|  |  |  |
| --- | --- | --- |
|  | | |
| **Projet informatique**  **FISE - 2021** | | |
|  | ENSTA Bretagne  2 rue F. Verny  29806 Brest Cedex 9, France  LE TOLGUENEC, Paul-Antoine  SAINTE-ROSE-FANCHINE, Mélanie |  |

Sommaire

[Introduction 2](#_Toc10730142)

[1. Présentation du jeu 3](#_Toc10730143)

[1.1. L’objectif du jeu 3](#_Toc10730144)

[1.2. Les péripéties 3](#_Toc10730145)

[1.3. Les déplacements 4](#_Toc10730146)

[1.4. Lancement du jeu 4](#_Toc10730147)

[2. Description générale du problème 4](#_Toc10730148)

[2.1. Les différentes pistes envisagées 4](#_Toc10730149)

[2.2. Les hypothèses réductrices 5](#_Toc10730150)

[3. Description générale du programme 7](#_Toc10730151)

[3.1. Présentation générale 7](#_Toc10730152)

[3.2. La classe Elément 8](#_Toc10730153)

[3.2.1. La méthode *update()* : 8](#_Toc10730154)

[3.3. La classe Monde 10](#_Toc10730155)

[3.3.1. La méthode *creation\_mur()* : 10](#_Toc10730156)

[3.3.2. Les autres méthodes de création : 10](#_Toc10730157)

[3.3.3. La méthode *update()*: 10](#_Toc10730158)

[3.4. Les classes d’ennemis 10](#_Toc10730159)

[3.4.1. Les classes *Bombe* et *Robot*: 10](#_Toc10730160)

[3.4.2. La classe Missile : 10](#_Toc10730161)

[3.5. La classe Personnage 11](#_Toc10730162)

[3.6. La classe Simulation 12](#_Toc10730163)

[3.7. La classe Fenetre 12](#_Toc10730164)

[3.7.1. La méthode *clique\_start()* : 13](#_Toc10730165)

[3.7.2. La méthode *paintEvent()* : 13](#_Toc10730166)

[3.7.3. La méthode *keyPressEvent()* : 14](#_Toc10730167)

[4. Conclusion 14](#_Toc10730168)

[4.3. Tests effectués 14](#_Toc10730169)

[4.4. Limitations 14](#_Toc10730170)

[4.5. Perspectives et améliorations 15](#_Toc10730171)

# Introduction

Dans ce rapport nous allons décrire le projet que nous avons réalisé dans le cadre de l’UV 2.4. Notre projet est un jeu de plateforme que nous avons nommé Portal+. Le jeu consiste à déplacer un bonhomme uniquement avec les flèches du clavier pour que le bonhomme réalise les missions propres au jeu.

Dans un premier temps nous vous expliquerons le fonctionnement du jeu que nous avons voulu atteindre et le fonctionnement finalement atteint ; donc les règles du jeu, la problématique de jeu etc. Ensuite, nous vous expliquerons comment nous avons réalisé le code pour obtenir le jeu.

# Présentation du jeu

## L’objectif du jeu

Portal+ est un jeu de plateforme de type Mario super bros, sauf que dans ce jeu la fenêtre dans laquelle le personnage évolue ne bouge pas d’une part (alors que dans Mario le personnage peut se déplacer hors d’une fenêtre). D’autre part nous avons fait en sorte de nous servir des bords de la fenêtre pour représenter les bords de la map. Pour l’instant nous n’avons créé qu’un seul niveau qui fonctionne parfaitement. Nous aurions pu créer d’autres niveaux différents mais les utilisateurs s’apercevront assez vite que le seul accomplissement de ce niveau est assez compliqué.

Voici maintenant les règles : Le bonhomme apparaît à un endroit de la map. Il n’y a pas d’endroit précis d’apparition de façon à ce que le joueur ait à s’adapter en fonction de l’environnement où il est.

* La première mission : Le bonhomme doit récupérer une clef qui apparait à un endroit de la map qu’il peut déterminer dès le lancement du jeu. La clef est modélisée par une clef alène pour la petite blague d’ingénieur (dans les jeux arcade la clef est usuellement représentée par une vraie clef).
* Deuxième mission : Une fois la clef récupérée, elle disparaît instantanément et un portail s’ouvre à un autre endroit de la map.

L’objectif est de rejoindre le portail. Le portail est modélisé par une image de la célèbre porte de Stargate-SG1. Si le bonhomme atteint la porte alors qu’elle est ouverte, la personne gagne.

## Les péripéties

Bien sûr si la seule contrainte pour ouvrir le portail était de le rejoindre cela serait trop simple. Sur son chemin le bonhomme va rencontrer de nombreux ennemis. Les différents types d’ennemis sont :

* Les bombes : elles sont les ennemies les plus simples. Elles ne peuvent pas se déplacer. Si le bonhomme les touche, il meurt instantanément et c’est la fin du jeu. Les bombes sont représentées par une image orange de prévention d’un risque d’explosion.
* Les robots : ils sont un peu plus difficiles à gérer. En effet, ils tuent le personnage de la même façon que les bombes (par contact), mais en plus ceux-ci peuvent se déplacer. Globalement ils sont fixés sur un mur (à l’horizontale) et lorsque le bonhomme est sur le même horizon qu’eux les robots se déplacent vers le bonhomme pour le toucher. Or le robot ne peut pas aller plus loin que la fin du mur. Il reste toujours sur le mur où il apparaît. Le robot est représenté par une image blanche de robot.
* Les missiles : dans le jeu il n’y a en fait qu’un seul missile. Mais nous aurions pu en créer plus. Il s’agit de l’ennemi le plus dur à gérer. Le missile part toujours d’en haut à gauche de la fenêtre là où il y a une image de lanceur de missile (pour être cohérent). Le missile détecte le bonhomme quel que soit la direction et le sens et se dirige vers lui. Il tue le bonhomme de la même façon que les autres, en le touchant. Mais l’utilisateur peut se débarrasser d’un missile pour un temps restreint. En effet, si le missile touche un mur, il explose instantanément. Une fois l’explosion réalisée, un autre missile est lancé depuis le lanceur de missile. Et cela jusqu’à l’accomplissement du niveau ou la mort du personnage.

Une autre difficulté est que le personnage a un temps de vie précis. Etant donné qu’après avoir fait essayer le jeu à plusieurs personnes nous nous sommes rendu compte que les gens avaient assez de mal sans la contrainte de temps. Nous avons décidé de mettre une contrainte de temps très facile pour que le niveau soit réalisable. Pour l’instant, nous n’affichons pas le temps de vie car il n’a pas grand intérêt à ce stade du jeu. Mais si nous réalisons d’autres niveaux plus simples, cette contrainte pourrait être exploitable.

## Les déplacements

Pour réaliser sa mission le personnage peut se déplacer à l’aide 3 touches :

* La flèche de droite permet d’aller à droite.
* La flèche de gauche permet d’aller à gauche.
* La barre espace permet de réaliser des sauts.

Pour la spécificité des déplacements : Le personnage peut sauter uniquement sur les murs. Il peut sauter lorsqu’il est en contact avec le haut du mur ou avec un coté du mur. De plus le personnage peut se déplacer avec les flèches même s’il n’est pas en contact avec un mur. Les déplacements peuvent êtres quelque peu perturbants au début et nous aurions pu changer pour nous rapprocher de quelque chose comme Mario de façon très simple, mais nous avons gardé cette méthode car elle me semble être assez unique à notre jeu.

Pour se déplacer le bonhomme peut sauter sur les murs. Il y a deux types de murs :

* Les murs qui restent fixes et qui sont représentés par un rectangle.
* Les plateformes qui sont représentées de la même façon et qui, elles, peuvent bouger d’un bout à l’autre de la map.

## Lancement du jeu

Pour le lancement du jeu, il faut ouvrir un éditeur de texte permettant de compiler du code python donc par exemple *spyder*. Ensuite il faut ouvrir au sein de ce logiciel le fichier Monde\_1.py. Une fois le fichier exécuté, une fenêtre s’ouvre, et l’utilisateur doit appuyer sur le bouton *Start* pour lancer le niveau. Si l’utilisateur perd ou gagne et veut refaire le niveau, il lui faut fermer la fenêtre et réexécuter le fichier.

# 2. Description générale du problème

## 2.1. Les différentes pistes envisagées

Au début il nous a fallu déterminer quelles allaient être les différentes classes que nous pourrions créer. D’autant qu’il s’agit d’un sujet libre qui est assez vaste au premier abord et donc nous ne savions pas vraiment par où commencer. Premièrement on a associé à chaque élément du jeu des caractéristiques qui lui était propre. Nous savions qu’il y allait avoir de la physique et des équations différentielles car nous voulions conserver la particularité des jeux de plateforme qui est l’inertie du bonhomme. En effet lorsqu’on joue à Mario on s’aperçoit que le personnage à des mouvements fluides et qu’il conserve une certaine vitesse si aucune touche n’est pressée. Donc nous avons associer à chaque élément des coordonnées, une vitesse et une accélération. C’est pour cela que nous avons créé une super classe *Elément* pour associer à chaque élément les caractéristiques de bases.

De plus nous voulions que tous les éléments évoluent dans un monde qui a une certaine dimension. Nous avons alors créé une classe *Monde* à laquelle on associe une longueur et une largeur. Comme nous voulions que tous les éléments restent dans le monde (qu’ils ne puissent pas sortir) nous avons créé une fonction setter pour le tuple comportant les coordonnées. Dans cette fonction qu’on appelle à chaque fois qu’on veut rentrer de nouvelles coordonnées, on analyse les coordonnées et on les change pour que le bonhomme reste dans le monde (la map).

Enfin pour déterminer les coordonnées dans le temps nous devions utiliser une fonction qui permet de recalculer l’équation différentielle de la position à chaque instant. Donc on a réalisé une méthode d’Euler basique. Ensuite on a créé deux classes concernant les murs (les murs fixes et ceux qui bougent, les plateformes). Enfin pour jouer il fallait les ennemis et le portail. Les ennemis sont composés de 3 classes.

## 2.2. Les hypothèses réductrices

Pour la physique il nous a fallu faire des hypothèses réductrices :

La géométrie des éléments : nous ne voulions pas avoir une géométrie très complexe à traiter d’autant que nous avions des difficultés à démarrer. Donc on a choisi de représenter tous les éléments par des rectangles. C’est pour cela que chaque élément a une hauteur et une largeur. De plus il nous fallait déterminer une convention pour savoir à quoi correspondent les coordonnées. On a choisi les coordonnées d’un élément telles qu’elles correspondent aux coordonnées de l’angle en bas à gauche du rectangle par lequel est modélisé un élément.

La physique : Premièrement tous les éléments sont représentés dans un repère orthogonal comme représenté ci-dessous.



**X**

**Y**

Pour ce qui est de la physique, nous avons grandement simplifié la chose. On ne peut même plus vraiment parler de Principe fondamental de la dynamique. En effet, on ne parle pas vraiment de force dans notre projet à part celle du poids. Alors que se passe-t-il lorsqu’un élément entre en contact avec un mur alors qu’il a une certaine vitesse ? C’est pour cette question que nous avons simplifier la physique. On ne voulait pas avoir à traiter la partie énergétique lorsqu’il y a un choc avec un mur ou autre. C’est pour cela que le setter des coordonnées d’un élément est très important car c’est vraiment là que réside toute l’intelligence de notre jeu. En effet, pour chaque coordonnée calculée par la méthode d’Euler on vérifie directement si le personnage touche un mur ou autre et si c’est le cas on traite le cas. Par exemple, si le bonhomme est en contact avec un mur on considère que son accélération et sa vitesse dans le sens de la direction du mur sont nuls.

Par rapport à la gravité, on considère que l’accélération sur Y est toujours égale au poids. Sauf lorsque le personnage touche le dessus du mur alors l’accélération vaut 0. Bien sûr dans la fonction qui s’exécute à chaque fois qu’on appelle les coordonnées on vérifie bien que le bonhomme ne rentre pas dans un mur ou autres problèmes.

Pour ce qui est de la dernière mise à jour du jeu avec les missiles ; dans le but de repérer le bonhomme par rapport au missile, on détermine l’angle que fait la droite passant par le bonhomme et le missile et l’axe X, cela donne la direction. Donc le missile se dirige instantanément dans la bonne direction. Il est vrai que l’on perd un peu l’aspect calibrage du missile sur le bonhomme qui est souvent imparfait dans les jeux habituels et qui permet une mobilité plus facile au personnage.

# 3. Description générale du programme

## 3.1. Présentation générale

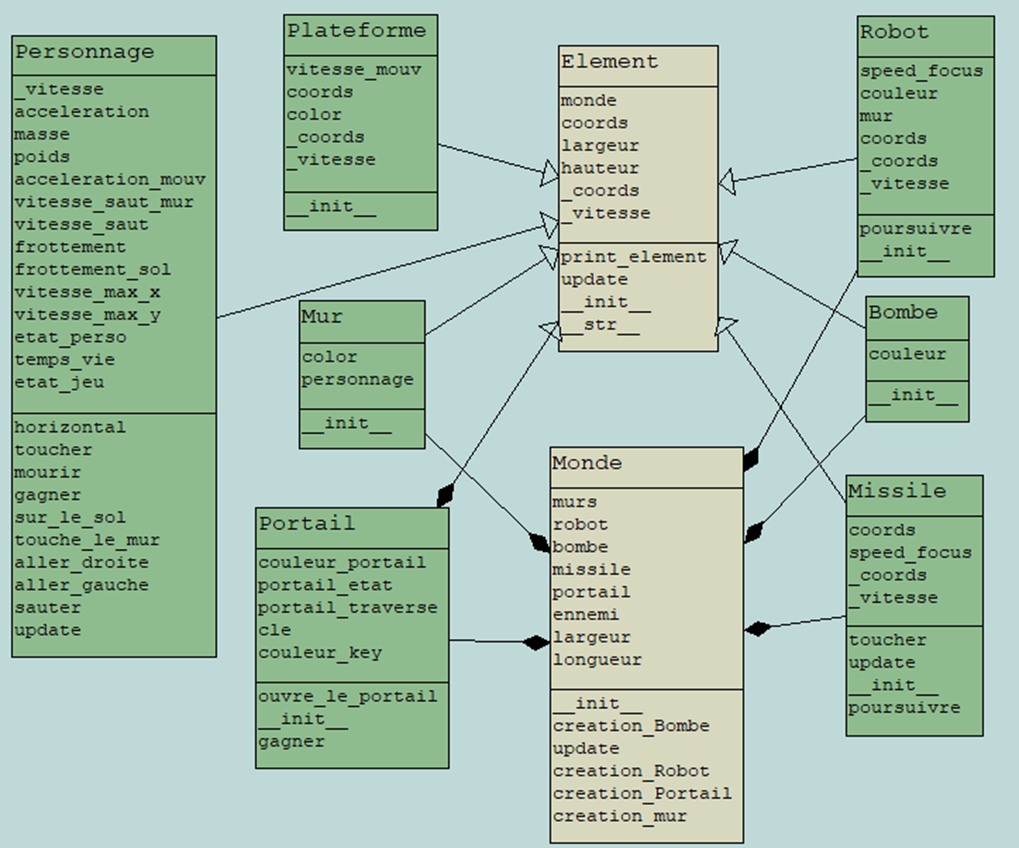


Figure 1 Diagramme de classe

La classe *Element* est la super-classe autour de laquelle toutes les autres classes constituant le monde héritent. Elle contient les variables d’instances du constructeur qui sont communes à tous les éléments. La classe Monde a le but de créer le monde avec tous les éléments nécessaires. Le personnage défini par la classe *Personnage* hérite de *Element*. Elle contient toutes les méthodes pour faire déplacer le personnage dans le monde.

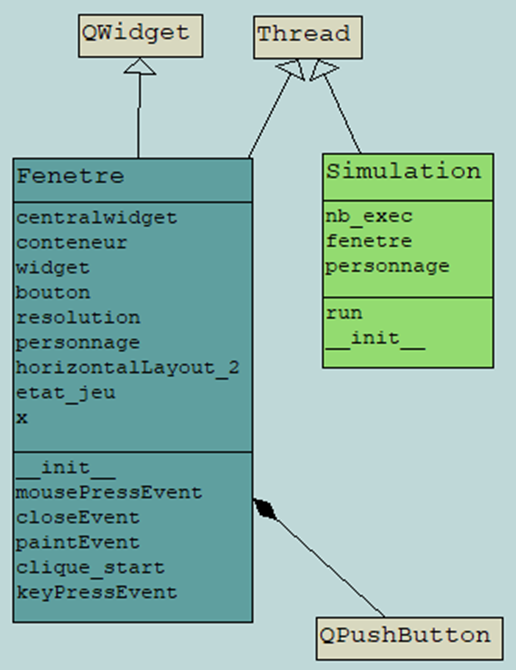


Figure 2 Diagramme de classe

La classe Simulation a pour but de mettre à jour toutes les variables des éléments. La classe Fenetre gère l’espace graphique du jeu. Elles doivent être exécutées en même temps car la fenêtre doit être mise à jour en même temps que la simulation, avec les variables qui changent en fonction des touches du joueur. On a donc choisi de faire hériter ces deux classes de la super-classe Thread, qui permet d’exécuter deux programmes simultanément.

## 3.2. La classe Elément

La classe *Elément* est la super-classe dont tous les éléments spécifiques du monde héritent. Un élément est défini par un monde, ses coordonnées dans le monde (X,Y), sa largeur et sa hauteur. Initialement, chaque élément a une vitesse nulle.

### La méthode *update()* :

La méthode update permet de recalculer les coordonnées de l’élément à chaque instant dt. Elle est fondamentale pour chaque élément ayant un mouvement. Elle consiste à recalculer les coordonnées à l’instant t+dt par la méthode d’Euler basique.

* + 1. Le setter des coordonnées :

La méthode setter des coordonnées consiste à vérifier que l’élément reste bien dans le monde quoi qu’il arrive. Cette méthode est propre à chaque élément. Mais on verra que nous avons utilisé le polymorphisme pour redéfinir cette méthode dans beaucoup de classes.

## La classe Monde

La classe *Monde* contient son constructeur et les méthodes permettant de créer un monde avec tous les éléments. Un monde est d’abord défini comme un rectangle. Ensuite on ajoute les murs et les plateformes dans une liste avant de mettre des ennemis. La liste *ennemi* sera composée des deux listes *bombe* et *robot*.

### La méthode *creation\_mur()* :

La méthode *creation\_mur* permet de placer les murs qu’on souhaite dans le monde. Elle teste le type de mur, c’est-à-dire un mur ou une plateforme, qu’on veut ajouter. Elle rajoute ensuite l’élément dans la liste correspondante.

### Les autres méthodes de création :

La méthode *creation\_bombe()* remplie la liste *bombe* puis rajoute toutes les bombes dans la liste *ennemi*. La méthode *creation\_robot()* ajoute les robots à la liste *robot* puis elle met la liste *robot* dans *ennemi*. La méthode *creation\_portail()* permet de créer un portail et de la mettre dans le monde.

### La méthode *update()*:

Cette méthode permet de recalculer la position des éléments qui se déplacent dans le monde. Les plateformes, les robots et le missile sont concernés. On parcourt alors les listes *murs* et *robot* pour mettre à jour les coordonnées de toutes les plateformes et des robots en faisant appel à la méthode update héritée de la classe *Element*. Les coordonnées du missile sont aussi mises à jour.

## Les classes d’ennemis

### Les classes *Bombe* et *Robot*:

Les deux classes *Bombe* et *Robot* sont des classes filles de *Element*. Elles définissent les particularités d’une bombe et d’un robot. Une bombe est un simple cube rouge et stationnaire (initialement la couleur me servait à différencier les différents ennemis sur l’interface graphique, mais maintenant des images le font beaucoup mieux). Un robot est stationnaire et se déplace en direction du personnage dès qu’il est sur la même horizontale.

* + 1. Les méthodes de la classe robot :

Comme on peut le voir, la classe robot contient deux méthodes : le setter des coordonnées et poursuivre.

Dans cette classe on utilise le polymorphisme pour redéfinir la méthode setter de façon à ce que le robot reste tout le temps sur le mur et que lorsqu’il arrive en fin de mur le robot s’arrête pour ne pas tomber. En plus on conserve dans cette méthode le bout de code du setter d’élément qui permet de rester dans le monde quoi qu’il arrive. Normalement si on crée correctement les murs il n’y a pas de raison que cette partie du code serve, mais c’est une sécurité.

Pour ce qui est de la méthode *poursuivre()*, elle prend en argument le côté duquel se trouve le personnage par rapport au robot et elle permet d’initialiser la vitesse du robot vers le personnage. Cette méthode ne peut s’exécuter que dans le setter de personnage. En effet c’est dans le setter des coordonnées de *Personnage* que je vérifie où sont les robots par rapport au personnage. Et dans le setter on vérifie que le personnage passe sur le même horizon que le robot et si c’est le cas la fonction poursuivre du robot est exécutée.

### 3.4.2. La classe Missile :

Cette classe initialise d’abord le missile avec le constructeur. Les méthodes *poursuivre()* et *update()* permettent le mouvement du missile dans la direction du personnage.

Pour la méthode *poursuivre()*, de la même façon que la méthode *poursuivre()* de robot elle est exécutée ou non dans le setter des coordonnées de personnage. Mais cette fois, elle prend en argument l’angle que fait la droite passant par le robot et le personnage avec l’axe des abscisses. Elle redéfinit aussi la vitesse du missile pour qu’il se dirige vers le personnage.

## La classe Personnage

Cette classe est une classe fille qui hérite de *Element*. Elle contient toutes les instances initiales nécessaires pour le déplacement du personnage et sa taille.

Faisons une petite parenthèse pour ce qui de l’ordre d’instanciation :

1. On instancie un monde avec les dimensions qui lui sont propres
2. Ensuite on instancie le personnage qui prend en argument le monde qu’on vient de créer.
3. Enfin on remplit le monde avec les fonctions création murs ….

Le personnage et le monde « échangent » en permanence. Comme dit précédemment, la force du jeu est le setter de personnage. Dans cette fonction qui est appellé à chaque fois qu’on rentre des coordonnées, on vérifie toutes les conditions pour que le jeu fonctionne. Donc encore une fois on a utilisé le polymorphisme pour redéfinir le setter. On conserve le code qui permet de garder le bonhomme dans le monde. En plus on rajoute des bouts de codes qui gèrent chacun l’interaction avec un élément du jeu. Par exemple, pour ce qui est des murs, on a vu que le monde dans lequel le personnage évolue contient des listes où il stocke les éléments. A chaque exécution du setter, la fonction lit la liste contenant les murs. Pour définir l’interaction avec tous les murs contenus dans la liste. La méthode est même pour tous les éléments du jeu.

Pour alléger le code qui est déjà très lourd nous avons créé des fonctions comme *toucher()* qui prend en argument n’importe quel élément et qui ressort True si le personnage touche l’élément et False sinon.

Enfin, parlons de ce qui est du plus important à savoir les déplacements. La méthode est très simple, le personnage peut se déplacer qu’il soit sur un mur ou en l’air. Cette méthode, est appelée *aller\_droite()* et *aller\_gauche()* et lorsqu’elle est appelée elle met la vitesse sur X à plus ou moins la vitesse de déplacement en fonction de la méthode. Pour les sauts l’idée est la même. On vérifie que le bonhomme soit bien sur un mur pour le faire sauter. Aussi, pour garder l’idée des jeux de plateforme, pour les déplacements gauche droite c’est l’accélération qui est modifiée alors que pour les sauts c’est la vitesse que l’on change.

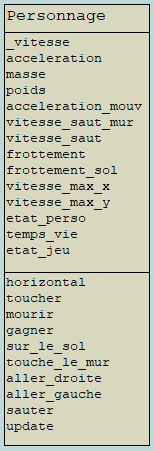


Figure 3 Diagramme de la classe Personnage

## 3.6. La classe Simulation

Cette classe a pour but de mettre à jour toutes les variables des éléments du monde. D’abord on initialise les variables d’instance dans le constructeur. Thread.\_\_init\_\_(*self*) permet bien faire hériter *Simulation* de *Thread*. Le nombre d’exécution nb\_exec est initialisé à 0. On met à jour les variables seulement si nb\_exec est nul. S’il passe à 1, cela signifie que l’utilisateur a terminé sa partie donc on n’a plus besoin d’exécuter la simulation.

La méthode *run()* est le centre de cette classe. On initialise les deux variables de temps. Quand l’état du jeu n’est pas en mode « off », on met à jour toutes les 0.03 secondes les coordonnées du personnage en appliquant *update()* puis les coordonnées de tous les éléments du monde ( les robots, le missile par exemple). On teste les cas où la simulation s’arrête c’est-à-dire s’il gagne ou s’il meurt. Dans ce cas, on stoppe la simulation grâce à break.

## 3.7. La classe Fenetre

Cette classe fait l’interface graphique du jeu. Elle doit s’exécuter en même temps que la classe *Simulation* car la fenêtre doit être mise à jour dès qu’un élément bouge dans le monde. Le constructeur assure d’abord le bon héritage de *QWidget* pour l’interface graphique et de *Thread*. Comme le repère de la fenêtre n’est pas le même que le repère choisi pour les coordonnées des éléments du monde, on a converti les coordonnées pour qu’elles soient correctes dans la fenêtre. On a alors pris une résolution de 10 à appliquer à toutes les coordonnées dès qu’on les utilise dans la fenêtre. L’état du jeu est initialement « off ».

### 3.7.1. La méthode *clique\_start()* :

Afin de pourvoir démarrer la simulation, l’état du jeu doit passer en mode « on ». Cette méthode permet de passer l’état du jeu en mode « on » dès que le joueur appuie sur le bouton Start. Quand l’état du jeu est sur « on », on doit désactiver l’action du bouton sinon les commandes du joueur ne sont pas prises en compte.

### 3.7.2. La méthode *paintEvent()* :

Cette méthode permet d’afficher pour chaque mise-à-jour les bons événements. On affiche en premier le personnage dans la fenêtre. Les trois boucles *for* permettent de convertir les coordonnées des murs, plateformes, robots et bombes dans le repère de la fenêtre afin d’afficher les éléments. Chaque boucle dessine les murs et les ennemis. On fait après la même chose pour le missile seul.

On sépare quatre événements différents grâce aux *if* :

* Lorsque l’état du portail est « open » : on dessine le portail car cela signifie que le joueur n’a pas encore toucher la clé.
* Lorsque l’état du portail n’est pas « open » : on dessine la clé jusqu’à ce que le joueur la touche.
* Lorsque le personnage meurt : on affiche une image par-dessus le monde informant la mort du joueur et la fin de la simulation

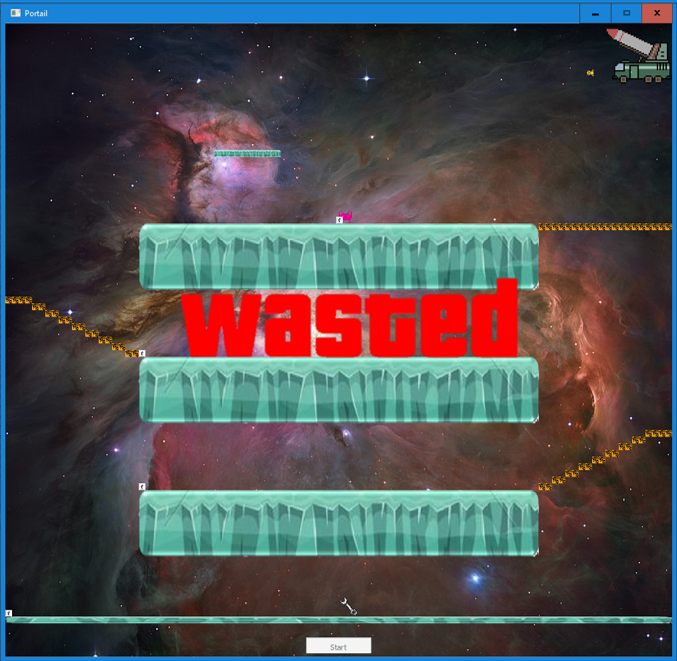


Figure 4 Image de l'échec du joueur

* Lorsque le personnage gagne : on affiche une image par-dessus le monde informant la réussite du jeu et la fin de la simulation.



Figure 5 Image de la réussite du joueur

### 3.7.3. La méthode *keyPressEvent()* :

Cette méthode permet d’associer les touches du clavier à une action du personnage. Quand l’utilisateur appuie sur la barre d’espace, le personnage saute en appelant la méthode *sauter()* de *Personnage*. On associe les touches flèche-droite et flèche-gauche aux mouvements de déplacement vers la droite et la gauche respectivement.

# Conclusion

## Tests effectués

Afin de vérifier le bon fonctionnement des classes et des méthodes, nous avons réalisé des tests unitaires simples. Nous vérifions d’abord que les classes principales sont bien initialisées. Ensuite, les méthodes ayant un rôle fondamental sont testées. Il n’y pas eu d’exception à vérifier.

## Limitations

D’après la complexité du jeu, nous nous sommes d’abord réduits aux ennemis principaux. Les bombes et les robots sont les ennemis basiques du jeu. Nous avons donc d’abord créé les deux classes Bombe et Robot. De même pour le monde. Nous avons choisi de mettre des murs et des plateformes avec un mouvement horizontal.

## Perspectives et améliorations

Comme l’idée de notre jeu vient d’un jeu existant sur la console portable PSP, on aurait voulu rajouter d’autres éléments dans la carte tels que des ennemis plus difficiles à éviter. Nous avons finalement pu mettre un missile suivant le joueur.

Nous aurions pu aussi faire des mondes de différents niveaux et avec des graphismes différents. Nous pourrions ajouter un menu où le joueur pourrait choisir en cliquant avec la souris les mondes auxquels il veut se confronter. Il faudra qu’il passe le portail d’un monde pour avoir une fenêtre qui s’ouvre avec le menu.

Dans le but de faciliter la compréhension du jeu et la méthode à employer pour passer le monde, nous aurions voulu mettre au point une démonstration pour le joueur. Dans notre cas, le monde n’est pas très difficile. Avec la perspective d’inventer de nouveaux mondes plus compliqués, le joueur pourrait vouloir une explication de la solution pour passer le niveau. Nous avons une idée précise de la réalisation de cette démonstration. Il faudrait créer une nouvelle classe *Scanner* regardant la date et la touche appuyée à chaque instant. On crée une liste avec des couples comportant le temps écoulé à partir de t0 et la touche appuyée à cet instant. Le premier couple contient alors le temps initial t0 et la première commande. Nous jouons alors au jeu et nous gagnons pendant que les mouvements sont scannés. Cette liste est affichée puis copiée dans un fichier correspondant à la démonstration. Le joueur pourra ainsi ouvrir une fenêtre pour regarder une version gagnante du niveau.

Dans notre version, en cas d’échec du joueur, la fenêtre se ferme et il doit exécuter une nouvelle fois le programme pour rejouer. Nous aimerions alors ajouter un bouton restart pour que le joueur puisse directement réessayer.