

1.2. Analyse des besoins fonctionnels

L'utilisateur doit disposer de diverses possibilités afin d'interagir avec l'outil et de créer la simulation de son choix.

La figure 1.1 représente un diagramme de cas d'utilisations mettant en avant les différentes fonctionnalités de notre programme, expliquées après le diagramme.

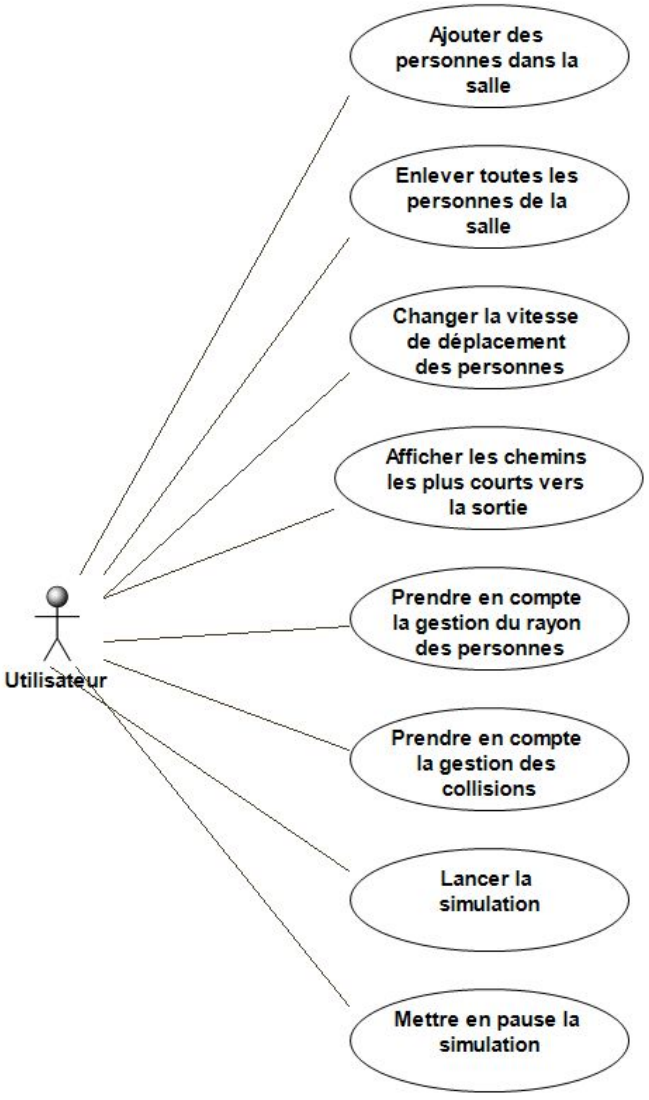


Figure 1.1 : Diagramme de cas d'utilisations

Premièrement, l'utilisateur doit avoir la possibilité d'ajouter des personnes dans la salle. Ensuite, l'utilisateur doit aussi pouvoir choisir de prendre en compte certains paramètres tels que : la gestion des rayons ou encore la gestion des collisions.

Par ailleurs, il doit aussi avoir accès à d'autres interactions pour simplifier la visualisation des résultats tels que l'affichage des chemins les plus rapides vers la sortie ou encore la modification de la vitesse des personnes.
Pour finir, il y aura différents boutons à sa disposition, afin de pouvoir contrôler l'état de la simulation c'est-à-dire :

- lancer la simulation
- mettre en pause la simulation
- stopper et enlever toutes les personnes dans la salle

1.3. Analyse des besoins non-fonctionnels

Afin de mener à bien ce projet, nos tuteurs souhaitent nous laisser de grandes libertés afin que nous puissions par nous-mêmes, découvrir un grand nombre de contraintes au fur et à mesure de l'avancement du projet, et après, nous aider à les résoudre.

Cependant, nous avons tout de même certaines indications à suivre pour mener à bien ce projet. De manière générale, il est nécessaire de mettre en place une interface graphique, même minimaliste. Aussi, étant donné que l'objectif du projet est de faire une simulation, il faut donc un résultat au plus proche de la réalité.

Lorsqu'on fait une simulation d'un important nombre de personnes, il est nécessaire d'avoir un résultat toujours fluide. De ce fait, il est primordial de prendre en compte des éventuels problèmes d'optimisation.

2. Rapport technique

Dans cette partie technique, on s'intéresse à toutes les phases de développement du programme. Les sections 2.1. Pré-calcul de la salle à 2.5. Gestion de plusieurs personnes mettent en œuvre en détail les différents aspects algorithmiques primordiaux de ce projet et en 2.6. Algorithme principal,, il est expliqué comment on utilise ces derniers. Par la suite, en 2.7. Structure de données, il est évoqué la manière dont ces algorithmes ont été implémentés. Enfin, la section 2.8. Graphique, associe physique et affichage dynamique.

2.1. Pré-calcul de la salle

2.1.1. Obstacles et sorties

Tout d'abord, sans prendre en compte les personnes, une salle est composée d'obstacles et de sorties. Aussi, pour se déplacer entre deux points quelconques, en évitant les obstacles et en empruntant le plus court chemin, il est nécessaire de se déplacer uniquement vers les sommets des obstacles. Si on se déplace entre deux coins d'un même obstacle, alors on se déplace sur son arête. De ce fait, on peut définir un obstacle uniquement par ses sommets.

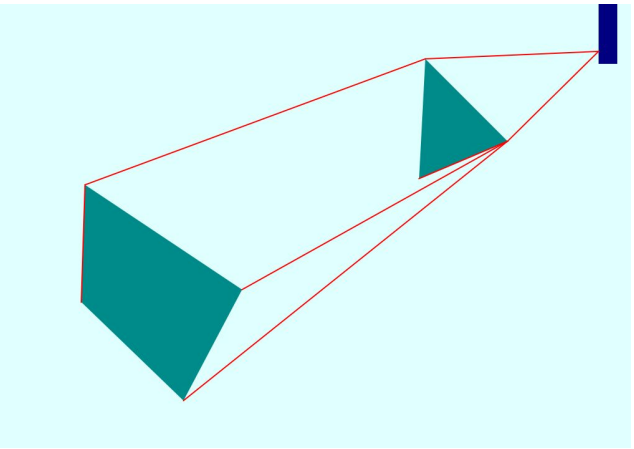
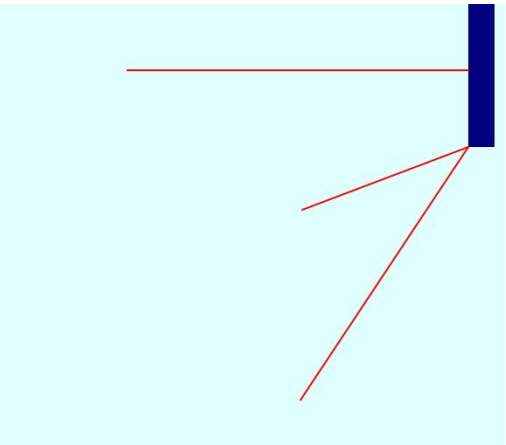


Figure 2.1: Chemin entre sommets

Figure 2.2: Chemin vers la sortie



Les sorties sont définies par des segments disposés sur les bords d'une salle. Ainsi, si on part d'un point A quelconque et qu'on souhaite aller à la sortie en prenant le plus court chemin, il existe donc un unique point B, contenu dans le segment de la sortie.
De plus, si on prend plusieurs points A différents, on peut obtenir plusieurs fois le même point B. En outre, chaque A possède un et un seul B alors qu'un B peut posséder plusieurs A. Les points A représentent donc les coins d'obstacles ayant un accès direct à une sortie. Remarque : Dans le cas où une salle possède plusieurs sorties, le segment

le plus court entre un coin d'obstacle et un point du segment d'une sortie, n'est pas forcément le chemin le plus court entre ce point et le point d'une autre sortie.

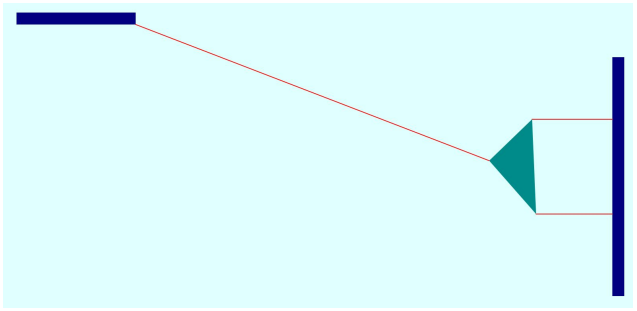


Figure 2.3: Différents chemin vers des sorties

Ci-dessous, le code de la fonction findPointSortieDirect, servant à déterminer, pour un point de départ et une sortie donnée, quelle point de cette même sortie est le plus proche. Une sortie est toujours définie par un segment [extremum1 ; extremum2]. Le point extremum1 est toujours celui qui a le plus petit x si sur le mur du haut, ou sinon, le plus petit y si sur un mur latéral. Dans la boucle, on teste donc pour tous les points entre les deux extremums, lequel est le plus proche du point en paramètre, sans intersecter un obstacle (voir : 2.3.2 Intersection entre une personne et un obstacle).

```
// Cette fonction est dans la classe Sortie.
// Une sortie est définie par les bords de son segment, extremum1 et
extremum2.
// Pour un point de départ en paramètre, retourne un point correspondant
à l'endroit de la sortie (this),  le plus proche et direct par rapport
au départ.
// S'il n'y a pas de point direct, alors return null.

public Point findPointSortieDirect (Salle salle,  Point depart) {
   // Extrémité du segment de la sortie avec le plus petit x ou y
suivant orientation.
   Point courant = new Point(extremum1);
   Point pointSortie = null;
   double plusPetiteDistance = Double.POSITIVE_INFINITY;

   // Pour tous les points du segment de la sortie.
   for (double i = 0; i <= getLargeurPorte(); i++ ) {
       // Si la sortie est sur le mur du haut ou du bas.
       if (estMurHaut() || estMurBas()) {
           // courant prend tous les points du segment de la sortie.
           courant.setX(extremum1.getX() + i);
           // Si le segment [depart ; courant] n'intersecte aucun
obstacle de la salle.
           if (!salle.intersecObstacle(depart, courant)) {
```

```
                if (MathsCalcule.distance(depart, courant) <=
plusPetiteDistance) {
                    // On cherche la distance minimale de départ à
courant.
                    plusPetiteDistance = MathsCalcule.distance(depart,
courant);
                    pointSortie = new Point(courant);
                }
            }
        }
        else { // Est donc sur le mur de droite ou de gauche.
            // Même fonctionnement mais on modifie le y du point courant.
        }
    }
    return pointSortie; // Retourne null si pas de point direct.
}
```

Maintenant, on sait donc que chaque obstacle est défini uniquement par ses sommets et aussi, qu'une sortie ne l'est que par certains points de son segment. Après ça, pour une salle, on cherche à déterminer pour chaque sortie, tous les points "utiles". Pour ce faire, on crée une liste de points utiles. Pour remplir cette liste, pour chaque sommet d'obstacle de la salle, on détermine le point de sortie direct le plus proche et on ajoute donc ce point à la liste.

A ce stade, on a donc une liste de tous les points de sorties possibles et une autre liste contenant tous les sommets des obstacles. Pour rappel, on ne prend toujours pas en compte les personnes.

2.1.2. Graphe

Pour cette partie, comme expliqué précédemment, on part du principe qu'on a une liste de points obstacles et une autre liste de points sorties. Dès lors, on peut représenter tous ces points dans un graphe. Par exemple, sur la figure ci-dessous, on peut voir que sur les trois sorties de la salle, huit points ont été calculés à partir des différents sommets de la salle, grâce à la fonction findPointSortieDirect(). Ainsi, à partir de ces huit points de sorties et de ces quatorze sommets d'obstacles, sans prendre en compte les collisions avec les obstacles, on obtient le graphe ci-dessous.

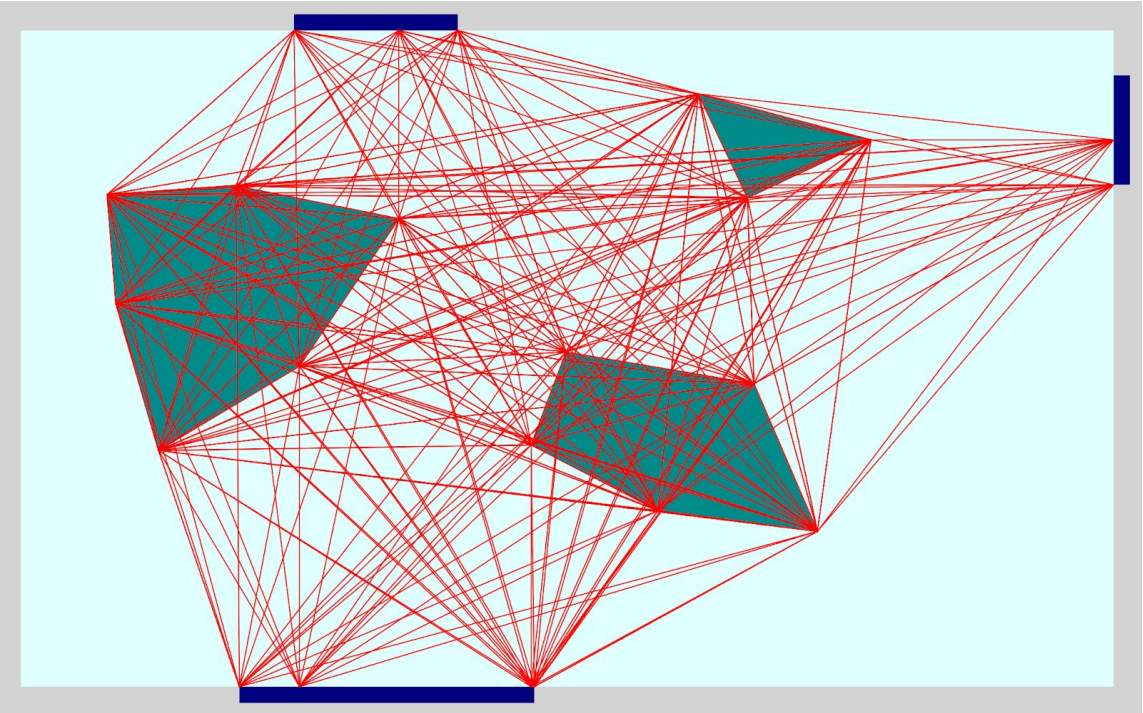


Figure 2.4. : Représentation d'un graphe à partir d'une liste de sommets d'obstacles et une liste de points de sorties calculée préalablement.

Par la suite, on ne prendra plus en compte les différentes arêtes internes aux obstacles. Toute la partie technique de ce calcul est expliquée dans la section 2.3. Géométrie. De plus, voir exemple ci-dessous.

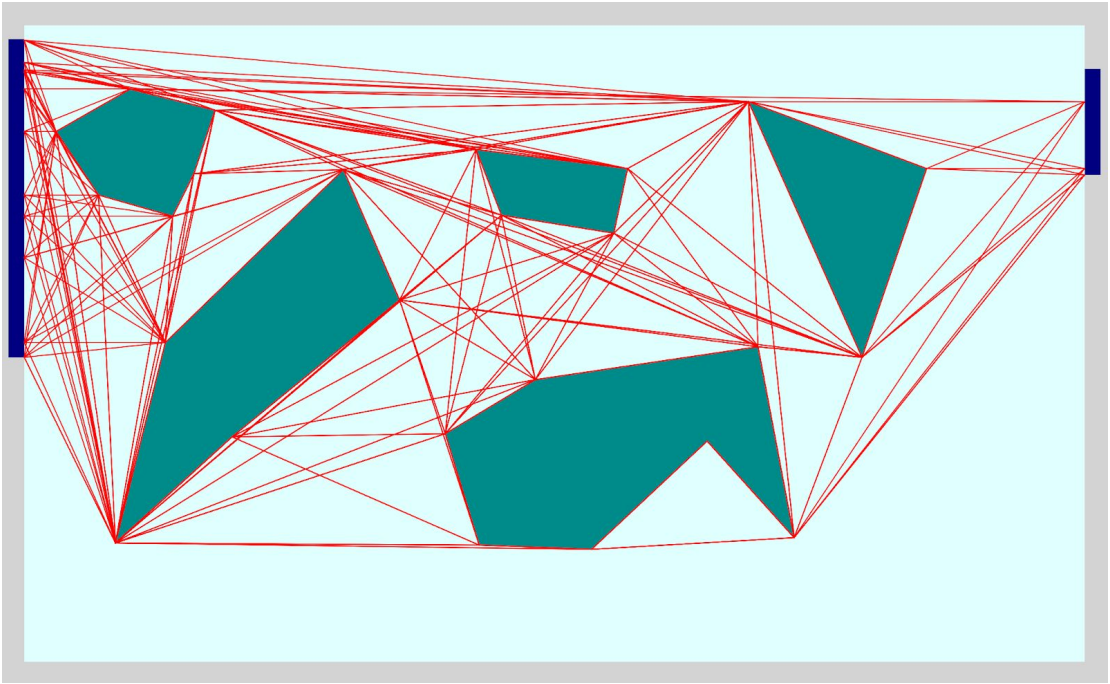


Figure 2.5. : Représentation du graphe avec la gestion des intersections des obstacles.

Avec ce graphe, on souhaite maintenant déterminer tous les plus courts chemins pour arriver aux sorties, à partir de chaque sommet d'obstacle. Pour ce faire, on utilise l'algorithme de Dijkstra (Annexe 2 : Algorithme de Dijkstra) à partir d'une sortie, sans prendre en compte les autres points de sorties et surtout, sans prendre en compte les chemins internes des obstacles. Pour chaque point d'obstacle, on connait donc maintenant quel est son prochain point du plus court chemin vers la sortie, ainsi que la distance jusqu'à cette dernière (grâce à Dijkstra).

Après, on stocke ces valeurs en mémoire, puis on effectue la même opération avec le point de sortie suivant dans la liste. On compare les nouvelles distances pour chaque point, et si cette dernière est inférieure à l'ancienne, alors pour ce point, on met à jour la mémoire du point suivant et de la distance.
Pour finir, on effectue cette même opération pour tous les points de sorties et on obtient donc pour chaque point obstacle, le plus court chemin pour arriver à la sortie, en prenant en compte les autres obstacles et les différentes sorties. Sur l'exemple ci-dessous, voici donc pour chaque sommet d'un obstacle, son plus court chemin vers une sortie.

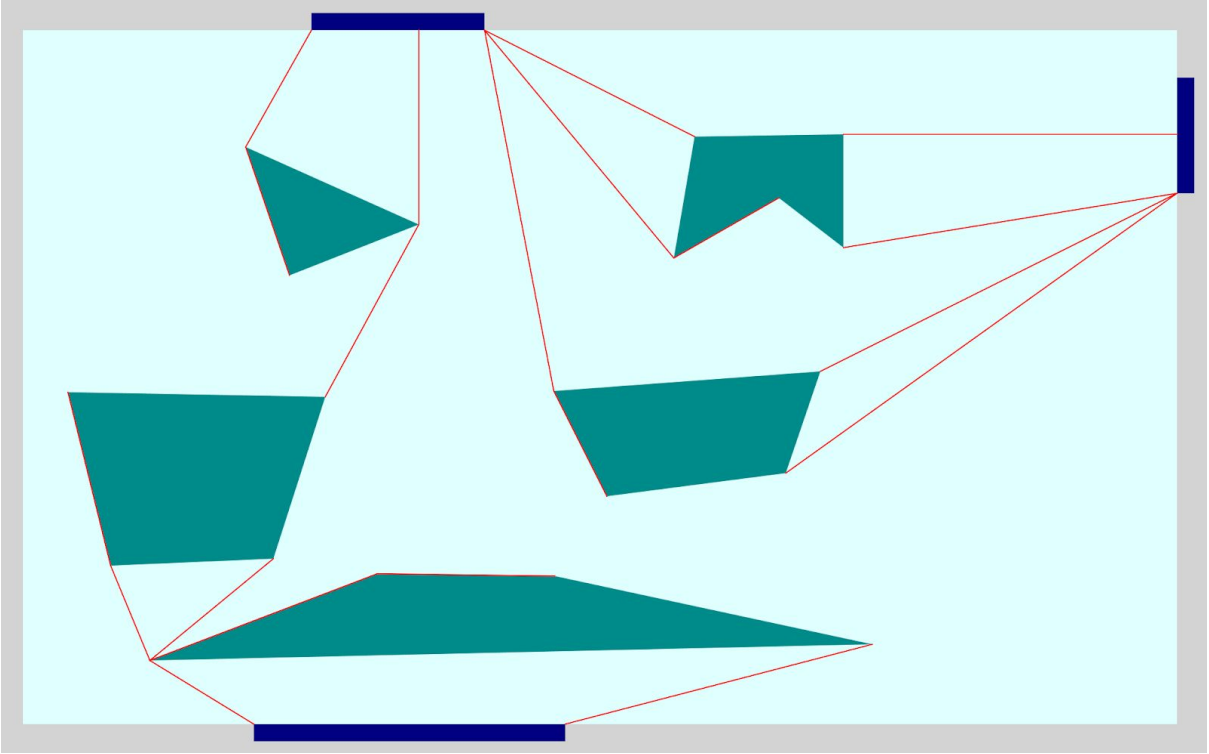


Figure 2.6. : Représentation des plus courts chemins vers une sortie de chaque sommet d'obstacle.

2.2. Trajectoire d’une personne réduite à un point

2.2.1. Chemin d’une personne

Dans cette partie, on s'intéresse au comportement d’une personne placée à un endroit quelconque, souhaitant se déplacer le plus rapidement possible vers la sortie en évitant les obstacles. Pour savoir si un obstacle barre la route, il faut utiliser la fonction expliquée dans la section 2.3.2. On ne prend donc pas en compte les potentiels autres personnes de la salle. Aussi, on ne définit une personne que par un point étant le centre du cercle graphique qui la représente. De plus, on part du principe que pour chaque sommet d’obstacle de la salle, on connaît le point suivant vers où aller, pour prendre le plus court chemin vers la sortie (cf : 2.1.2. Graphe). Aussi, pour chaque sommet, on connaît la distance totale à parcourir pour atteindre la sortie. On est donc dans la disposition suivante :

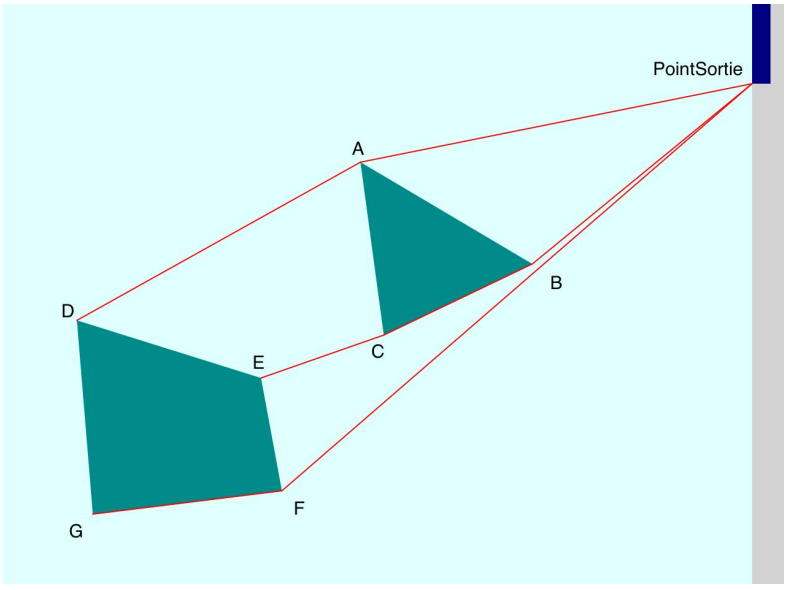


Figure 2.7 : Représentation du graphe du plus court chemin.

Avec :

A(681.0 ; 205.0) | Distance sortie : 325.36 px ; Point suivant : PointSortie(1000.0 ; 141.0)
B(821.0 ; 288.0) | Distance sortie : 231.62 px ; Point suivant : PointSortie(1000.0 ; 141.0)
C(700.0 ; 346.0) | Distance sortie : 365.81 px ; Point suivant : Point B(821.0 ; 288.0)
D(450.0 ; 334.0) | Distance sortie : 589.94 px ; Point suivant : Point A(681.0 ; 205.0)
E(600.0 ; 381.0) | Distance sortie : 471.76 px ; Point suivant : Point C(700.0 ; 346.0)
F(617.0 ; 473.0) | Distance sortie : 506.87 px ; Point suivant : PointSortie(1000.0 ; 141.0)
G(463.0 ; 492.0) | Distance sortie : 662.03 px ; Point suivant : Point F(617.0 ; 473.0)

Par la suite, lorsqu'on souhaite calculer le plus court chemin d'une personne, il y a deux cas possibles suivant le placement de cette dernière :

- Son plus court chemin est de se diriger directement vers un point d'une sortie, sans passer par aucun obstacle.
- Son plus court chemin passe par plusieurs obstacles.

Pour ce faire, dans un premier temps, on détermine, si c'est possible, quelle sortie directement accessible est la plus proche du personnage. Alors, on utilise le même algorithme que pour les sommets d'obstacles. C'est-à-dire, qu'on détermine un point sur n'importe quelle sortie de la salle, tel que, le segment entre la personne et ce point soit le plus petit. Evidemment, il faut que ce segment soit direct et donc ne passe pas par un obstacle.
Remarque : il est possible que la personne n'ait aucun accès direct à une sortie.

Dans un second temps, on détermine quel chemin (passant par des sommets d'obstacles), est le plus court pour arriver à une sortie. Pour cela, grâce à des calculs expliqués à la partie 2.3.2, on détermine tous les sommets d'obstacles que la personne peut atteindre directement. Ensuite, pour chacun, on fait le calcul : (distance entre cette personne et le point d'obstacle) + (distance stockée dans ce point obstacle pour atteindre la sortie), puis, on prend la valeur minimale. Pour rappel, la distance à la sortie du point obstacle est calculée une seule fois (puis stockée dans l'objet), lors du calcul du graphe et n'est donc pas recalculée pour chaque personne.

Pour finir, on prend la distance minimale des deux valeurs calculées précédemment et on obtient donc le point correspondant. Si la première distance n'existe pas car la personne n'a pas de sortie directement accessible, alors on prend le point correspondant calculé dans la deuxième partie. De cette manière, on obtient donc le premier point vers lequel la personne doit se déplacer (point obstacle ou point sortie). Par la suite et dans le code, on appellera ce point "objectif" de la personne.

2.2.2. Déplacement

Soit un point $A(x_A ; y_A)$, définissant la position d'une personne, et un point $B(x_B ; y_B)$, définissant les coordonnées de l'endroit vers lequel on souhaite se déplacer, qu'on appelle "objectif". Ainsi, pour que le point A se déplace jusqu'au point B, on calcule le vecteur $\vec{u} = (x_B - x_A ; y_B - y_A)$ et on modifie A tel que $A = A + \vec{u}$.

Cependant, dans notre cas, chaque personne ne peut se déplacer que d'une distance prédéfinie, qui correspond à sa vitesse. De ce fait, il est nécessaire de normaliser le vecteur de déplacement afin qu'il fasse la taille souhaitée. Pour ce faire :

- Soit v, un réel positif définissant la taille souhaitée.
- Soit $\vec{u} = (x ; y)$, le vecteur de déplacement.
- Soit $\vec{u'}$, le vecteur $\vec{u}$, normalisé à la taille v.

Ainsi :

$$arg = \sqrt{(x^2 + y^2)}$$
$$\vec{u'} = \left(\frac{v}{arg} x \; ; \; \frac{v}{arg} y \right)$$

```
// Normalise dx dy par rapport à la vitesse de la personne.
public void setDxDyNormalise (Point pointObjectif) {
   double dx = getDx(pointObjectif);
   double dy = getDy(pointObjectif);

   //argument = sqrt(x^2 + y^2)
   double argument = Math.sqrt( (dx * dx) + (dy * dy) );

   this.dx = (vitesse/argument) * dx;
   this.dy = (vitesse/argument) * dy;
}
```

De par ces calculs, nous pouvons maintenant déplacer une personne sur une droite continue partant de A, vers un point B, en modifiant ses coordonnées toutes les x secondes. Néanmoins, on souhaite que la personne s'arrête précisément au point B, or, la somme de ces déplacements n'est jamais précisément égale à la distance entre A et B. Ainsi, avant chaque déplacement, il faut vérifier qu'il n'engendre pas de dépassement du point B, dans le sens du vecteur directeur de la droite. Dans le cas où il dépasserait, les coordonnées de la personne deviennent celles de son objectif, le point B.

Soit :

- A (x_A ; y_A), les coordonnées courant de la personne.
- B (x_B ; y_B), les coordonnées de son objectif.
- $\vec{u}$ (a ; b), le vecteur normalisé de déplacement de la personne.

Ainsi :

$$Si \; ((|a| > \left| x_A - x_B \right|) \; ou \; (|b| > \left| y_A - y_B \right|))$$
$$alors \; A \leftarrow B$$
$$Sinon$$
$$A \leftarrow A + \vec{u}$$

C'est donc de cette manière que qu'est géré le déplacement des personnes.

2.2.3. Récapitulatif du déplacement

On détermine donc le premier objectif de la personne. Ensuite, on calcul le vecteur de déplacement normalisé par la vitesse de la personne vers cet objectif. On déplace la personne en vérifiant si elle a atteint ou non son objectif. Lorsqu'elle l'atteint, soit elle est arrivée à la sortie et tout est fini, soit elle atteint un sommet d'obstacle et donc, on modifie son objectif pour le point suivant du point obstacle, calculé préalablement dans le graphe de plus court chemin. On répète donc cette opération jusqu'à ce que la personne atteigne un point de sortie (voir Figure 2.8.)

Pour réitérer cette dernière, toutes les vingt millisecondes, l'opération se relance automatiquement et ainsi, le résultat peut s'actualiser frame par frame.

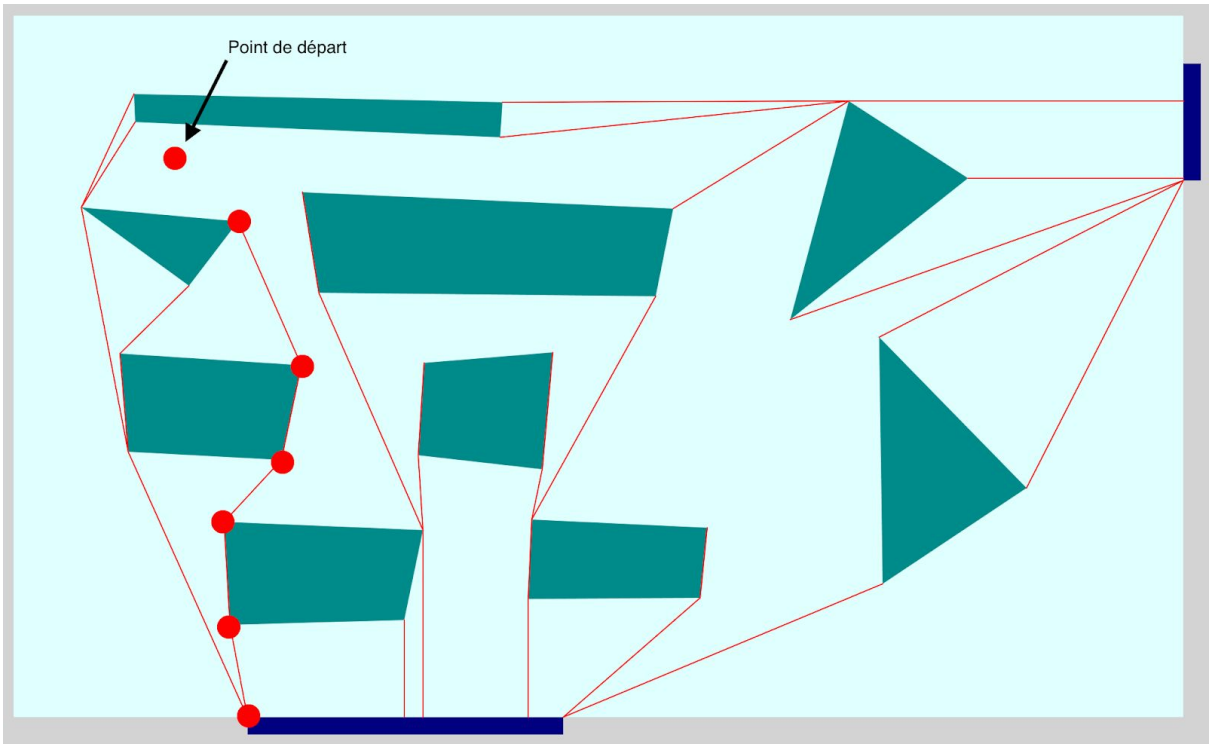


Figure 2.8 : Représentation du plus court chemin emprunté par une personne.

2.3. Géométrie

2.3.1. Intersection de deux segments

Pour calculer les valeurs des arêtes du graphe, on avait besoin de savoir pour chaque sommet d'un obstacle s' il avait un chemin direct vers une sortie ou si au contraire, il avait des obstacles sur son chemin. Pour cela nous avons dû réfléchir à un algorithme qui permet de savoir si 2 segments sont sécants.

Soit un point A(x_A ; y_A) et un point B(x_B ; y_B) représentant un segment. Un point C(x_C ; y_C) et D(x_D ; y_D) correspondant à un autre segment.

Pour faciliter la compréhension du code vous pouvez trouver en Annexe 1 le système mathématique qu'on a utilisé pour faire ce programme.

```
// Cette fonction est une fusion clarifier de 2 fonctions séparées
estCoupe(...) et determinant(...) de notre code. return true si le
segment AB créé à partir du point A et du point B est sécant au segment
CD créé à partir du point C et du point D
public static boolean estSecante(Point A, Point B, Point C, Point D){
double a = B.getX() - A.getX();
double b = B.getY() - A.getY();
double c = C.getX() - D.getX();
double d = C.getY() - D.getY();
double u = C.getX() - A.getX();
double v = C.getY() - A.getY();
if((a*d - b*c) != 0){
   double mat1 = (1 / determinant) * d;
   double mat2 = (1 / determinant) * (-b);
   double mat3 = (1 / determinant) * (-c);
   double mat4 = (1 / determinant) * a;
   double k1 = (mat1 * u) + (mat3 * v);
   double k2 = (mat2 * u) + (mat4 * v);
   return k1>0 && k1<1 & k2>0 && k2<1;
} else return false;
}
```

Comme vous avez pu le remarquer le code ne suit pas exactement le système mathématique car d'après le système, k (qui est représenté par K1 dans le code) et k' (qui est représenté par K2 dans le code) doivent appartenir à l'intervalle [0;1].
Or, nous avons décidé de considérer que si les segments ne se coupent pas complètement (figure ci-dessous), alors ils ne sont pas sécants, d'où le strictement inférieur et supérieur pour K1 et K2.

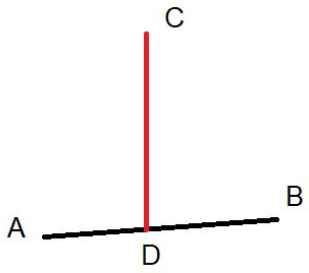


Figure 2.9: Segments qui se coupent mais qui ne sont pas sécants.

2.3.2 Intersection entre une personne et un obstacle

On va maintenant définir la primitive qui ont été utilisées dans la section du graphe 2.1 par exemple. Lorsqu'on l'utilise pour le déplacement d'une personne. Donc, dans la classe Personne, on appelle la fonction primitive suivante, avec le Point coordA qui représente le point de départ d'une personne et le Point coordB, qui représente son point d'arrivée. Ces points forment donc un segment.

Pour la suite, on considère "diagonale" d'un obstacle, l'ensemble des segments, de sommet à sommet qui ne sont pas voisins, internes à cet obstacle.

```
//Cette fonction return true si le segment créé par les points coordA et
coordB mis en argument sont sécants à l'une des arêtes de l'obstacle qui est
lui aussi en argument. Pour cela il utilise la fonction estSecante(...).
public static boolean coordSegments(Point coordA,Point coordB,Obstacle o)
```

Dans l'exemple ci-contre, notre fonction est appelée avec en argument les points coordA et coordB ainsi que l'obstacle ABCD.
Elle utilise ensuite la fonction estSecante(...) avec chaque arête de l'obstacle, dès qu'il trouve qu'une arête est sécante alors il return true pour exemple une fois arrivé à l'arête CD, on a donc l'appel de la fonction estSecante(coordA, coordB, CoordC, coordD) avec coordC qui est les coordonnées du point C et coordD qui est les coordonnées du point du D. Si ces deux segments sont bien sécants alors la fonction coordSegments(...) return true.Si elle ne trouve aucune arête sécante alors elle return false.

Cependant un problème c'est vite montré. En effet les diagonales que formait les sommets n'était pas vu comme sécante à cause de la simplification qu'on a fait en décidant qu'ils fallaient que les segments soient complètement coupés pour les considérer sécants.
Donc on regarde aussi au début du programme si le segment coordAcoordB est superposé à l'une des diagonales de l'obstacles. Dans ce cas la fonction return aussi true.

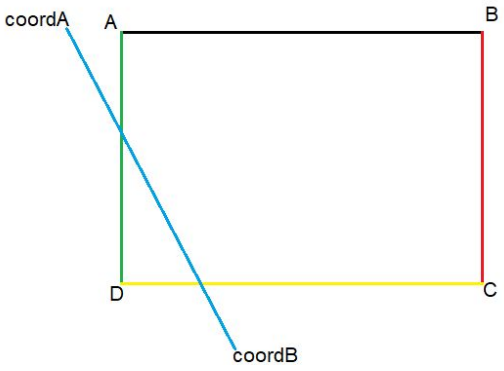


Figure 2.10 : Représentation d'une personne qui traverse un obstacle.

2.4. Gestion du rayon des personnes

Dans cette section, on prend en compte la partie graphique d'une personne. En effet, une personne est définie par son centre, mais est graphiquement représentée par un cercle de rayon donné. Le but est donc de faire déplacer les personnes en leur faisant frôler les obstacles par rapport à leur bord. Dans cette optique, plusieurs méthodes ont été testées et sont expliquées ci-dessous.

2.4.1. V1 : Déplacement des objectifs

Pour cette version, on souhaite que chaque personne puisse avoir un rayon bien particulier. Ainsi, pour un objectif donné, on souhaite calculer le vecteur de déplacement de la personne vers un autre objectif fictif correspondant au sommet de l'obstacle, déplacé de la taille du rayon, dans la bonne orientation.
Cependant on rencontrait des problèmes sur des obstacles qui étaient alignés car, lorsqu'on a plusieurs objectifs potentiels alignés, on considère uniquement le plus éloigné. Comme le montre le schéma ci-dessous, la personne(le rond rouge) se décale bien pour frôler le premier obstacle par rapport à son rayon. Mais rentre dans le deuxième obstacle car comme expliqué plus haut, son objectif qui est dans ce cas là la sortie, ne change pas et ne permet donc pas de frôler l'obstacle du haut.

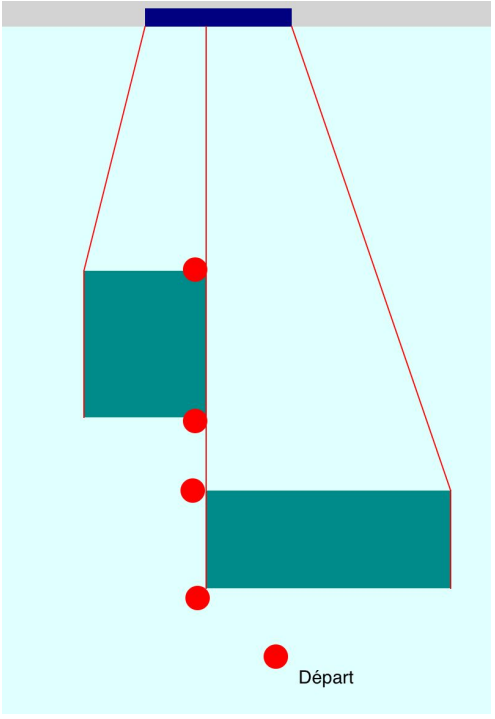


Figure 2.11 : V1 du déplacement d'une personne par rapport à son rayon.

En accord avec nos tuteurs, on peut partir du principe que toutes les personnes d'une même salle ont le même rayon. De ce fait, il est possible que pour un obstacle donné, on augmente sa taille, dans le calcul du graphe, de la taille du rayon.

2.4.2. V2. Agrandissement des obstacles physiques.

Dans cette section, on sépare donc les points physiques d'un obstacle, qui servent au calcul du graphe, et les points graphiques du graphe, qui servent à l'affichage.
Soit AB un segment représentant l'arête d'un obstacle et BC un autre segment représentant une autre arête de cet même obstacle. Ainsi, on calcule A' et B' tel que AB parallèle à A'B' avec un espacement de la taille du rayon des personnes de la salle.
Soit $\vec{u}$, le vecteur directeur du segment AB.
Ainsi, soit $\vec{v}$ son vecteur orthogonal tel que $\vec{v} = (-y_{\vec{u}} ; x_{\vec{u}})$. Ensuite, on normalise $\vec{v}$ à la taille du rayon.
Puis, $A' = A + \vec{v}$

On fait la même chose pour tous les sommets de l'obstacle puis, on détermine les équations des droites des nouvelles arêtes. Enfin, les nouveaux points physiques deviennent les intersections de ces droites (voir la figure ci-dessous).

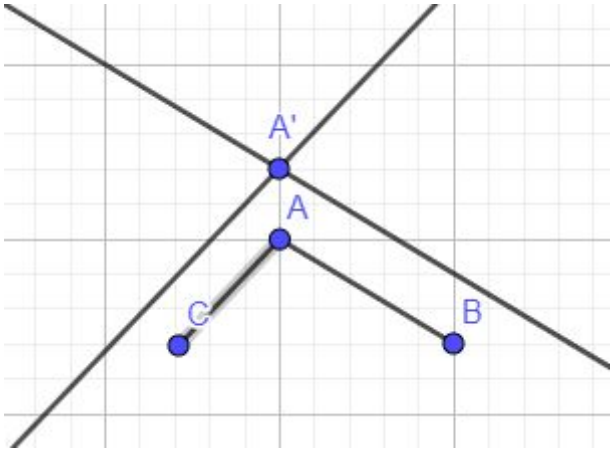


Figure 2.12. : Représentation de l'agrandissement d'un coin d'un obstacle ABC.

Cependant, cette technique à quelques limites. En effet, cette dernière force l'utilisateur à ajouter les sommets d'un obstacle qu'il souhaite créer dans le sens des aiguilles d'une montre. Aussi, dans le cas de sommets à angles très faibles, l'intersection des parallèles est finalement assez loin du sommet graphique. Cette technique fonctionne mais n'est pas totalement satisfaisante.

2.5. Gestion de plusieurs personnes

Jusqu'à présent, on prend chaque personne en compte indépendamment des autres et donc, elles peuvent se "marcher dessus". Dans cette partie, on verra donc comment sont gérées les collisions afin que le simulateur soit plus réaliste.

2.5.1. Collisions V1

Dans cette première version basique, on souhaite juste que les personnes s'arrêtent lorsqu'il y a une collision. Ainsi, dans la boucle principale qui à chaque frame, tous les x ms, modifie les coordonnées de toutes les personnes de la salle, on ajoute simplement une condition afin que le déplacement soit annulé s'il y a collision.
Pour ce faire, lorsqu'une Personne p doit se déplacer, on teste avec toutes les autres personnes de la salle, que la fonction estEnCollision, ne renvoie pas vraie.

```
// Calcule la distance entre les personnes p et compare grâce à la
fonction MathsCalcule.distance(). Si la distance entre les deux
personnes est inférieure ou égale à deux fois le rayon d'une personne,
renvoie vrai car il y a collision.
public boolean estEnCollision(Personne p, Personne compare){
   return MathsCalcule.distance(p.getProchainMouvement(),
compare.getCoordCourant()) <= (p.getRayon() * 2);
}
```

Remarque : pour la personne p, on compare avec les coordonnées qu'elle pourrait avoir après son prochain déplacement. Ainsi, on arrête de déplacer la personne avant qu'elle ne "marche" sur une autre graphiquement.

2.5.2. Collisions V2

Dans cette seconde partie, on introduit la notion de distance sociale. Cette distance correspond à une augmentation invisible du rayon des personnes dans laquelle, elles évitent d'entrer en collision. En outre, les personnes ne collent pas lorsqu'elles se déplacent ; elles respectent chacune leurs propres "rayons sociaux".
Dès lors, on commence par détecter si les personnes sont en collision non plus avec leur rayon normal, mais avec leur rayon agrandi (l'augmentation n'est pas graphique). Ainsi, lorsque deux personnes entrent en collisions dans leur rayon, on fait avancer la plus proche des deux vers la sortie. Ci-après, le pseudo-code explicatif.

Etant donné le rayon social R' et le rayon classique R, avec R' >= R

A chaque frame:

pour chaque personne p:
soit c = coordonné du point d'arrivé du prochain déplacement prévu de p

soit X_{R'} = ensemble des personnes qui seraient intersectées (au sens de R') si p allait en c

soit X_R = ensemble des personnes qui seraient intersectées (au sens de R) si p allait en c

soit estPlusProche = true ssi p est la personne la plus proche (dans sa pos actuelle) (parmi les personnes de X_{R'}) de sa sortie

si ((X_{R'} est vide) OU (estPlusProche ET X_R est vide))

faire le mouvement de p

Fin

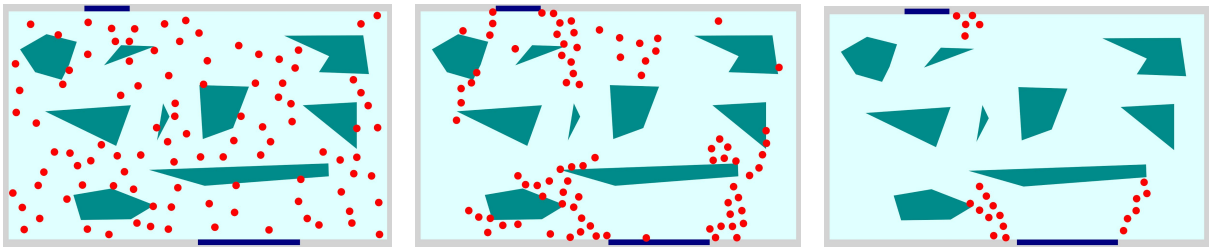


Figure 2.13. : Représentation du désengorgement d'une salle collisions entre les personnes.

Remarque : on peut voir sur la figure ci-dessus, que les personnes gardent toujours une certaine distance "sociale" entre elles.

Cependant, un problème se pose quant à la taille de cette distance sociale. En effet, si cette dernière est inférieure au rayon classique, beaucoup de collisions ne sont pas évitées. Ceci semble logique, car si toutes les personnes s'évitent avec chacunes un rayon ou plus, comme on prend en compte les collisions avec le rayon social, cela signifie qu'entre chaque personne, il y a au minimum, l'espacement d'un diamètre. De ce fait, dans la très grande majorité des cas, il y a toujours au minimum la place pour laisser passer une personne. Cependant, il est possible, dans certains cas très particuliers et ainsi très peu proches de la réalité, que cette technique ne soit pas salvatrice et donc que deux personnes se bloquent. Ces cas étant très rares, cette version de la gestion des collisions s'avère très satisfaisante.

2.5.3. Collisions V3

Cette troisième version n'a pas été implémentée et est uniquement à titre indicatif pour de futures évolutions.

Dans les versions précédentes, on ne modifie pas la trajectoire des personnes. Ainsi, on peut imaginer une modification de ce dernier afin que les personnes s'évitent tout en gardant la même vitesse. Aussi, on peut imaginer que lorsque plusieurs personnes ont le même objectif et que le "trafic" est engorgé, alors on les fait se déplacer sur un itinéraire bis.

2.6. Algorithme principal

Une fois, les différentes méthodes implémentées, on souhaite les appeler pour faire fonctionner l'algorithme principal.

2.6.1. Etapes clés du programme

Pour mettre en lien toutes les fonctions, on crée une méthode pour le démarrage de la simulation afin de générer à l'avance le chemin de chaque Personne dans la Salle.
Tout d'abord, on initialise le graphe des chemins les plus courts dans la Salle (voir partie 2.1.2).
Ensuite, pour chaque personne, on définit son point d'objectif et on normalise sa variation de trajectoire en fonction de sa vitesse et de son objectif.
Pour finir, avec la méthode de démarrage, on répète une nouvelle fonction toutes les vingt millisecondes permettant le déplacement des Personne en prenant en compte les collisions (voir partie 2.5.2). Lorsqu'elles atteignent leurs objectifs, alors on recalcule le vecteur de déplacement avec l'objectif suivant.
Enfin, lorsqu'une personne atteint une sortie, on la retire de la boucle principale et on la désaffiche.

2.7. Structure de données

2.7.1. Implémentations majeurs

Les implémentations des précédents algorithmes sont réalisées dans les classes ayant les structures décrites dans la partie 2.7.2 et 2.7.3.
Premièrement, la méthode findPointSortieDirect() est située dans la classe Sortie.
La création de graphe de tous les chemins ou des chemins les plus courts (voir partie 2.1.2) sont présents dans la classe Graphe.
Par la suite, les fonctions afin de gérer le déplacement d'une personne (voir partie 2.2.2) sont stockées dans une classe nommée Personne.
Les opérations géométriques (voir partie 2.3) n'appartiennent à aucun objet de notre salle et sont donc répertoriées dans une classe à part entière qui est MathsCalcule.
Ensuite, la gestion du rayon est implémentée lors de la création d'un Obstacle dans son constructeur. (voir partie 2.4.2)
Par ailleurs, la gestion des collisions se fait directement dans la Salle entre plusieurs individus. (voir partie 2.5.2)

2.7.2. Données triviales

Pour pouvoir mettre en place l'algorithme principale il faut structurer les classes triviales de la manière suivante : (voir Figure 2.14)

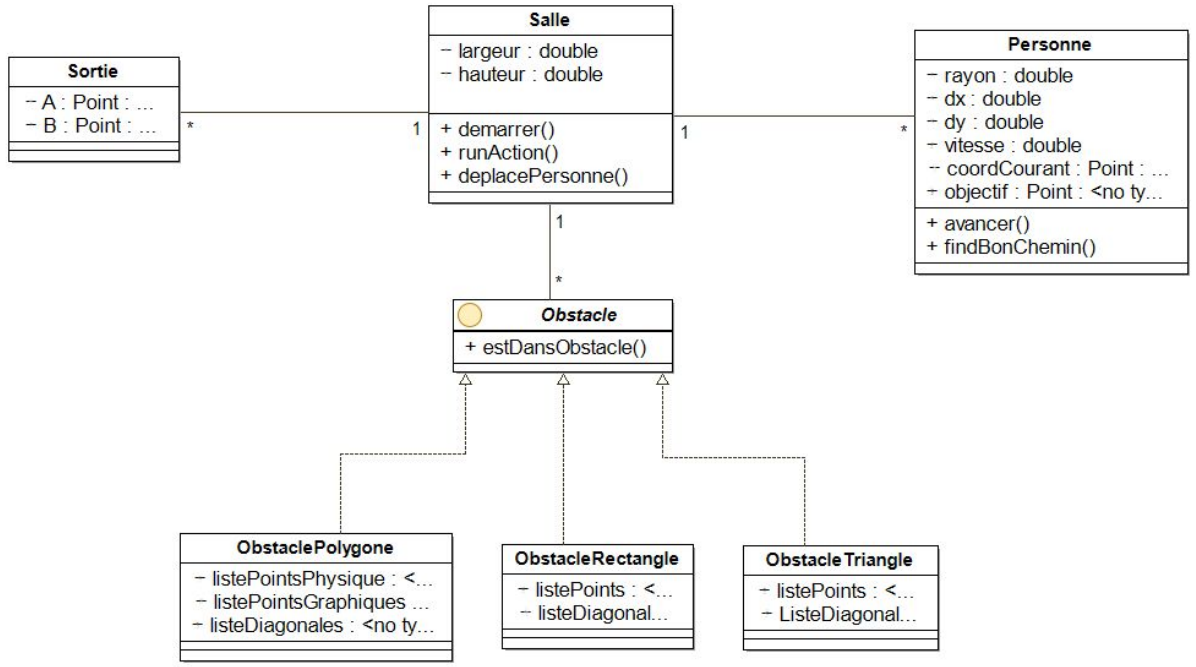


Figure 2.14 : Diagramme de classes (Données triviales)

L'élément au centre des données est la Salle, une Salle est définie par une largeur et une hauteur.
Cette dernière contient une liste de plusieurs Sortie, celles-ci ont pour paramètres deux points permettant de définir une Sortie par un segment d'un point A à un point B.
Ensuite la Salle comporte aussi une liste d'Obstacle, qui peuvent être des ObstaclePolygone, ObstacleRectangle et ObstacleTriangle, qui implémentent l'interface Obstacle. Toutes ces classes sont définies par la liste des points de leurs sommets, ainsi qu'une liste composée des segments des diagonales.
Pour finir, la salle possède une liste de Personne qui sont chacune définies par un rayon, une vitesse, une coordonnée et un objectif.
Depuis une Salle, on peut démarrer les différents calculs pour la simulation.

2.7.3. Données non triviales

Suite à la structure des classes principales, il est nécessaire d'implémenter des classes supplémentaires : (voir Figure 2.15)

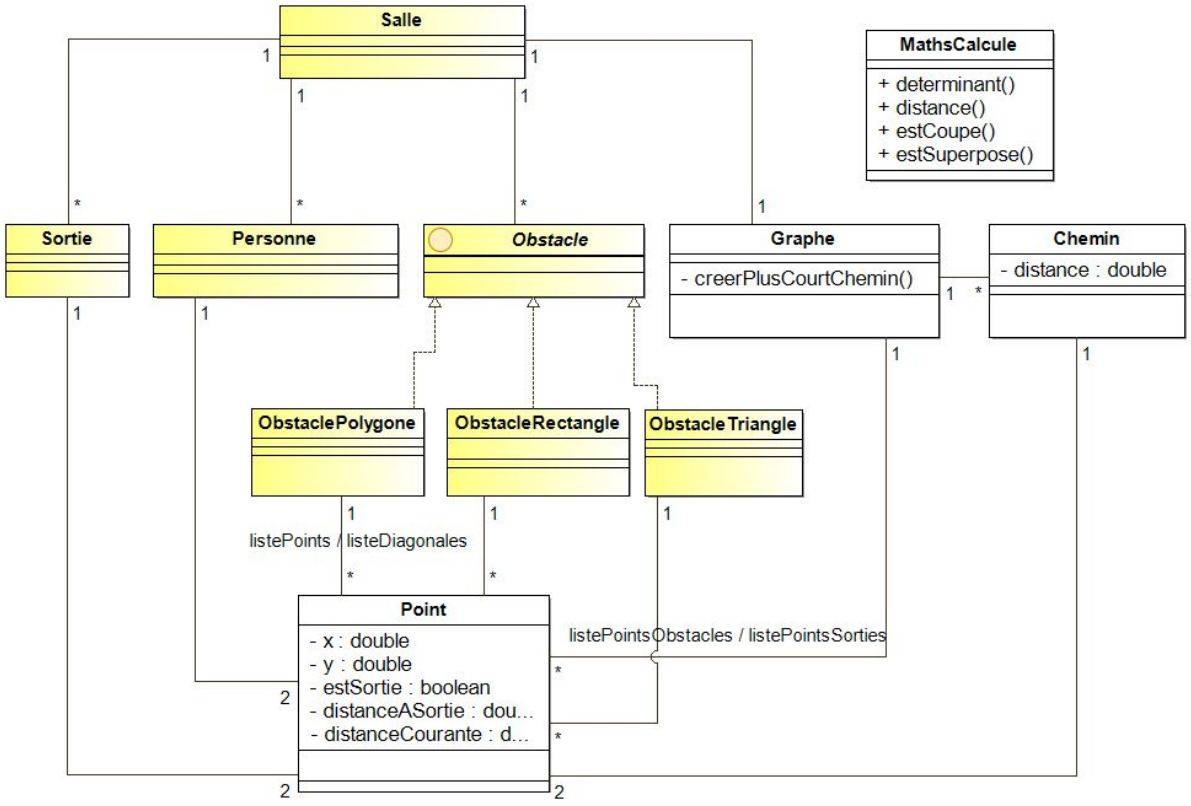


Figure 2.15 : Diagramme de classes (Données non triviales)

Comme indiqué sur la figure une salle est associée à un graphe, la classe Graphe contient une liste de points d'Obstacle et de Sortie afin de générer la liste de chemins la plus courte vers une Sortie depuis un Obstacle.

Ensuite la classe Chemin est représentée par une distance et deux Point, ces données correspondent à un segment.
La classe Point a pour attribut une coordonnée (x ; y) mais possède aussi un Point qui correspond à son suivant lorsqu'un chemin est généré.
Pour finir, une dernière classe MathsCalcule est implémentée qui n'est associée à aucune des autres classes, elle stocke simplement différentes opérations mathématiques que l'on utilise dans nos différentes méthodes.

2.8. Graphique

Pour visualiser les résultats provenant de l'algorithme, il est nécessaire d'afficher les différents éléments graphiquement.
L'interface graphique est décomposée en deux parties (voir Figure 2.16), une première correspondante à l'affichage de la salle et de tous ses composants (obstacles, sorties, personnes). Et, une seconde partie, un panneau de contrôle pour que l'utilisateur puisse interagir avec le simulateur.

Figure 2.16: Décomposition de l'interface graphique

Afin d'arriver à ce résultat, il existe différentes classes qui ont toutes pour père la classe Parent de JavaFX et la structure ci-dessous. (voir Figure 2.17)

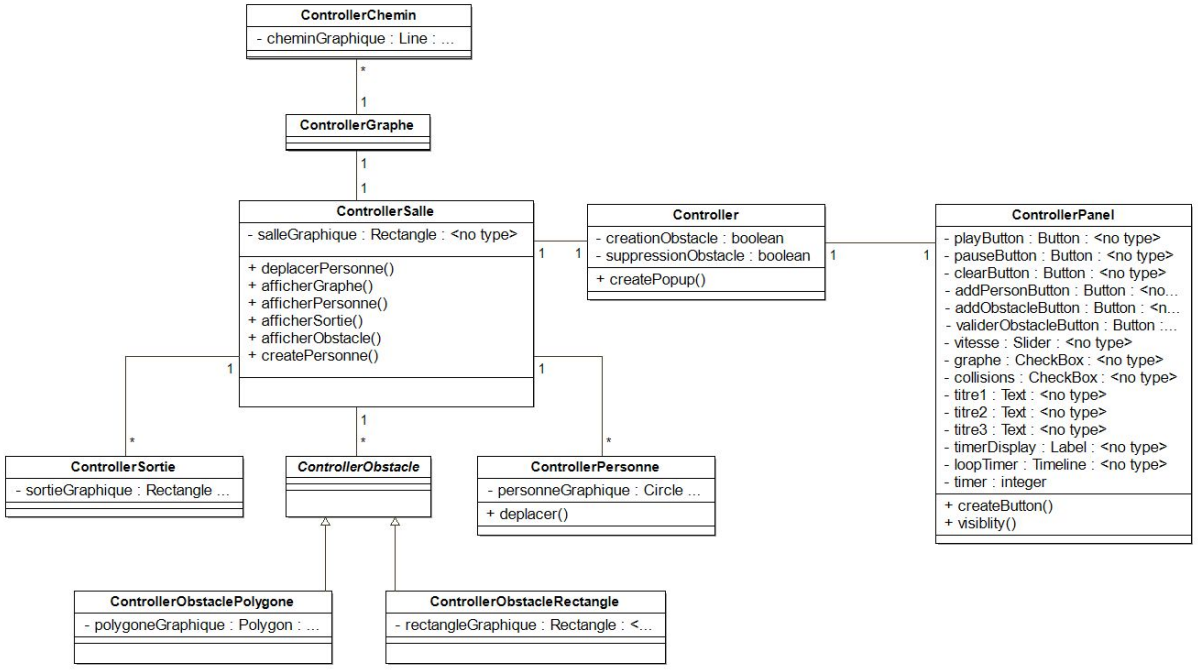


Figure 2.17 : Diagramme de classes de l'interface graphique

La classe Parent de JavaFX permet d'ajouter les éléments graphiques des classes à la scène principale, celle qui est affichée lors de l'exécution du programme.
En effet, toutes les classes controller sont stockées dans un Groupe (JavaFX).

Chacune de ces classes est associée à sa classe physique. (exemple: ControllerSortie a un attribut Sortie)
Premièrement le ControllerSalle représente la salle à partir de la classe Rectangle(JavaFX), elle est associée aux éléments graphiques qu'elle contient c'est-à-dire au ControllerSortie, ControllerObstacle, ControllerPersonne et ControllerGraphe. C'est à partir de cette classe que les composants s'affichent et que l'actualisation du déplacement des personnes graphiquement s'effectue.
Ensuite, le ControllerSortie permet de représenter l'élément par un rectangle avec une largeur définie dans le code.
Par ailleurs, le ControllerObstacle une classe abstract parent des classes ControllerObstaclePolygone et ControllerObstacleRectangle représenté par la classe Polygon(JavaFX) et Rectangle(JavaFX).
Par la suite, le ControllerPersonne est symbolisé graphiquement par un cercle et il y a possibilité de mettre à jour la position graphique d'une personne depuis cette classe.
Pour finir sur les composants du ControllerSalle, il existe une classe ControllerGraphe représentant une liste de chemins affichés sous forme de segments à partir de la classe Line de JavaFX.

La seconde partie de l'interface graphique est le ControllerPanel, cette classe regroupe un lot d'éléments graphiques de JavaFX comme un bouton pour lancer la simulation, un slider pour modifier la vitesse de déplacement des personnes dans la salle, ou encore une checkbox pour afficher/désafficher le graphe.

Enfin, la classe regroupant les deux parties est le Controller.
Cette dernière enregistre les différents évènements (onMouseClicked méthode de JavaFX) dont les suivants par exemple:

- lorsque l'utilisateur clique dans la salle sous certaines conditions : une personne est créée à cet emplacement.
- lorsque l'utilisateur clique sur le bouton play : la simulation est lancée.

3. Résultats

3.1 Test et validation

Au début du programme on a fait une classe PersonneTest surtout pour nous permettre de tester les fonctions sur l'intersection de deux segments et sur l'intersection entre une personne et un obstacle voir les différents test dans l'Annexe 3. Ensuite l'implémentation graphique a été réalisée et on a beaucoup plus utilisé la partie graphique car on la trouvait plus lisible et mieux adaptée pour voir où étaient les problèmes car une class test nous dit juste que ça va se passer comme prévu alors que la partie graphique permet de voir comment le code fonctionne concrètement.
Donc une majeure partie des tests se sont effectués graphiquement, grâce à l'interface graphique, il est possible de proposer différentes situations de simulation et de visualiser le résultat provoqué par le code.
L'affichage du graphe est un paramètre supplémentaire qui est très utile aux tests graphiques.

3.2 Manuel d'installation

Utiliser le simulateur implique l'installation de divers outils, il est donc nécessaire de suivre les étapes suivantes :

- Disposer d'un IDE tel que Intellij Idea.
- Se procurer le SDK JavaFX.
- Récupérer le dossier source du simulateur. Le fichier est privé sur gitHub. Pour y avoir accès, il faut envoyer un mail à tom.sartori-letourneau@etu.umontpellier.fr avec votre identifiant gitHub, afin d'être ajouté au projet.
- Ouvrir ce dernier en tant que projet dans l'IDE.
- Importer les librairies de JavaFX au projet. Pour ce faire, une fois le projet ouvert dans l'IDE, aller dans project structure > Librairies. Avec le "+", ajouter le fichier "lib" du répertoire téléchargé précédemment.
- Importer les librairies de tests JUnit les plus récentes directement à partir de l'IDE en allant dans la classe de tests.
- Lancer le fichier java nommé main dans l'IDE.

3.3 Manuel d'utilisation

Une fois l'installation complétée, vous avez à disposition le guide ci-dessous afin d'utiliser toutes les fonctionnalités présentes sur l'interface graphique.

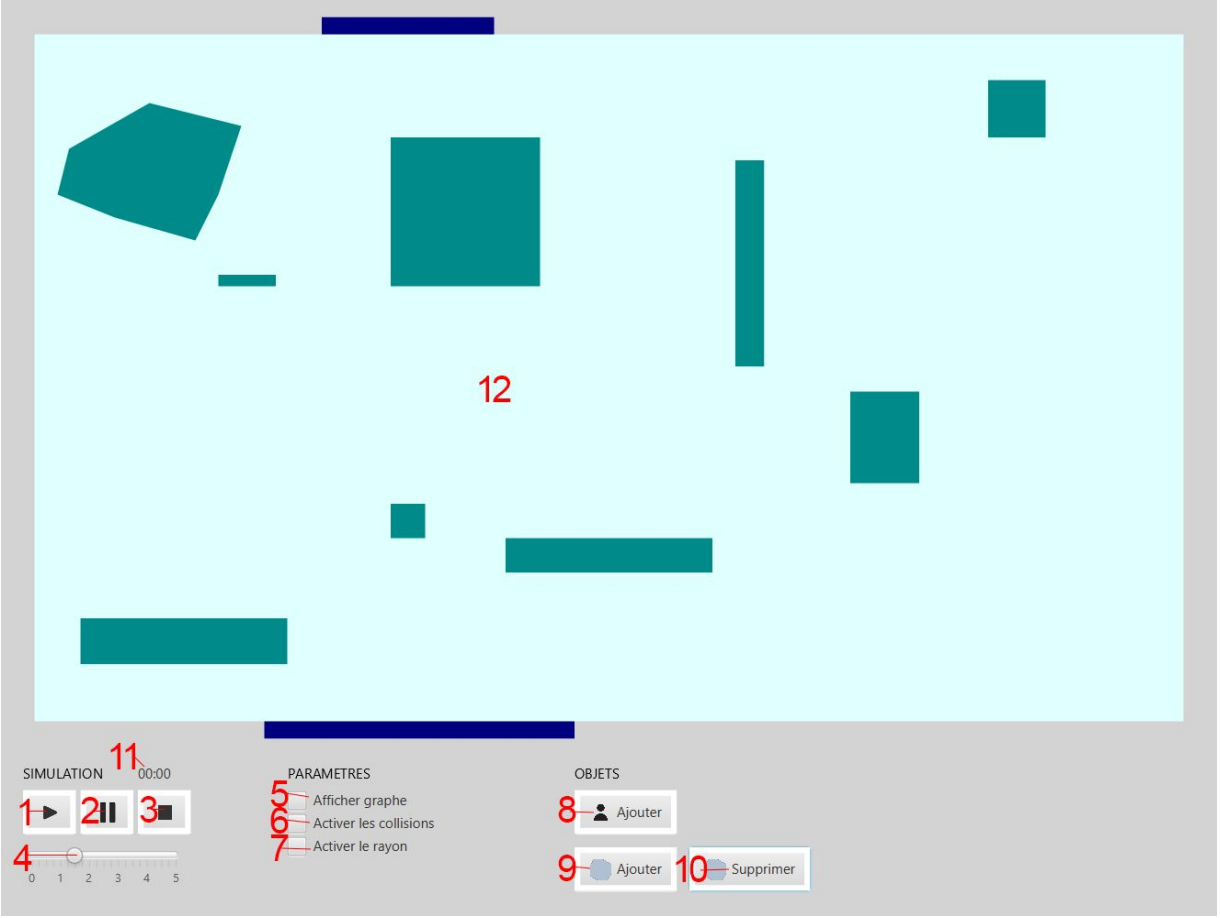


Figure 2.15 : Manuel d'utilisation

1- Lancer la simulation
2- Mettre en pause la simulation
3- Stopper la simulation, enlever toutes les personnes de la salle et remettre à 0 le chronomètre
4- Gérer la vitesse des personnes
5- Afficher le graphe des obstacles
6- Activer les collisions entre personnes
7- Activer la prise en compte du rayon des personnes
8- Ajouter des personnes aléatoirement dans la salle
9- Tracer un obstacle, chaque clic ajoute un point à l'obstacle
10- Supprimer un obstacle
11- Chronomètre de la simulation
12- Affichage de la simulation, si la simulation est en arrêt, cliquer dans la salle ajoute une personne

4. Rapport d’activité

4.1 Démarche personnelle

Notre projet a été un projet assez libre, comme vous avez pu voir dans la partie analyse des besoins non-fonctionnels, nous n’avions pas de directives précises quant à la réalisation de ce dernier. En effet, nous avons travaillé en autonomie lors de la recherche de solutions concernant des problèmes algorithmiques ou mathématiques.
Tout de même, nos tuteurs étaient là pour nous indiquer le chemin à suivre sans rendre la démarche linéaire. Par exemple, nous avons dû revenir sur nos décisions concernant les algorithmes de gestion de rayon de personne car notre solution n'était pas assez efficace, voire inconsistante. Nous avions donc, à proposer nos propres solutions comme pour les mathématiques autour de l’algorithme d’intersection des segments.
Cet état d’esprit est conservé lors de la conception de l’interface graphique qui reste sobre mais simple d’utilisation.
Nous avons également débattu avec nos tuteurs quant à l’ajout de fonctionnalités, leur pertinence et leur fonctionnement.

4.2 Gestion du projet

Pour ce projet, nous n’avions pas de langage de programmation imposé ou de bibliothèque graphique imposée. Nous avons donc choisis d'utiliser Java comme langage car il s'agit du langage principal étudié à l’IUT informatique. Pour la partie graphique, nous avons utilisé JavaFX car il s’agit de l’outil le plus adapté au contexte.

Concernant nos moyens de communications, au vu de la situation épidémique qui a eu lieu lors du déroulement de ce projet, nous étions en contact principalement via discord, autant entre nous qu’avec nos tuteurs. Nous nous envoyons régulièrement des informations compte tenu du travail à faire et fait.
Le projet était présenté une fois par semaine minimum à nos tuteurs pour avoir des retours et pour présenter nos prochaines idées de développement sur le projet. Nous envoyons aussi régulièrement des mails aux tuteurs dans le même but.

Le code du projet était sur une plateforme de versionning pour faciliter le travail en groupe et permettre aux tuteurs d’avoir accès au code de manière facile. Nous avons choisi d’utiliser GitHub pour faire cela car c’est la plateforme que nous utilisons régulièrement lors des travaux dirigés à l’IUT et donc, nos connaissances sur ce dernier nous ont permis de s’en servir facilement et efficacement.
Nous nous sommes servi également de GitHub pour y inscrire une todo list ou chacun de nous y accède pour voir quelles étaient les tâches à effectuer. Nous y indiquons également l’importance de chaque tâche et qui effectuait quelle tâche.

Conclusion

Tout d'abord, on a essayé de réaliser tous les objectifs que les tuteurs nous ont imposés. Un utilisateur doit pouvoir rajouter une personne dans la salle avec un simple clic, enlever toutes les personnes de la salle, changer la vitesse de déplacement des personnes, afficher les chemins les plus courts vers la sortie, prendre en compte la gestion du rayon des personnes, considérer la gestion des collisions, lancer la simulation, mettre en pause la simulation.
Pour satisfaire ces objectifs nous avons dû faire des choix. Pour ajouter et enlever des personnes de la salle, on a d'abord ajouté une classe ControllerPersonne et ControllerSalle afin de séparer la partie physique et la partie graphique, on a ensuite utilisé des fonctions simples de javaFx comme la fonction OnMouseClicked qui permet de faire une action quand l'utilisateur clique avec sa souris(chaque action graphique a une répercussion sur la partie physique). Pour le changement de la vitesse de déplacement des personnes ainsi que pour le lancement et la mise en pause du simulateur, il nous a juste suffi d'utiliser des fonctions de javaFx comme la fonction button pour nous permettre de savoir ce que l'utilisateur choisit de faire. Pour le lancement et la mise en pause de la simulation, on a dû aussi faire nos propres fonctions, tout d'abord une timeline qui rafraîchit l'image affichée toutes les 20ms cela permet de voir une simulation fluide et en même temps réaliste car on voit la personne se déplacer et non pas se téléporter.
Pour la prise en compte de la gestion du rayon des personnes, après avoir essayé de réfléchir à plusieurs solutions, nous avons trouvé la solution qui nous paraissait la plus adaptée. Cette solution consiste à augmenter la taille des obstacles (uniquement que dans la partie "physique" du code) par rapport au rayon unique des personnes. Pour la gestion des collisions, on a aussi beaucoup réfléchi. On a fait 2 versions différentes, la 2ème version qui est la plus complète d'après nous, consiste à utiliser une distanciation sociale entre chaque personne d'une taille prédéfinie à l'avance et qui permet donc de réduire au maximum les cas où les personnes se rentraient dedans. Afin d'afficher les plus courts chemins vers une sortie, il nous a fallu d'abord ajouter le principe de graphe ainsi que le principe d'intersection afin de savoir quel chemin était le plus court sans traverser d'obstacle.
Pour les différentes perspectives d'évolution, la solution des collisions entre personnes peut être améliorée en faisant en sorte que si les personnes sont bloquées alors ils bougent dans une direction non bloquante avec un petit vecteur afin de se débloquer et de changer son objectif par un objectif un peu plus éloigné. La gestion du rayon peut être aussi améliorée, en prenant en compte que chaque personne a un rayon différent et de les faire se déplacer en fonction. Une évolution possible à laquelle nous avions pensé est la mise en place d'un jeu où l'utilisateur a un temps limité pour faire sortir un nombre défini de personnes et son seul moyen pour les faire sortir le plus rapidement possible et de rajouter des obstacles afin de contrôler les flux de personnes.

En plus, ce projet s'est avéré très formateur. En effet nous avons dû faire des choix sur la programmation et le fonctionnement de notre simulateur sans être forcément aiguillés par les tuteurs sur quels étaient les bon choix à prendre. Par exemple pour le côté programmation on a décidé de rajouter une classe MathsCalcule afin de généraliser toutes les applications mathématiques complexes, pour le côté fonctionnement nous avons décidé

à l’aide des professeurs cette fois-ci qu’ils n’étaient pas faux de partir du principe que toutes les personnes ont le même rayon afin de faciliter la mise en oeuvre du déplacement des personnes par rapport à leur rayon. Ces décisions étaient prises lors de réunions faites avec tous les membres du groupe et les solutions étaient testées et expliquées au professeur uniquement si tous les membres du groupe étaient d’accord avec cette solution. Avec ce fonctionnement aucun problème n'est né au sein du groupe et même en discuter entre nous nous permettait de combiner les idées de plusieurs personnes. Ce projet nous a aussi permis de comprendre la puissance d’un langage orienté objet, et nous a permis de mettre en œuvre des applications mathématiques complexes sous forme de code. Les principales difficultés techniques qui ont été surmontées sont la gestion des rayons, des collisions, l’intersection d’un segment AB à un obstacle quand le segment AB est superposé à une diagonale de l’obstacle ainsi que la solution pour trouver les chemins les plus courts.

Enfin les tuteurs ayant préalablement réalisé un projet semblable au nôtre. Cela nous a permis dès le début des réunions de comprendre leurs attentes ainsi que leurs objectifs, ils ont ainsi pu anticiper certains de nos problèmes et nous en faire part en amont. Ceci ne nous a pas empêchés de proposer et de mettre en œuvre nos propres solutions. Cette méthode de travail s'est trouvée très instructrice et nous a permis d’avoir une plus grande autonomie.

Références bibliographiques

[1] *B. Maury, « Simulation de mouvements de foules », p. 20.*

[2] *« Images des mathématiques ». https://images.math.cnrs.fr/Modelisation-de-mouvements-de-foules (consulté le déc. 09, 2020).*

[3] *« ONHYS - Simulateur ONHYS ». https://www.onhys.com/fr/solutions/simulator (consulté le déc. 09, 2020).*

[4] *S. Lemercier, « Simulation du comportement de suivi dans une foule de piétons à travers l'expérience, l'analyse et la modélisation », p. 125.*

[5] *« La simulation de foule au service du divertissement ». https://interstices.info/la-simulation-de-foule-au-service-du-divertissement/ (consulté le déc. 09, 2020).*

Annexes techniques

Annexe 1 : Système mathématique de segments sécants

$A(xA, yA)$

$B(xB, yB)$

$C(xC, yC)$

$D(xD, yD)$

$$k \in ?[0;1] \qquad \overrightarrow{AM} = k\overrightarrow{AB} \qquad M\begin{cases} x = xA + k(xB - xA) \\ y = yA + k(yB - yA) \end{cases}$$

$$k' \in ?[0;1] \qquad \overrightarrow{CM} = k\overrightarrow{CD} \qquad M\begin{cases} x = xC + k'(xD - xC) \\ y = yC + k'(yD - yC) \end{cases}$$

$$\Leftrightarrow M\begin{cases} (xB - xA)\,k + (xC - xD)\,k' = xC - xA \\ (yB - yA)\,k + (yC - yD)\,k' = yC - yA \end{cases}$$

$$\Leftrightarrow M\begin{cases} ak + bk' = u \\ ck + dk' = v \end{cases}$$

on met le système sous forme de matrice A

$A\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ On calcul son déterminant $\det = ad - bc$

Si le déterminant est égal à 0 alors les segments sont Colinéaires.

Si $\det \neq 0$ alors

On calcul la matrice inverse de A

$$A^{-1} = \frac{1}{\det}\begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$$

Ensuite on la multiplie par $\begin{pmatrix} u \\ v \end{pmatrix}$

$$A^{-1} \times \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} k \\ k' \end{pmatrix}$$

$$\Rightarrow \left.\begin{matrix} 0 < k < 1 \\ 0 < k' < 1 \end{matrix}\right\} \exists M \text{ qui coupe AB et CD}$$

Annexe 2 : Algorithme de Dijkstra

Paramètres :
    Soit un graphe G
    Soit un noeud depart
    Soit un noeud arrive

Début
    Soit Q, une liste vide de noeuds
    Pour tout noeud n dans G, faire
        distance[n] ← infinie
        Q.ajouter(n)
    FinPour

    distance[arrive] ← 0
    Tant que Q n'est pas vide, faire :
        courant ← Q.noeudAvecMinDistance()
        Q.retirer(courant)
        listeVoisins ← G.voisins(courant)
        Pour tout noeud voisin dans listeVoisins, faire :
            newDistance ← distance[courant] + G.dist(courant, voisin)
            Si (newDistance < distance[n] alors :
                distance[n] ← newDistance
                precedent[n] ← courant
            FinSi
        FinPour
    FinTantQue

    return faireListe(depart, precedent)
Fin

Annexe 3 : Class PersonneTest

```java
@Test
 public void test_Obstacle_Sur_Chemin(){
   Personne p = new Personne(20,20);
   Point sortie = new Point(60,40);
   Point coordC=new Point(20,40);
   Point coordD=new Point(30,20);
   assertTrue(p.estTouche(sortie,coordC,coordD));
 }
 @Test
 public void test_Chemin_Sans_Obstacle(){
   Personne p = new Personne(20,20);
   Point sortie=new Point(60,40);
   Point coordC=new Point(10,40);
   Point coordD=new Point(20,30);
   assertFalse(p.estTouche(sortie,coordC,coordD));
 }
```

```
@Test
public void test_Personne_Longe_Mur(){
  Personne p = new Personne(20,20);
  Point sortie=new Point(60,40);
  Point coordC=new Point(28,24);
  Point coordD= new Point(10,30);
  assertFalse(p.estTouche(sortie,coordC,coordD));
}

@Test
public void test_Chemin_Colinéaire_Pas_Superpose_Xdep_Plus_Petit_Que_XCoinA(){
  Personne p = new Personne(20,20);
  Point sortie = new Point(60,20);
  Point coordC= new Point(80,20);
  Point coordD= new Point(100,20);
  assertFalse(p.estSuperpose(sortie,coordC,coordD));
}

@Test
public void test_Chemin_Colinéaire_Pas_Superpose_Xdep_Plus_Grand_Que_CoinA(){
  Personne p = new Personne(80,20);
  Point sortie= new Point(100,20);
  Point coordC=new Point(60,20);
  Point coordD= new Point(20,20);
  assertFalse(p.estSuperpose(sortie,coordC,coordD));
}

@Test
public void test_Chemin_Colinéaire_Superpose_Sur_X(){
  Personne p = new Personne(20,20);
  Point sortie= new Point(100,20);
  Point coordC= new Point(60,20);
  Point coordD= new Point(80,20);
  assertTrue(p.estSuperpose(sortie,coordC,coordD));
}

@Test
public void test_Chemin_Colinéaire_Pas_Superpose_ydep_Plus_Petit_Que_YCoinA(){
  Personne p = new Personne(20,20);
  Point sortie=new Point(20,60);
  Point coordC=new Point(20,80);
  Point coordD=new Point(20,100);
  assertFalse(p.estSuperpose(sortie,coordC,coordD));
}

@Test
public void test_Chemin_Colinéaire_Pas_Superpose_Ydep_Plus_Grand_Que_YCoinA(){
  Personne p = new Personne(20,80);
  Point sortie=new Point(20,100);
  Point coordC=new Point(20,60);
  Point coordD=new Point(20,20);
  assertFalse(p.estTouche(sortie,coordC,coordD));
}

@Test
public void test_Chemin_Colinéaire_Superpose_Sur_Y(){
  Personne p = new Personne(20,20);
  Point sortie=new Point(20,100);
  Point coordC=new Point(20,60);
  Point coordD=new Point(20,80);
  assertTrue(p.estSuperpose(sortie,coordC,coordD));
}
```

```java
@Test
public void test_Coord_X_pas_obstacle(){
  Personne p =new Personne(20,20);
  Point sortie = new Point(100,20);
  Obstacle o = new ObstacleRectangle(40,40,40,20);
  assertFalse(p.segmentObstacle(sortie,o));
}

@Test
public void test_parallèle_Diagonales_0_2_mais_pas_obstacles(){
  Personne p =new Personne(20,40);
  Point sortie = new Point(40,60);
  Obstacle o = new ObstacleRectangle(60,40,20,20);
  assertFalse(p.segmentObstacle(sortie,o));
}

@Test
public void test_parallèle_Diagonales_1_3_mais_pas_obstacles(){
  Personne p =new Personne(60,40);
  Point sortie = new Point(40,60);
  Obstacle o = new ObstacleRectangle(60,40,20,20);
  assertFalse(p.segmentObstacle(sortie,o));
}

@Test
public void test_Coord_Y_pas_obstacle(){
  Personne p =new Personne(20,20);
  Point sortie = new Point(20,100);
  Obstacle o = new ObstacleRectangle(40,40,40,20);
  assertFalse(p.segmentObstacle(sortie,o));
}

@Test
public void test_Coord_Obstacle_Avec_Coin_De_Obstacle_Sur_Chemin(){
  Personne p =new Personne(20,20);
  Point sortie = new Point(100,60);
  Obstacle o = new ObstacleRectangle(40,40,20,20);
  ArrayList<Point> listSolution = new ArrayList<>();
  assertFalse(p.segmentObstacle(sortie,o));
}

@Test
public void test_Coord_Obstacle_Avec_Obstacle_Surperpose_Sur_Chemin(){
  Personne p =new Personne(20,20);
  Point sortie = new Point(100,20);
  Obstacle o = new ObstacleRectangle(40,20,20,20);
  assertFalse(p.segmentObstacle(sortie,o));
}

@Test
public void test_Coord_Obstacle_Avec_Chemin_Travers_Obstacle_Par_Diagonale_0_2(){
  Personne p =new Personne(20,20);
  Point sortie = new Point(100,100);
  Obstacle o = new ObstacleRectangle(20,20,20,20);
  assertTrue(p.segmentObstacle(sortie,o));
}

@Test
public void test_Coord_Obstacle_Avec_Chemin_Travers_Obstacle_Par_Diagonale_1_3(){
  Personne p =new Personne(20,80);
  Point sortie = new Point(80,20);
  Obstacle o = new ObstacleRectangle(20,20,20,20);
  assertTrue(p.segmentObstacle(sortie,o));
}
```

Quatrième de couverture

Résumé

Ce rapport présente un projet réalisé en 2020-2021 par 4 étudiants en filière informatique de DUT avec l'aide des professeurs de l'IUT Informatique Montpellier-Sète. Ce projet concerne la mise en place d'un simulateur de mouvement de foule en passant par le développement, la méthodologie et la technicité de ce dernier, il est exclusivement réalisé en Java et se compose majoritairement d'algorithmes mathématiques vectoriels. Ce rapport présentera le cahier des charges, un rapport technique, un manuel utilisateur et un rapport d'activité.

Ce projet était enrichissant quant à la liberté laissée pour développer le projet, nous avons donc pu appliquer diverses méthodes apprises lors de notre enseignement à l'IUT tel que la méthodologie de projet ou les connaissances de programmation.