**Programmation Orientée Objet – Prosit 1**

*Prosit Aller : 2/11/2021*

**Prendre connaissance de la situation et clarifier**

*Mots-clés*

* **Repère orthonormé**
* **Coordonnées**
* **Unité de mesure**
* **Programme informatique**
* **Point**
* **Point de vérification (checkPoint) :** Un point de contrôle ou un point de contrôle d'application est un point d'un programme informatique où s'effectue, lors de l'exécution du programme, un arrêt au cours duquel sont copiées, sur un support externe (par exemple, un disque dur), les informations nécessaires afin de permettre, en cas d'arrêt ultérieur, la reprise du traitement à partir du dernier point de contrôle.
* **Standardiser :** La standardisation est l'action de ramener un produit, une production à une norme, à un modèle unique ou à un petit nombre de modèles aux caractéristiques définies.
* **Fonctionnalité et compilateur**
* **Modéliser**
* **Classe :** En programmation orientée objet, la déclaration d'une classe regroupe des membres, méthodes et propriétés (attributs) communs à un ensemble d'objets.

*Établir le contexte*

On travaille actuellement sur un projet qui cherche à gérer la déformation de points sur un repère orthonormé.

**Analyser le besoin**

*Analyser et décomposer le problème*

* Distance entre 2 points
* Modélisation dans un repère

*Livrables*

* Algorigramme
* Programme informatique (C++)
* Rapport

*Contraintes*

* 2 points
* Repère orthonormé
* Accessibilité du programme

**Généraliser**

*Saisir la nature du problème et déterminer la famille de problèmes à laquelle il appartient*

Modélisation dans un repère (2D)

**Trouver des pistes de solutions**

*Identifier des approches de solutions possibles*

*Théophile : Est-ce que la distance séparant les points est différent de 1 ? -> Oui.*

*Olivier : Faudra-t-il créer une échelle ? -> Oui.*

*Ilhan : Pourrons-nous faire le programme sans faire l’algorigramme ? -> Non.*

*Florian : L’algorigramme à réaliser sera-t-il composé de diagrammes UML ? -> Oui.*

*Celian : Un calcul mathématique sera-t-il nécessaire pour calculer la distance entre les points ? -> Oui.*

*Pierre : Faudra-t-il créer une ou plusieurs classes dans le programme ? -> Oui.*

*Celian : Un point peut-il être modélisé par un objet en C++ ? -> Oui.*

**Elaborer le plan d’actions**

*Identifier les éléments d’étude nécessaires à la résolution du problème*

* *Diagramme UML*
* *Langage de programmation C++*
* *« This » ? (Pierre)*
* *Classes / Objet*
* *Interface (POO)*

*Identifier les éléments de réalisation à produire*

* Algorigramme
* Programme informatique (C++)
* Rapport

*Liste, organiser, planifier et formaliser les actions à mener pour résoudre le problème*

* Colliger les informations
* Analyse le besoin
* Produire les diagrammes UML
* Faire le code
* Faire la note explicative
* Optionnel : Réaliser une simulation

**Phase AER :**

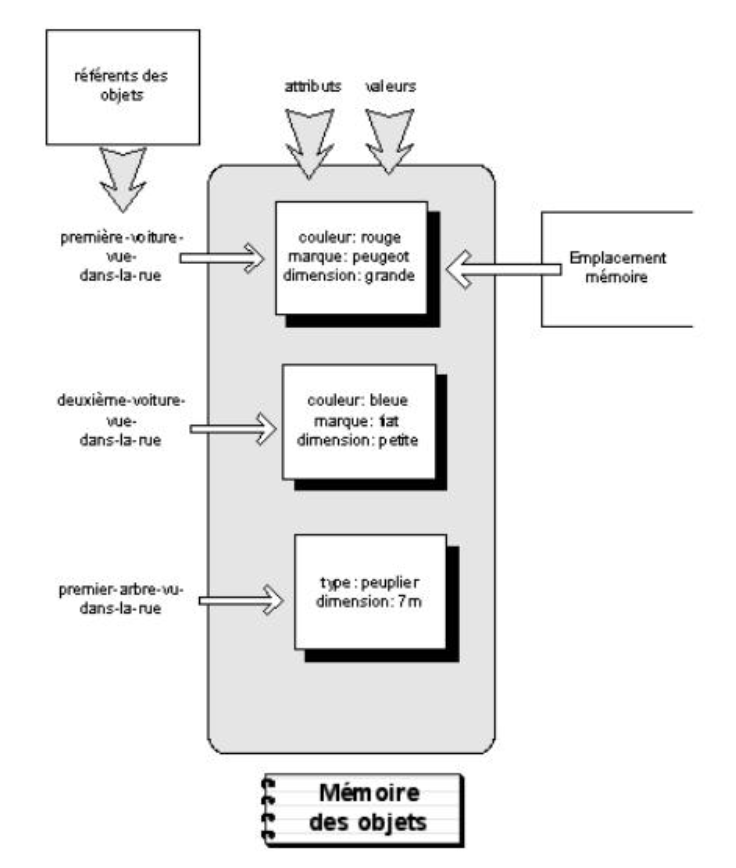
Durant ce prosit, nous nous sommes familiarisés avec les notions de base de la POO (Programmation Orientée Objet). Nous allons donc commencer par décrire ce qu’est la programmation orienté objet.

La programmation orientée objet (POO) est un paradigme informatique consistant à définir et à faire interagir des objets grâce à différentes technologies, notamment les langages de programmation. La programmation orientée objet se déroule en plusieurs étapes. La première consiste **à modéliser les données** en identifiant les objets que le programmeur souhaite manipuler ainsi que leurs interactions. Une fois que les objets ont été́ modélisées, **ils sont ensuite conceptualisés dans une classe d’objets**, qui aura des attributs et des méthodes. **Les attributs correspondent aux caractéristiques de l’objet et les méthodes aux actions que peut faire cet objet.** Une fois l’objet créé, il communique avec une interface bien définie appelée message.

Nous allons donc maintenant mettre en avant les concepts de bases de la POO :

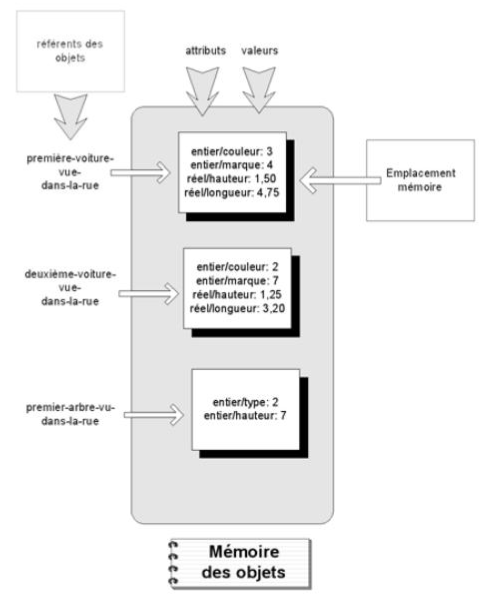
* **La classe** : une classe est un ensemble de code contenant des variables et des fonctions permettant de créer des objets. Une classe peut contenir plusieurs objets.
* **Les objets :** un objet est un bloc de code mêlant des variables et des fonctions, appelées respectivement attributs et méthodes. Les attributs définissent les caractéristiques d’un objet d’une classe, les méthodes définissent quant à elles les fonctions propres aux instances d’une classe.
* **L’encapsulation :** l'encapsulation permet d’enfermer dans une capsule les données brutes afin d’éviter des erreurs de manipulation ou de corruptions des données. L’encapsulation permet ainsi de cacher des méthodes et des attributs à l’extérieur de la classe.
* **L’abstraction :** le concept d’abstraction consiste à cacher les détails inutiles à l’utilisateur final d’une classe. Il va ainsi pouvoir utiliser une classe dans son code de programmation tout en ne sachant pas comment celle-ci a été́ développée.
* **L’héritage :** le concept d’héritage signifie qu’une classe B va hériter des mêmes attributs et méthodes qu’une classe A. Lorsqu’une instance de la classe B est créée, on peut alors appeler les méthodes présentes dans la classe A par la classe B. Cela va permettre de faire gagner du temps au programmeur.
* **Le polymorphisme :** lorsqu'une classe hérite des méthodes d’une classe parent, il est possible de surcharger une méthode, qui consiste à redéfinir la méthode de la classe parent pour que les deux classes ne fassent pas les mêmes taches.

Nous allons maintenant voir comment les objets sont stockés en mémoire. En effet lorsque nous faisons de la programmation orientée objet, nous pouvons voir différents objets apparaître. Il faut donc les stocker en mémoire.

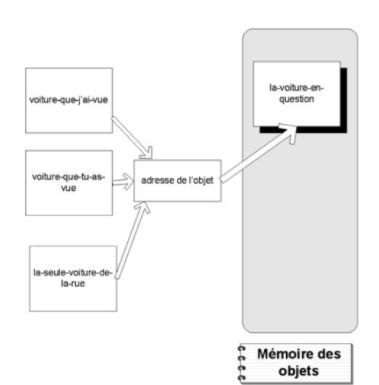


Il faut savoir que les objets sont structurellement décrits par un premier ensemble d’attributs de type primitif, tels qu’entier, réel ou caractère, qui sert à déterminer précisément l’espace qu’ils occupent en mémoire.

Nous allons maintenant parler des types primitifs. Lorsque nous parlons de type primitif, nous parlons d’entier ou de chaînes de caractères par exemple. Les types primitifs correspondent au types par défaut dans les langages de programmation.

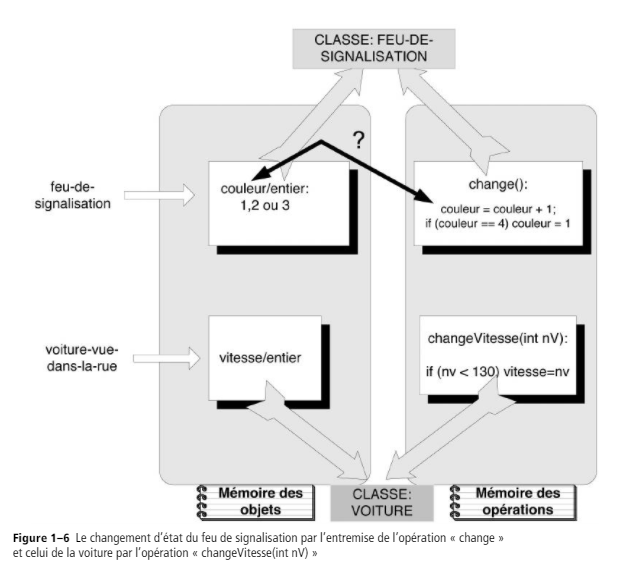


Il faut savoir que les objets que nous utilisons ont tous une référence. Le nom de l’objet en question est ce qui nous permet d’y accéder, on parle donc de référence de l’objet. Cette référence est l’adresse mémoire de l’objet.



Il existe aussi des relations entre les objets, ces relations ont un type : composition. Il peut donc y avoir certains objets qui sont contenue dans les autres objets et ne sont donc accessible que par le biais d’autre objets.

Un objet peut être aussi amener à changer d’état, c’est-à-dire qu’il va passer de l’état inactif l’état actif. Il peut aussi changer de mode. Cependant, le fait qu’il changent d’état ne change en rien leur identité. Ils peuvent changer d’état par des mécanismes qui leur sont propres ou encore avec des interactions avec d’autres objets. Pour changer d’état, il faut donc relier les opération par des attributs :



Nous allons maintenant détailler les classes. En effet, en c++ (le langage que nous utilisons pour ce projet et prosit) nous pouvons créer des classes. Cependant, nous pouvons aussi créer des structures. Nous allons donc, dans un premier temps expliquer concrètement ce qu’est une classe, puis une structure et comprendre la différence entre les deux.

Concernant la classe, c’est un plan qui définit les variables et les méthodes communes à tous les objets d'un certain type. C'est un type de données de type référence. Une classe peut hériter d'autres classes ou structures alors qu'une structure ne peut pas hériter d'autres classes ou structures. De plus, le destructeur est une autre différence entre Class et Structure. La classe peut avoir un destructeur, mais la structure n'a pas de destructeur. Il faut savoir qu’un destructeur est une fonction membre spéciale qui est appelée à la fin de la vie d'un objet. L'objectif du destructeur est de libérer les ressources que l'objet a pu acquérir pendant sa durée de vie. De plus, une autre différence entre Class et Structure est qu'une classe instancie un objet en utilisant un nouveau mot-clé, tandis qu'une structure instancie un objet sans utiliser de nouveau mot-clé. Le mot clé pour définir une classe est « class ». Aussi, dans une classe, s'il n'y a pas de spécificateurs d'accès déclarés, les membres sont privés.

Inversement, une structure est un type de données de type valeur qui peut contenir des données associées de différents types de données. Par conséquent, la principale différence entre la classe et la structure réside dans le type de données. De plus, l'instance d'une classe est un objet alors que l'instance d'une structure est une variable de structure. C'est donc une autre différence entre classe et structure. Le mot clé pour définir une structure est « struct ». Dans une structure, s'il n'y a pas de spécificateurs d'accès déclarés, les membres sont publics.

Pour conclure sur les classes et les structures, en effet lorsque nous programmons, nous utilisons des variables et donc nous stockons des données. Pour rappel, une variable est un emplacement utilisé pour stocker des données. Il existe deux type de variables :

* Types de valeurs ;
* Types de références.

Donc la différence fondamentale entre les classes et les structures est qu’une classe est un type de données de référence, alors qu’une structure est un type de données de valeur.

Nous allons maintenant voir la corbeille d’exercices. Cette corbeille nous a permis de nous familiariser avec le programmation POO.

#include <iostream>

#include <list>

using namespace std;

class CLvoiture

{

public:

CLvoiture();

CLvoiture(int newVitesseMax, int newAcceleration);

CLvoiture(CLvoiture& voiture)

{

acceleration = voiture.acceleration;

vitesse = voiture.vitesse;

vitesseMax = voiture.vitesseMax;

idV = voiture.idV;

idVoiture = voiture.idVoiture;

}

int getAcceleration() { return acceleration; }

int getVitesseMax() { return vitesseMax; }

int getID() { return idV; }

int setAcceleration(int newAcceleration) { acceleration = newAcceleration; }

int setVitesseMax(int newVitesseMax) { vitesseMax = newVitesseMax; }

void ini\_obj(int iniAcceleration, int iniVitesseMax)

{

iniAcceleration = acceleration;

iniVitesseMax = vitesseMax;

}

void accelerer() { cout << "Acceleration" << endl; }

private:

int vitesseMax, vitesse, acceleration, idV, idVoiture;

};

CLvoiture::CLvoiture()

{

}

CLvoiture::CLvoiture(int newVitesseMax, int newAcceleration)

{

vitesseMax = newVitesseMax;

acceleration = newAcceleration;

}

class Cla

{

public:

//Partie exercice 3

Cla();

Cla(int parametre);

//Partie exercice 2

void afficher(void)

{

cout << "bonjour" << endl;

}

};

Cla::Cla()

{

}

Cla::Cla(int parametre)

{

}

class CLobjet

{

private :

double masse, vitesse, ec;

void init()

{

masse = 2;

vitesse = 5;

ec = 0.5f \* masse \* pow(vitesse, 2);

}

public :

double GetMasse() { cout << "Masse : " << masse << endl; return masse; }

double GetVitesse() { cout << "Vitesse : " << vitesse << endl; return vitesse; }

double GetEc() const { return ec; }

CLobjet()

{

init();

}

CLobjet(int newMasse, int newVitesse);

CLobjet(const CLobjet& parametre)

{

masse = parametre.masse;

vitesse = parametre.vitesse;

ec = parametre.ec;

}

void cheatInit(int newMasse, int newVitesse)

{

masse = newMasse;

vitesse = newVitesse;

}

void Calculer(void)

{

ec = 0.5f \* masse \* vitesse \* vitesse;

}

void Comparer(CLobjet obj1, CLobjet obj2)

{

if (obj1.ec > obj2.ec)

cout << "L'energie cinetique de l'objet 1 est plus grande !" << endl;

else

cout << "L'energie cinetique de l'objet 2 est plus grande !" << endl;

}

};

CLobjet::CLobjet(int newMasse, int newVitesse)

{

masse = newMasse;

vitesse = newVitesse;

}

void Exercice1()

{

Cla o1;

Cla\* o2;

o2 = new Cla;

o1.afficher();

o2->afficher();

cout << "Adresse de o1 : " << &o1 << endl;

cout << "Adresse de o2 : " << &o2 << endl;

cout << "Adresse pointée de o2 : " << o2 << endl;

cout << "Taille du type Cla : " << sizeof(Cla) << endl;

cout << "Taille en octet de o1 : " << sizeof(o1) << endl;

cout << "Taille en octet de o2 : " << sizeof(o2) << endl;

delete o2;

}

void Exercice2()

{

CLobjet object1;

CLobjet object2;

object1.Comparer(object1, object2);

}

void Exercice3()

{

CLobjet o1;

CLobjet o2(2, 2);

CLobjet\* o3 = new CLobjet;

CLobjet\* o4;

o3->cheatInit(4, 32);

o4 = o3;

o1.Calculer();

o2.Calculer();

o1.Comparer(o1, o2);

o3->Calculer();

o4->Calculer();

o3->Comparer(\*o3, \*o4);

o4->GetMasse();

o4->GetVitesse();

}

void Exercice4()

{

CLvoiture voiture1(120, 20);

CLvoiture voiture2(140, 30);

CLvoiture\* voiture3 = new CLvoiture;

voiture1.accelerer();

voiture2.accelerer();

voiture3->accelerer();

}

int main()

{

cout << "Exercice 1 :" << endl;

Exercice1();

cout << "\nExercice 2 :" << endl;

Exercice2();

cout << "\nExercice 3 :" << endl;

Exercice3();

cout << "\nExercice 4 :" << endl;

Exercice4();

}

Au total, nous avons 3 classes.

* Cla ;
* CLobjet ;
* CLvoiture.

Dans chacune de ces classes, nous avons pu nous exercer à faire des méthodes. Nous avons aussi fait de l’allocation dynamique.

Nous allons maintenant passer à la résolution du prosit. Dans un premier temps, nous allons voir toute partie UML qui nous a permis de construire notre classe, mais aussi de créer l’algorithme que nous avons mis en place afin de résoudre le prosit.

Nous allons commencer par le diagramme de classe.

Une image contenant table

Description générée automatiquement

Comme nous pouvons le voir, nous avons créé une classe Matrice. Cette classe à pour but de représenter concrètement notre repère en 2D. L’avantage que nous avons lorsque nous utilisons des matrices, c’est que nous pouvons faire des matrices avec autant de dimensions que nous voulons. Nous allons donc maintenant parler des attributs de notre classe.

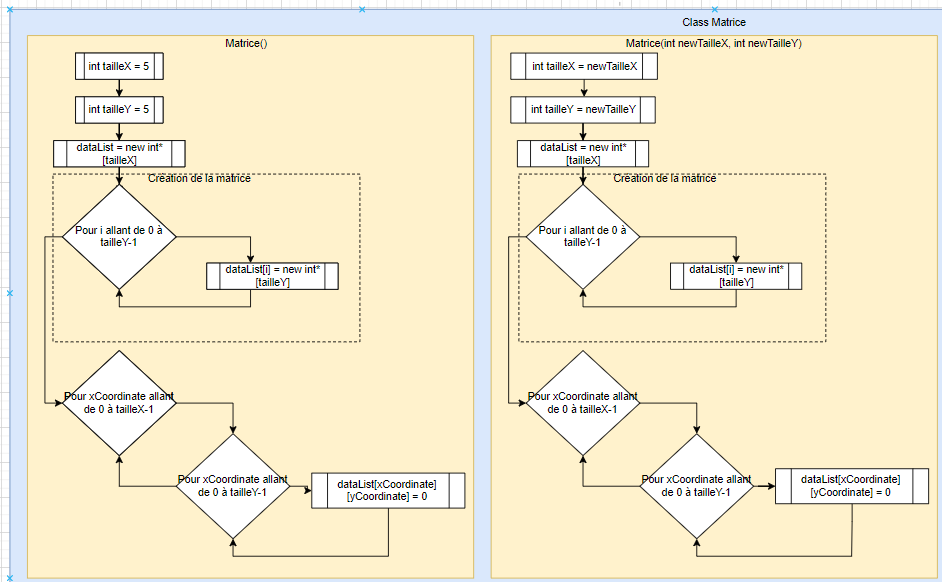
* tailleX est la taille en X de notre matrice. Par défaut la valeur affectée à cette variable est 5.
* tailleY est la taille en Y de notre matrice. Par défaut la valeur affectée à cette variable est 5.
* x est la coordonnée en x du point que nous allons placer dans notre matrice.
* y est la coordonnée en y du point que nous allons placer dans notre matrice.

Nous avons ensuite déclaré trois méthodes et un dernier attribut qui sera public à la classe. Parmi les méthodes, nous avons :

* setPoint(int xCo, int yCo) cette méthode permet de créer un point dans la matrice (ou de changer ses coordonnées si existe déjà). Cette méthode possède deux paramètres : xCo qui correspond aux coordonnées en x et yCo qui correspond aux coordonnées en y.
* getPoint(Matrice obj) cette méthode permet de d’afficher dans la console les coordonnées du point de l’objet en paramètre.
* dispMatrice() cette méthode permet d’afficher la matrice.

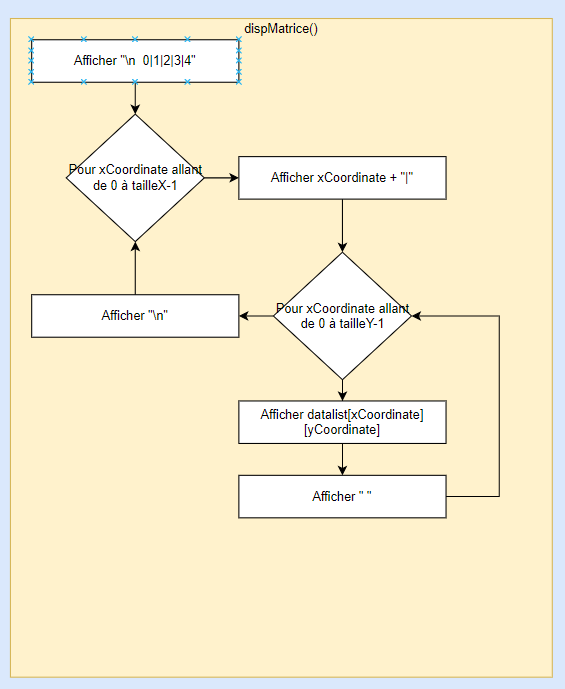
Le dernier attribut que nous avons créé et public car il nous permet de créer notre matrice. Mais nous irons plus dans le détails lorsque nous expliquerons notre code.

Nous avons ensuite fait un algorigramme pour mettre en place notre algorithme. Pour cela nous avons fait un algorigramme, découpé en deux parties. La première permet de décrire la classe Matrice que nous allons utiliser. La deuxième représente le main, c’est-à-dire code exécuté.

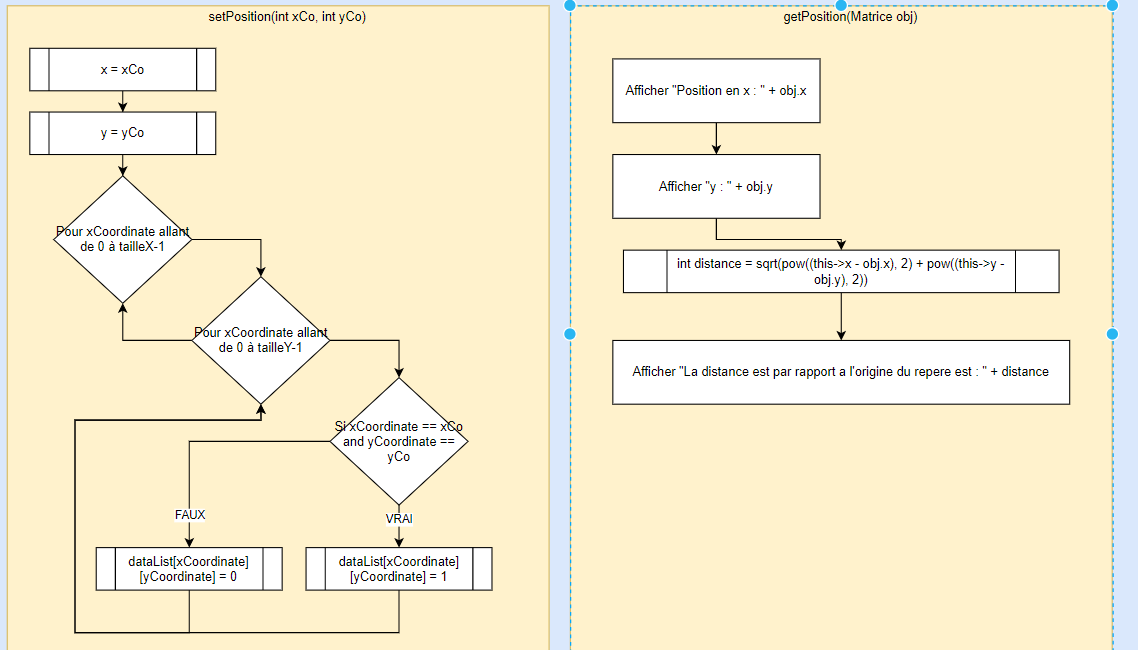


Dans un premier temps, nous avons créer le constructeur statique. Ce qui veut dire que lorsque nous initialisons un objet, et que nous ne lui mettons pas de paramètres, il prend les valeurs par défaut. Dans un premier temps, nous allons initialiser la tailleX et la tailleY à 5, pour faire une matrice en 5 par 5. Ensuite nous créons notre matrice qui est un double pointeur comme nous faisons un tableau en deux dimensions. Puis nous remplissons notre matrice de 0 pour dire qu’il n’y a pas de points.

Nous allons maintenant parler du constructeur statique paramétrer. Concrètement, nous faisons le même code que dans le constructeur statique, à la seule différence que les variables de taille de la matrice, sont affecter en fonction des paramètres.

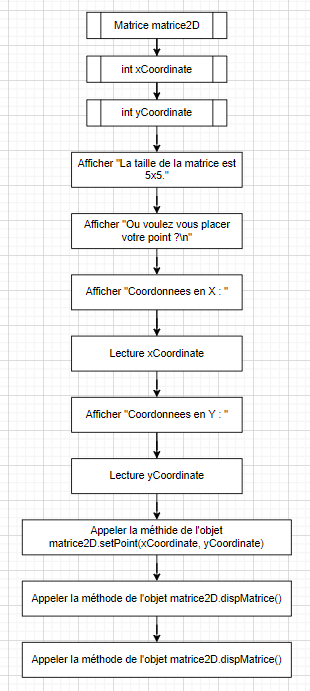


Nous allons maintenant décrire la méthode dispMatrice() qui va nous permettre d’afficher notre matrice une fois compléter. Comme nous pouvons voir, dans un premier temps nous affichons les numéros de colonnes. Ensuite, par le biais de deux for imbriquer, qui nous permettent de faire varier les coordonnées en x et y, nous affichons notre matrice.



A présent, nous allons parler de la méthode setPoint qui à pour paramètres les coordonnées en x (xCo) et en y (yCo). Dans un premier temps, nous affectons aux attributs x et y les valeurs des paramètres respectif. Ensuite, par le biais d’un double for (comme dans la méthode précédente), nous parcourons notre matrice. A l’aide d’une condition if, nous vérifions si nous ne sommes pas sur le point que nous voulons mettre. Dans le cas, où nous sommes sur ce point, nous affectons à la matrice (aux coordonnées indiquer) la valeur 1.

Nous allons maintenant, étudier la méthode getPoint(). Dans un premier temps, nous affichons les coordonnées de l’objet que nous récupérons. Ensuite, par nous calculons la distance parcourue dans le repère par rapport à l’origine. Puis nous l’affichons.



Nous allons maintenant décrire le main(), le code qui va être exécuter. Dans un premier temps, nous créons un objet de type Matrice, que nous allons appeler matrice2D. Cet objet correspond à notre matrice. Nous allons ensuite initialiser deux variable int qui vont êtres les coordonnées du point à placer. Nous allons lire la valeur des coordonnées. Puis nous allons appeler la méthode setPoint par le biais de l’objet matrice2D. Nous utiliserons en paramètre de cette méthode, les deux variables lue juste avant. De plus, nous utilisons la méthode getPoint avec pour paramètre notre matrice2D, sur notre objet matrice2D. Enfin, nous appelons la méthode dispMatrice de notre matrice2D.

Une image contenant texte

Description générée automatiquement

Voici donc le rendu que nous avons avec la console.

Nous avons donc cherché différentes piste d’amélioration du prosit. En effet, le but était de mélanger la programmation orientée objet à la représentation graphique dans un repère. Dans notre cas, nous avions fait un affichage de notre matrice.

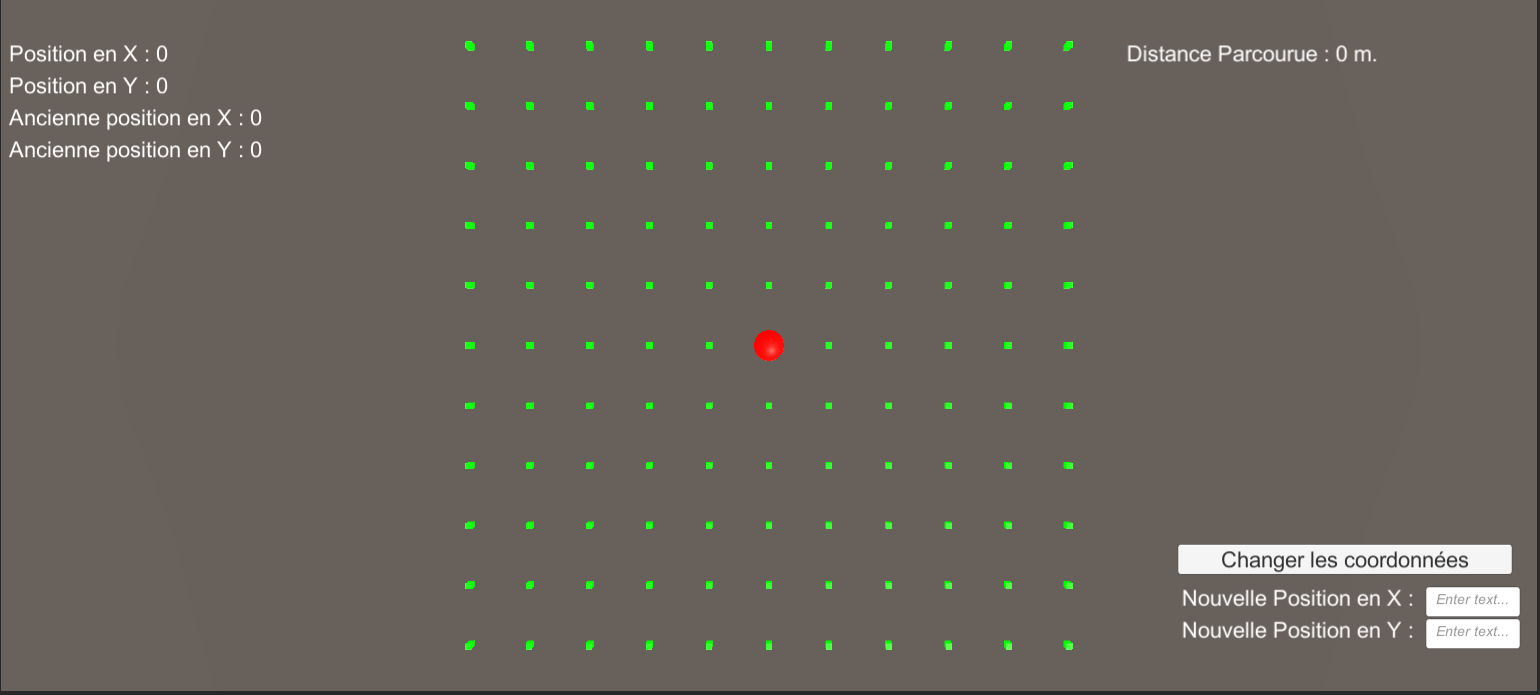
Nous aurions pu aussi simuler cette expérience autrement. Nous avons par exemple fait une simulation sur Unity. Dans le cadre de cette simulation, nous avons utiliser un autre langage que le C++. Ce langage est le C#. Le C# est un autre langage de programmation orienté objet. Unity est un éditeur de jeux vidéo de base aussi, mais c’est un très bon outils pour faire des simulations.

Dans un premier temps, nous avons chercher à créer une scène (le « monde » dans lequel va se dérouler la simulation). Nous avons donc dû recréer un repère. Nous avons utiliser des cubes pour simuler ce repère.

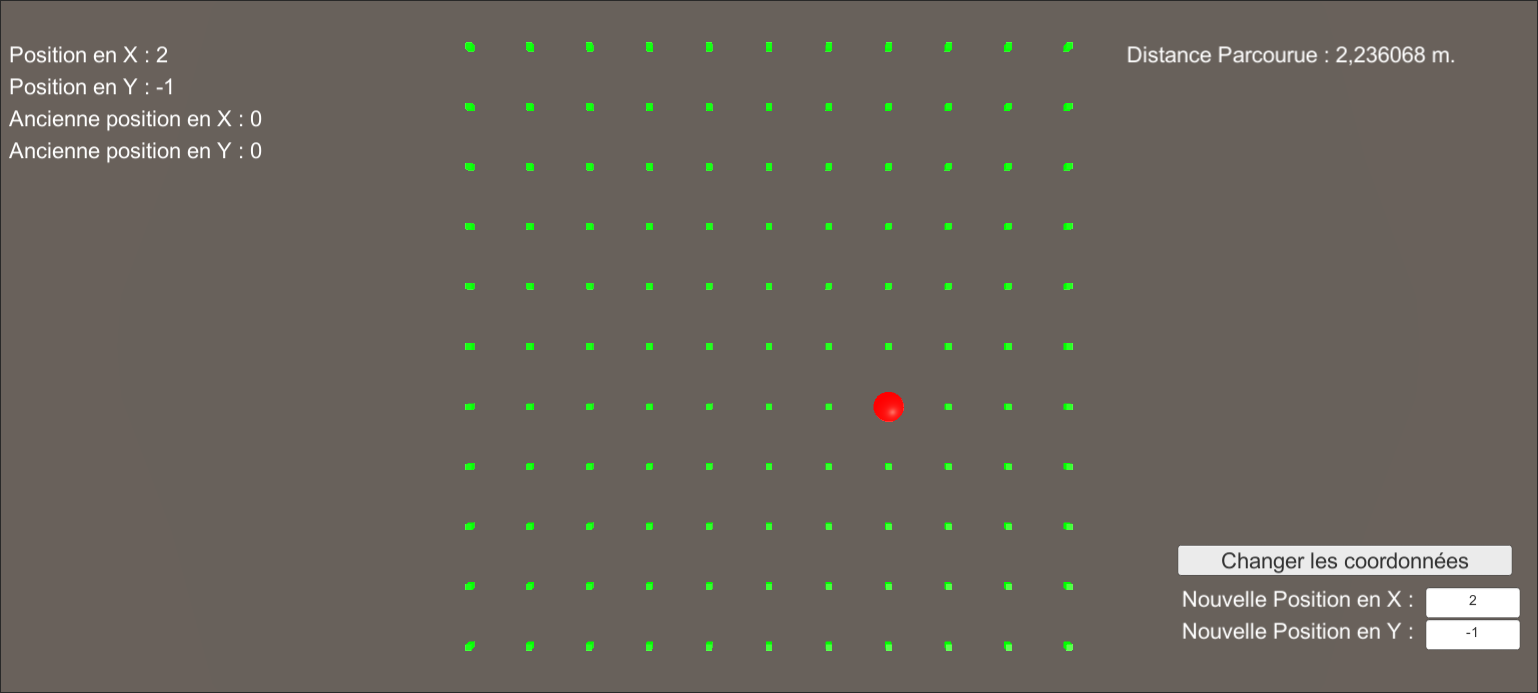
Une image contenant intérieur, lumière

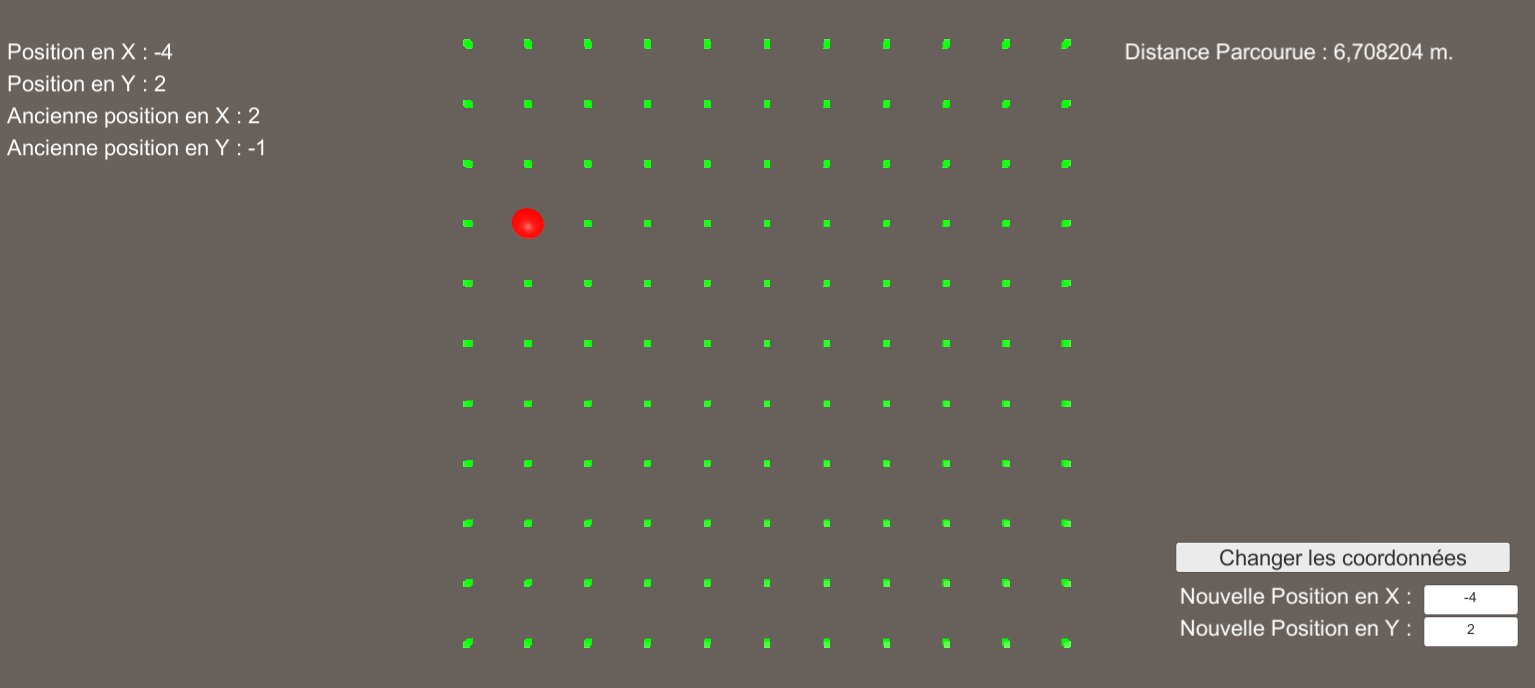
Description générée automatiquement

Comme nous pouvons le voir, nous avons, avec les cubes vert, notre repère 2D. Le point (symbolisé par la sphère rouge) est situé de base en [0;0]. Après avoir organiser correctement la scène (en ajoutant un canvas : la partie UI), nous avons cette vision lorsque nous lançons la simulation.



Comme nous pouvons le voir, nous avons un bouton qui permet de changer les données, lorsque les nouvelles coordonnées du point sont entrées. Lorsque nous appuyons sur ce bouton avec de nouvelles coordonnées, le point se déplace aux coordonnées entrée.

Pour l’exemple, nous avons pris un nouveau point de coordonnées [2 ;-1]. Nous pouvons voir que le point s’est déplacé aux bonne coordonnées. Nous avons affiché les coordonnées du point à l’instant T et la distance parcourue par rapport à sa dernière position. Ce qui signifie que, nous pouvons bouger notre point de nouveau et la distance sera calculée par rapport à la dernière position.



Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement