SuperEinstein

Forgione David

21.06.2020

Table des matières

1 Introduction

Super Einstein est un jeu développé avec Qt c++, c'est un jeu de plate-forme, le but est donc simplement de traverser toute la carte vers la droite en évitant les ennemis et les trous en gardant toutes ses vies. Une fois à la fin et tuer Newton le jeu redémarre. Les classes virtuelles ont été développées de tel manière qu'elle ne soit pas restreintes à ce jeu-ci mais peuvent être exportées pour d'autres utilisation. Les classes non virtuelles sont elles restreintes à ce projet, et sont plus difficilement exportables pour d'autres applications, dans la suite il sera présenté comment utilisé ces classes.

2 Cahier des charges et Git

Le premier cahier des charges a été modifié avec le temps pour se transformer en SuperEinstein. Les objectifs a atteindre ont étés :

- ☐ V0.0 Le personnage peut marcher en avant et en arrière.
- ☐ V0.5 Le personnage peut sauter et courir, des trous sont présents au sol.
- ☐ V1.0 Obstacles et différents étages
- ☐ V1.5 Degrés de difficultés différents
- ☐ V2.0 bots présents
- ☐ V2.X améliorations graphiques
- ☐ V3.X Newton à battre

2.1 Histoire et commandes

Le jeu parle d'Einstein qui doit battre Newton pour que sa théorie de la relativité soit validée, mais Newton refuse, il est persuadé que sa théorie classique de la physique est la seule bonne théorie et envoie donc ses sbires pour l'en empêcher.

Voici la liste des commandes :

— a : aller à gauche

— d : aller à droite

— espace : sauter

— A : courir à gauche

— D : courir à droite

Il est impossible d'appuyer 2 touches en même temps, pour changer de touche il faut relâcher la touche pressée.

2.2 Git et suivi

Pour voir l'historique des versions et du codage il est aussi possible de visualiser le graphe GIT fait grâce à GitLab. Une partie de l'historique est présenté à la figure Fig : 1 .

3 Fonctionnement et Principes

Le projet est divisé en 3 sections, les objets qui représentent toutes les classes des éléments du jeu qui sont eux-mêmes divisés en deux, les personnages qui sont les objets qui peuvent bouger et les éléments qui sont des objets immobiles. Il y a aussi la section des vues qui permet l'interaction du joueur avec le jeu, celle-ci aussi est divisée en 3 parties, l'affichage (OutputView),

FIGURE 1 – Git graph history

la lecture des entrées (InputView) et le menu (Menu) qui affiche un menu interactif avec le jeu. La dernière section comporte le contrôleur qui s'occupe d'exécuter les commandes adéquates au bon fonctionnement et les données (Data) qui enregistrent toutes les données relatives au jeu. La figure Fig :2 montre le lien entre toutes les classes.

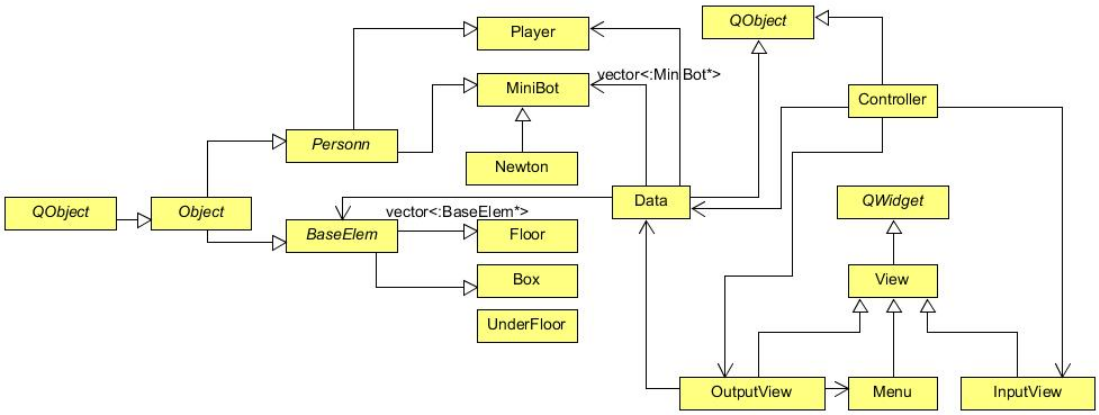


FIGURE 2 – UML class diagramms of all bonds between class.

Chaque classe est présentée dans la suite avec une explication de ses buts et de son fonctionnement.

3.1 Object

Object est la classe mère qui définit la base d'un objet. Cette classe hérite de la classe Qt *QObject*. La figure Fig :3 montre toutes les classes qui héritent de celle-ci, leurs liens et leurs méthodes. Pour initialiser un *Object* il faut lui donner une position de type *QPoint* comme paramètre à son constructeur.

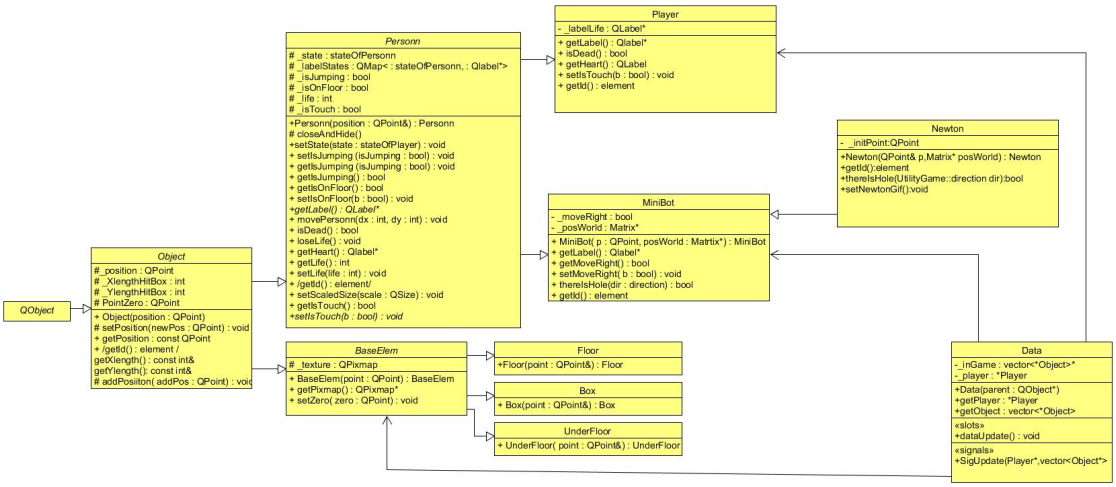


FIGURE 3 – Object family class

Attributs : Une classe objet possède 4 attributs qui sont protégés pour que ses filles puissent accéder à ces valeurs.

- *_position* de type QPoint contient la position de l'objet.
- *_X/YlengthHitBox* contiennent respectivement la longueur en x et en y de la Hit Box, ces valeurs sont ensuite utilisées pour calculer l'occurence d'une collision.
- *PointZero* de type Qpoint est un QPoint qui permet de calibrer le zéro du monde.

Méthodes

- *void setPosition(Qpoint newPos)* permet de définir une nouvelle position à l'objet.
- *const QPoint getPosition()* renvoie un *QPoint* constant qui représente la position et est égal à l'attribut *_position*.
- *element getId()* retourne l'id de l'objet, *Object* est une classe virtuelle et ne retounr donc pas d'id, son id vaut nullptr. Un objet qui n'est pas virtuelle renvoie un type *element*(Plus d'information voir Sec : 3.5) qui représente sa classe, ceci permet de différencier les classes lorsque la polymorphie est utilisée.
- *const int& getXlength()/getYlength()* renvoient respectivement la *_XlengthHitBox* *_YlengthHitBox* qui permet de connaître les dimensions de la hit box utiles pour gérer les collisions.
- *void addPosition(QPoint addPos)* permet de faire une translation de l'objet d'une valeur de *addPos*, cette fonction est surtout utilisée pour faire bouger les personnages.

3.1.1 Personnage

Personn est une classe fille de *Object* qui est elle-même virtuelle et qui est la classe mère de toute les classes qui bougent. La particularité de cette classe est qu'elle possède un fichier gif en texture, chaque gif représente un état de la personne, ces gifs doivent être initialisés dans le constructuer de chaque classe fille. Chacune des filles l'initialise et il est impossible de le changer, chacune a un ensemble de gif immuable. Ces gifs correspondent à des états qui sont contenus dans une énumération dans le *namespace UtilityGame* (voir section 3.5) appelée *stateOfPlayer*. Tous les personnages ne sont pas obligés de configurer tous les états mais ils doivent insérer un gifs pour chaque état voulue ou il y a risque de charger un élément qui ne fonctionne pas. Chaque personne doit entrer un QPoint comme position de départ à l'instar du constructeur de Object.

Les gifs sont stockés dans la classe Qt *QMovie* et sont enrengistrés dans la classe à l'aide d'une map de *QLabel* contenant chacun la vidéo associée, cette map a une relation *stateOfPlayer* et *QLabel* et fait donc la relation entre l'état et la vidéo correspondante. Ainsi en changeant l'état grâce à la méthode *setState* le bon *QLabel* sera donnée lors de l'appelle de *getLabel*.

Attributs : En plus des attributs de *Object, Personn* possède 6 attributs protégés.

- *_state* de type stateOfPersonn contient l'état de la personne, c'est à dire s'il marche, saute etc...
- *_labelStates* de type QMap<stateOfPersonn,QLabel> est la relation faite entre les états et les vidéos correspondantes.
- *_isJumping* de type booleen est indique si le personnage saute
- *_isOnFloor* de type booleen est indique si le personnage est sur un objet
- *_life* de type int contient le nombre de vie à disposition.
- *_isTouch* de type booléen indique si le personnage a été touché par un ennemei.

Les attributs *_isJumping*,*_isOnFloor* et *_isTouch* doivent être gérer par les utilisateurs de la classe (dans ce projet c'est Controller) à l'aide des méthodes correspondantes.

Méthodes :

- *void closeAndHide()* est une méthode protégée qui permet, lors d'un changement d'état de stopper les vidéos non utilisées dans la map et de cacher les labels correspondants. De cette manière seul ce qui est besoin occupe de la mémoire.
- *void setState(state stateOfPlayer)* permet de changer la variable *_state* qui représente l'état de la personne.
- *void setIsJumping(bool isJumping)* permet de changer la valeur de *_isJumping* et d'indiquer qu'un saut est en train de se faire.
- *bool getIsJumping()* renvoie la valeur de *_isJumping* et permet de savoir s'il y a un saut en cours.
- *bool getIsOnFloor()* renvoie la valeur de *_isOnFloor* et permet de savoir s'il y a un objet sur lequel la personne est posée.
- *void setIsOnFloor(bool b)* permet de changer la valeur de *_isOnFloor* et d'indiquer que le personnage se trouve sur un objet.
- *QLabel* getLabel()* renvoie un pointeur sur le QLabel correspondant à l'état de la personne, c'est à dire correspondant à *_state*. Cette méthode appelle toujours *void closeAndHide()* pour fermer les autres QLabel.
- *void movePersonn(int dx,int dy)* fait essentiellement la même chose que *addPosition(QPoint addPos)* mais chaque coordonné de translation peuvent être entrées séparément.
- *bool isDead()* renvoie *true* si le personnage n'a plus de vie, c'est à dire que *_life* est plus petit que 1.
- *void loseLife()* décrémente *_life* de un.
- *QLabel* getHeart()* renvoie un pointeur sur un QLabel correspondant au signe de la vie, chaque coeur représente une vie.
- *int getLife()* renvoie le nombre de vie
- *void setLife(int life)* permet de changer la valeur de vie du personnage, cette méthode est appelée seulement au début et au reset du jeu.
- *element getId()* comme *Object* renvoie nullptr
- *void setScaledSize(QSize scale)* permet de modifier les proportions des QLabel et des QMovie lorsque la taille totale de l'affichage a changée. QSize a 2 valeurs, une pour indiquer ce changement sur l'axe x et une sur l'axe y. Une valeur < 1 indique une résuction alors qu'une valeur > 1 indique un agrandissement, 1 signifie aucun changement.

- *bool getIsTouch()* retourne la valeur de *_isTouch* et donc *true* s'il y a eu contact.
- *void setIsTouch(bool b)* permet de changer la valeur de *_isTouch*.

  _isTouch n'affecte pas la classe *MiniBot* alors que pour *Player* elle va permettre de lancer un chrono qui empêchera de perdre de la vie lors de contact avec des *MiniBot* et affiche un point d'exclamation sur le joueur pour indiquer qu'il est dans cette phase.

Player

Player hérite de *Personn* est représente super Einstein le personnage joué par le joueur. Cette classe revoie comme Id *Einstein*. Elle a comme particularité de gérer la variable *_isTouch*, elle permet donc de savoir si le joueur a été touché. Cette variable correspond à *true* seulement si c'est un bot qui le touche par le côté, si c'est le cas le joueur perd une vie et il ne peut plus être touché pendant 5 secondes. Un point d'exclamation indique cet état (Fig : 4).

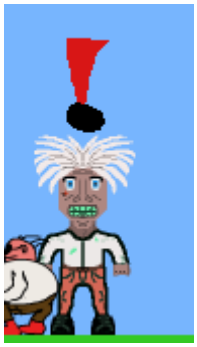


FIGURE 4 – Exclamation point when collision with a bot

Einstein possède 6 états différents, tous éléments de l'énumération *stateOfPersonn*. Les différentes images associées aux états sont présentée à la Fig : 5 . Seulement une direction est présentée car de l'autre sens c'est juste un miroir du gif.

MiniBot

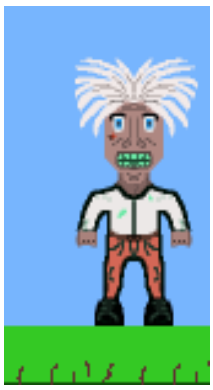
Les Bots sont les ennemis du jeu, ils avancent seulement de gauche à droite en changeant de direction s'il y a un trou, c'est le controller qui gère le mouvement des bots. Ils ont comme particularité de reconnaître un trou grâce à la méthode *thereIsHole(direction d* qui prend en paramètre une direction (gauche ou droite) et qui renvoie *true* s'il y a un trou. La détection de trous et d'autres objets se fait grâce à la matrice du monde passé en paramètre dans le constructeur et l'enregistre comme attribut du MiniBot. Plus d'explication sur la matrice dans la section 3.4.

Newton Newton fonctionne comme les minibots mais à une fonction *setNewtonGif()* pour permettre d'initialiser les gif de Newton.

Les *MiniBot* n'ont que trois états, marche gauche ou droite et mort. Sur la figure Fig : 7 les différents états sont montrés.

3.1.2 Element de base

BaseElem est la classe de base pour tout objet fixe. A la différence de son homologue *Personn*, il n'a pas de *QLabel* mais un *QPixmap* qui permet de sauvegarder une image, comme l'objet ne bouge pas il n'est pas nécessaire de placer un gif, qui prendrai plus de mémoire. Les classes filles sont ***Floor,Box, UnderFloor*** qui représentent les différents éléments fixes du jeu, la figure Fig : 7 montre une image en jeu de chaque élément. Toutes ses classes ont les mêmes caractéristiques, la seule chose qui change est l'image sauvegardée.



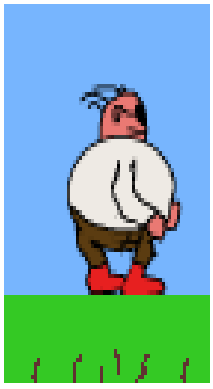
(a) Stay gif state.

(b) Jump state gif

(c) Walk to right gif.

(d) Jump to right gif.

FIGURE 5 – Sates gif of Einstein.



(a) Walk gif .

(b) Dead state gif

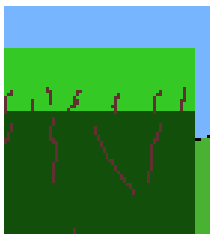
FIGURE 6 – Sates gif of Minibot.

3.2 View

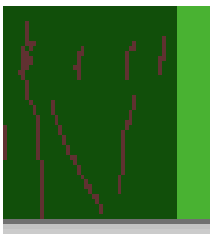
3.2.1 Affichage

La classe *OutPutView* se charge de l'affichage de tout ce qui est à l'écran, il reçoit l'affichage de la part du Controller pour mettre à jour l'affichage et lit les données stockées dans *Data*. Le diagramme UML de la Fig : 8 représente la séquence d'affichage.

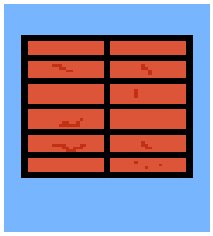
Comme il est possible de le voir, les données sont toutes recherchées dans *Data*. Deux variables *minDisplay* et *maxDisplay* permettent de savoir quel objet il faut afficher et où . Car Einstein est toujours affiché à 150 px du bord gauche en x, donc lorsque la position des objets est récupérées il faut faire une translation de celui-ci car il faut afficher l'objet relativement au joueur d'où la nécessité du calcul de ces variables. Les seuls cas où cela n'arrive pas c'est



(a) floor Image.



(b) under the floor image.



(c) box image

FIGURE 7 – Images of differents *BaseElem*.

lorsque les bords sont atteints (gauche ou droite), dans ce cas Einstein peut toucher les bords et l'image de fond ne fait plus de translation, à la sortie du bord droite le jeu se termine. La position en y n'est jamais translatée. Les seuls objets affichés sont ceux dont la distance sut l'axe x respecte *minDisplay*<Coordonnée en x de l'objet<*maxDisplay*, l'objet est ensuite dessiné à sa coordonnée en x moins *minDisplay*. Le changement de taille de la fenêtre est prise en compte, et pour chaque redimension de l'écran la classe recalcule la taille des objets en divisant la taille actuelle de l'affichage par la taille initiale, on obtient ainsi un chiffre entre 0 et 1 pour chaque coordonnée et il devient facile de redimensionner le tout.

Cette classe fait aussi le lien avec le menu qui est un de ces attributs.

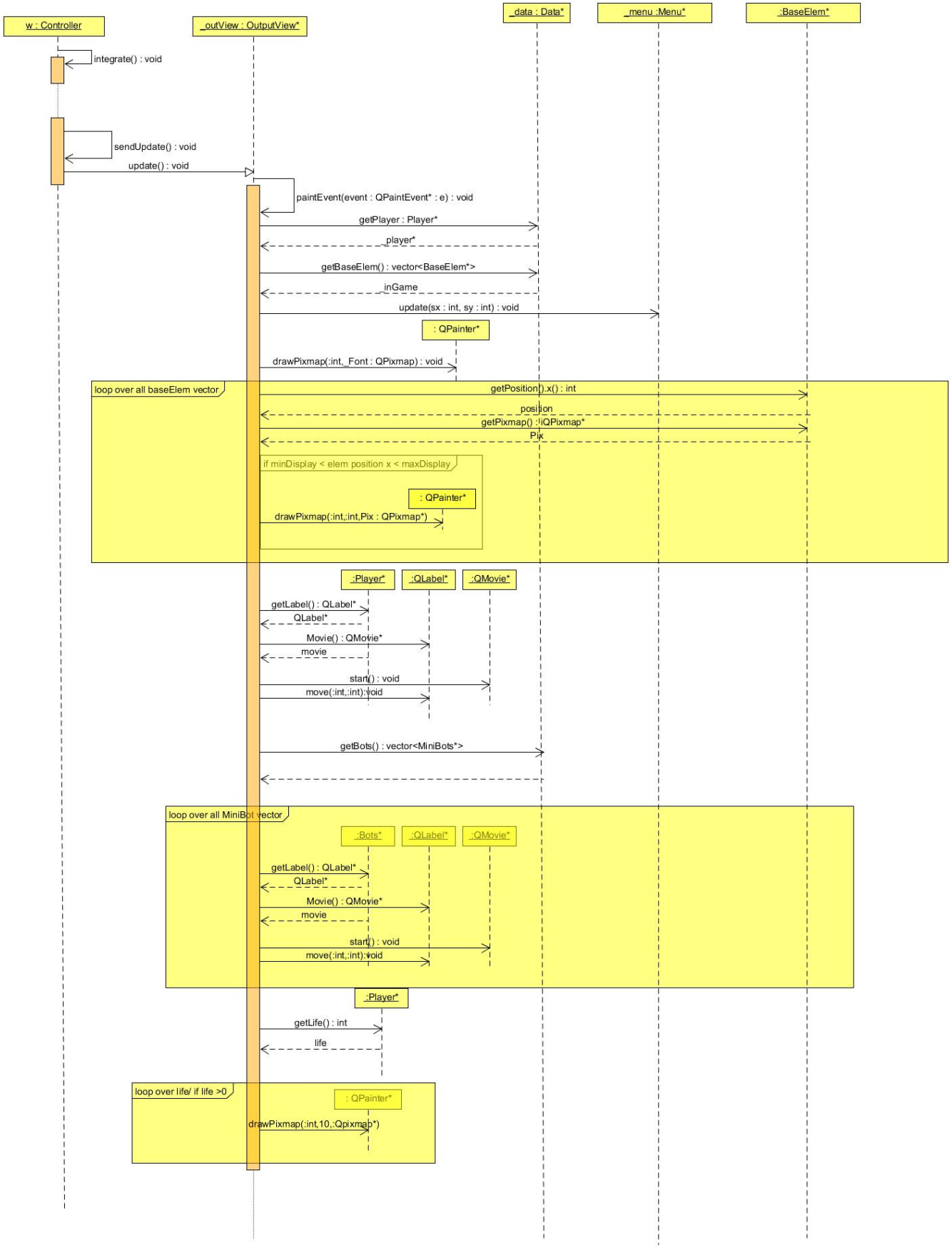


FIGURE 8 – Sequence diagramm of an *OutPutView*.

3.2.2 Entrées

Cette classe gère la pression des touches du clavier et les liens au type *UiEvent* (plus d'explication sec : 3.5) et envoie un signal avec la valeur correspondante au Contrôleur. Si la touche clavier pressée ne correspond à rien alors aucune donnée n'est envoyée. La figure Fig : 9 montre un diagramme UML qui représente ce qui se passe lors de la pression d'une touche.

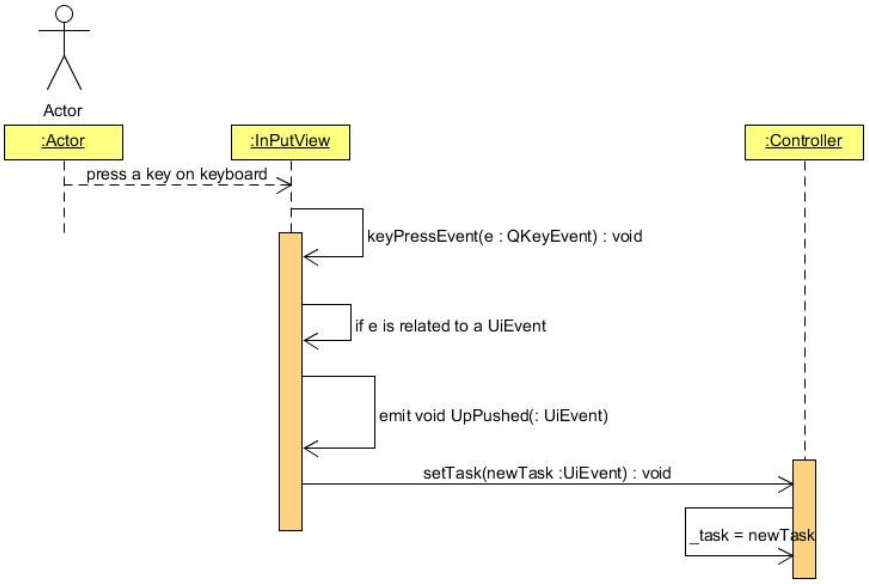


FIGURE 9 – UML sequence Diagramm when a key is pressed.

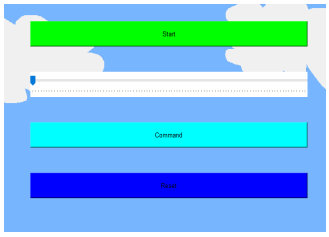
3.2.3 Menu

Le menu permet une interaction du joueur avec des boutons. Il permet de mettre le jeu en pause et de le relancer. Une liste des commandes est aussi visualisable en appuyant sur le bouton *list command* et un bouton reset pour redémarrer le jeu.

La partie du choix de la difficulté disparaît une fois qu'une des propositions a été sélectionnée, pour changer de difficulté il faut appuyer *Reset*.

Le slider est un peu particulier, il permet de changer le volume de la musique en envoyant un signal *volume(int)* au controller avec la valeur de la position du slider qui peut être entre 0 et 100. Il s'affiche toujours, lorsque le menu est quitté le slider s'affiche en haut à droite de l'écran, de cette manière il est toujours possible de manipuler le volume.

(a) menu before difficulty is choosed.



(b) menu in game and after difficulty is choosed

(c) slider

FIGURE 10 – Caption

3.3 Contrôleur

La classe *Controller* est le cerveau du jeu, il fait le lien entre les classes et calcule les changement à faire. Le contrôleur se charge de mettre à jour tous les objets, pour cela il possède une minuterie de type *QTimer** qui démarre dès que le jeu est lancé, et chaque 5 ms la minuterie émet un signal *timeout()*. C'est ensuite le slot *integrate()* du controlleur qui est appelé et qui exécute les tâches.

La variable *_task* du controleur est de type *UiEvent*, cette variable enrengistre l'évènement clavier a réaliser lors du prochain timeout. Cette tâche est lue lorsque *integrate()* fait appel à *executionOfTask()*.La figure Fig : 11 montre un diagramme de séquence UML qui représente ceci dans les cas le plus simples, à savoir lorsque le joueur ne tombe pas, qu'aucun bot n'est mort, que le joueur n'est pas à la fin du jeu et qu'il a encore des vies. Sur la figure Fig : 13 on peut voir un diagramme de classe détaillé avec les vues, le controller et les Data ainsi que leur relations.

La figure Fig : 12 montre comment les tâches sont exécutées chaque timeout. Ici l'exemple montre le cas lorsque le joueur doit se déplacer vers la gauche, c'est à dire que la valeur de *_task* vaut *UiEvent : :a*. Pour bouger Einstein il va ensuite falloir regarder si cela est possible grâce à la fonction *PersonnCanMove(Personn*p, UiEvent task)* qui renvoie une valeur booléenne pour signifier si l'action est faisable, si oui alors la personne est bougée et le label correspondant à son état est fixé. Dans ce cas *PersonnCanMove* renvoie *true*.

3.4 Data

Data concentre toutes les données des objects du jeu. Elle possède deux vecteurs, un qui contient des pointeurs sur *BaseElem*, ici c'est le polymorphisme qui est utilisé pour savoir si un élément du vecteur est plutôt une *Floor* ou *Box* etc.., et un des pointeurs sur *MiniBots*. Elle a aussi un pointeur de *Player* qui représente Einstein.

La méthode *init(diificulty d)* permet de générer un monde en rentrant une des trois difficultés disponibles. Chaque monde est enregistré dans un .txt qui contient des nombres entiers entre 0 et 3, chacun de ces nombres correspond à un type *element* soit à 0 : *Hole*, 1 : *UnderFloor*, 2 : *Floor* et 3 : *Box*. Chacun de ces éléments est un carré de 50 px de côté, donc le nombre à la collone c et la ligne l représente un élément dont l'angle supérieur gauche se trouve à la coordonné $(c \cdot 50, l \cdot 50)$, en lisant de cette manière les matrices la méthode init() initialise les niveaux. Le résultat de ceci est présenté à la figure Fig : 14. Un diagramme de séquence est

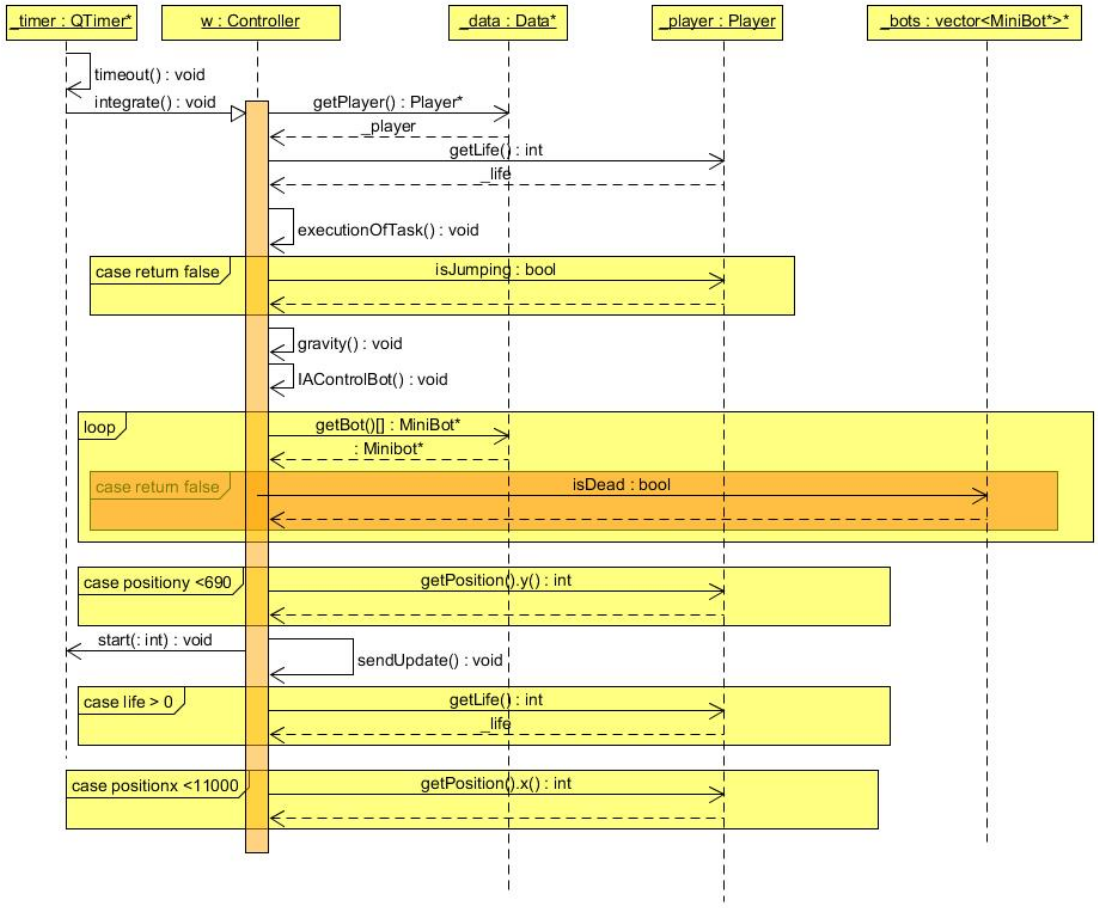


FIGURE 11 – Integrate()

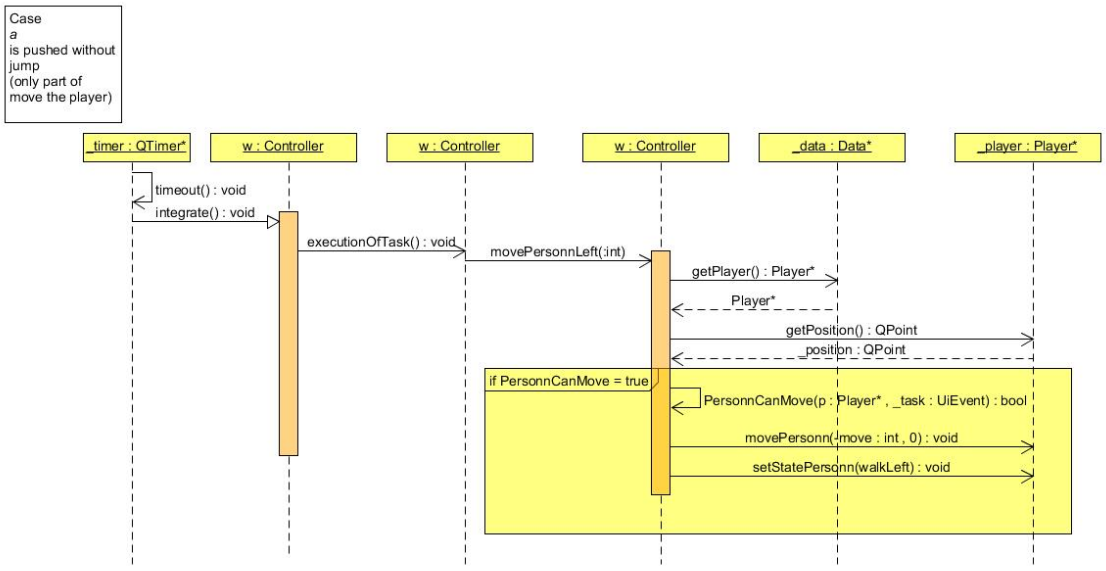


FIGURE 12 – executionOfTask()

présenté à la figure Fig : 15.

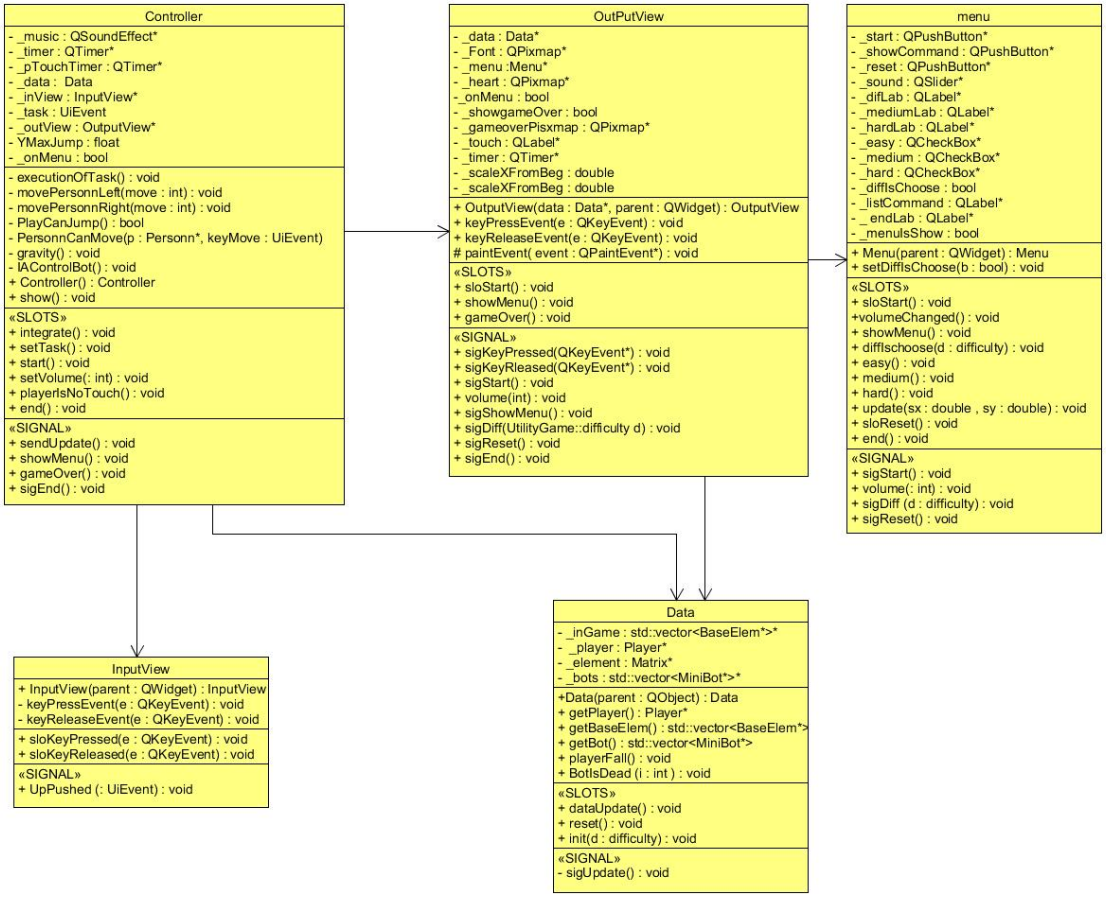
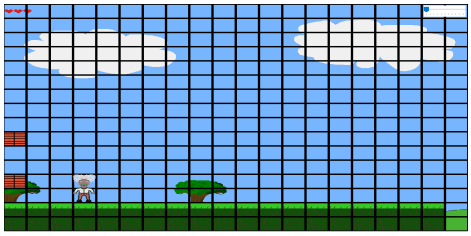
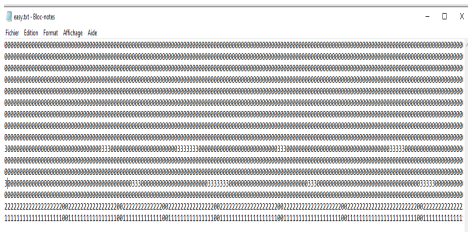


FIGURE 13 – UML class of controller data and views.



(a) Game with grid each 50px.



(b) txt with the number between 0 and 3.

FIGURE 14

3.5 Namespace *UtilityGame*

Le namespace *UtilityGame* regroupe toutes les énumérations utiles pour le jeu ainsi que la classe Matrice. La figure Fig : 16 montre ce namespace.

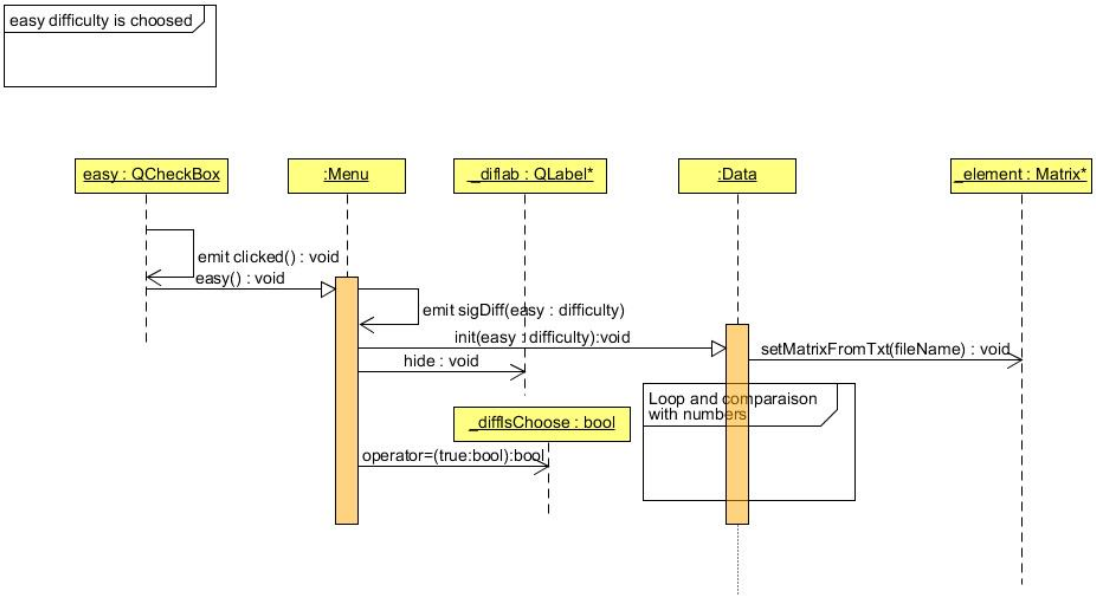


FIGURE 15 – Easy level creation, without the part where bot is created.

4 Tests et résultats

Cette section explique les tests faits ainsi que les résultats.

4.1 Object

4.1.1 Personnage

test Collisions

Les collisions doivent être faites lorsque le personnage touche n'importe quel objet, de plus Einstein doit perdre une vie lorsqu'il tombe dans un trou ou touche un bot. Les Bots ne doivent jamais aller dans un trou. La figure Fig : 17 montre les hitBox de chaque personnage, de cette manière il a été possible de tester les collisions et de s'assurer qu'elles ont bien lieu. Sur la figure Fig : 17a on teste si les personnages interagissent entre eux,Einstein se fait toucher et le point d'exclamation s'active de plus il ne reste que deux coeurs en haut à gauche, donc il a bien perdu une vie.Sur la figure Fig : 17b on voit que Einstein ne peut pas dépasser les objets, sur la figure Fig : 17c Einstein tombe effectivement dans un trou et sur la figure Fig : 17d le bot ne va pas dans le trou et il se retourne sur Fig : 17e. Les tests sont donc réussis pour ces cas là.

4.2 View

Les tests effectués pour la vue sont le changement de taille de la fenêtre et l'affichage correcte d'Einstein lorsqu'il est sur les bords.

changement de taille

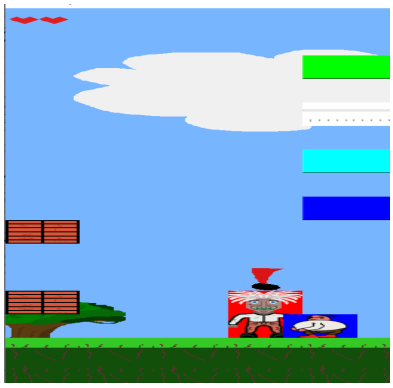
La figure Fig : 18 montre le changement de taille de l'écran d'affichage, sur la Fig : 18a la grandeur initiale et sur la Fig : 18b la taille modifiée. Les tests montrent que cela fonctionne très bien et que les proportions sont gardées.

Bord gauche et fin à droite

Il faut s'assurer qu'Einstein ne puisse pas sortir à gauche de la plateforme et de plus il ne doit pas rester à la position 150px en x lorsqu'il est au bord gauche ou droite, c'est à dire que

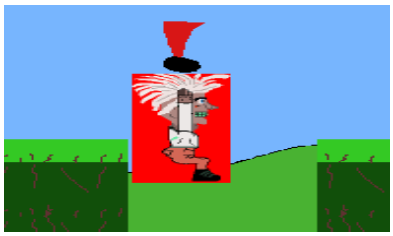
FIGURE 16 – *UtilityGames* namespace

sa corrdonnée en <0 et lorsqu'il passe au-delà de la grandeur maximum une fenêtre indiquant la fin du jeu s'affiche. La figure Fig : 19a montre que le bord gauche ne peut être franchi, la Fig : 19b le montre pour le bord droite et la Fig : 19 montre la fenêtre de fin quand newton est mort.

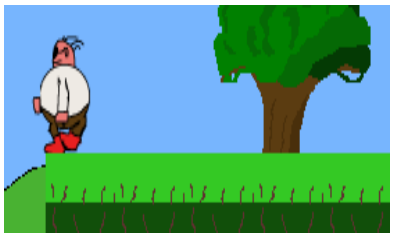


(a) Collisions between Einstein and bot.

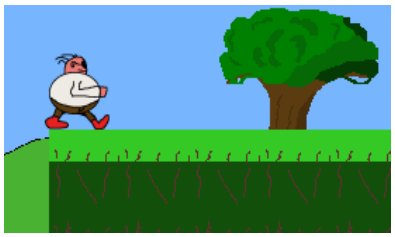
(b) collision between Player and a BaseElem.



(c) Einstein fall.



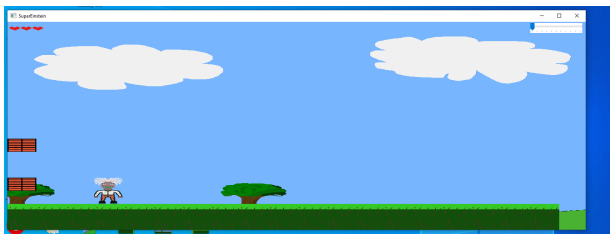
(d) Bot does not fall in a hole.



(e) bot walk in other side when he meet a ho

FIGURE 17 – Test personn collisions and fall

(a) Initial size



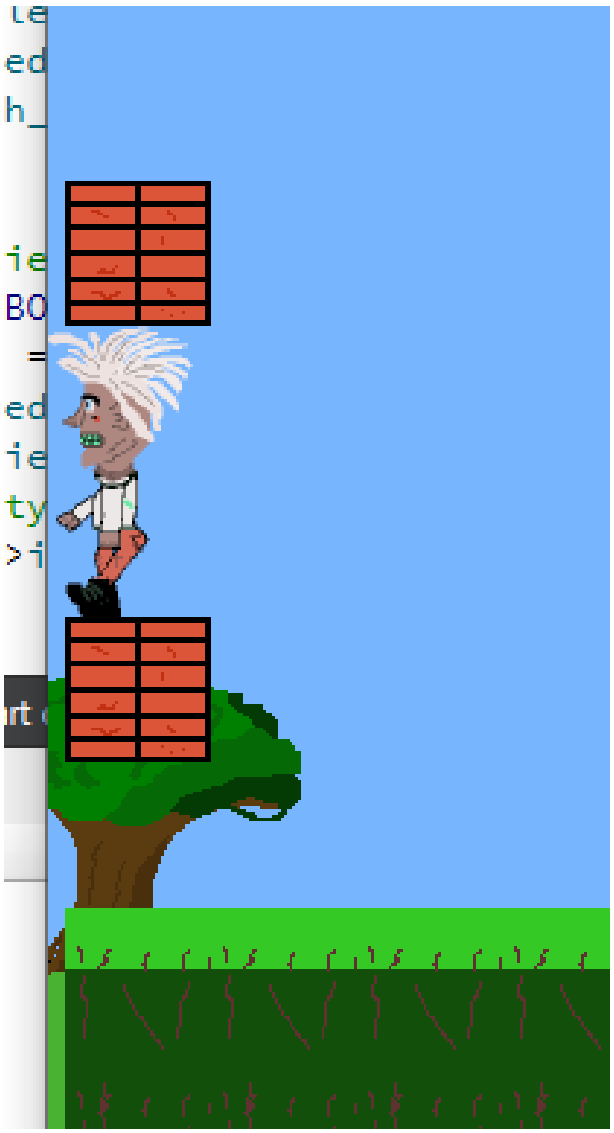
(b) Changed size

FIGURE 18 – Changed size screen

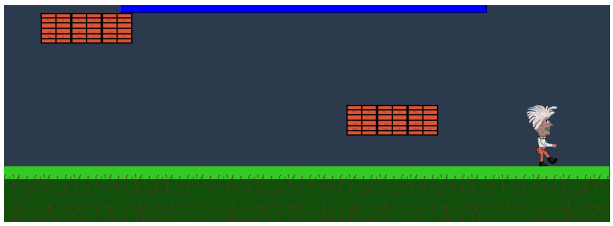
4.3 Data

Génération du monde

Pour tester les données il faut regarder que lorsque la difficulté est choisie les cases correspondent bien aux coordonnées comme la figure Fig : 14 le montre, ceci fonctionne donc bien.



(a) Left side Einstein doesn t pass

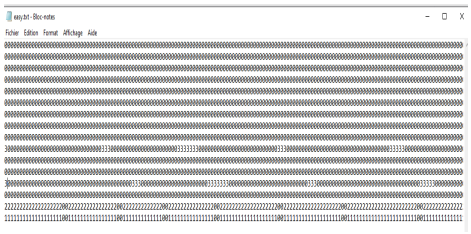


(b) Right side

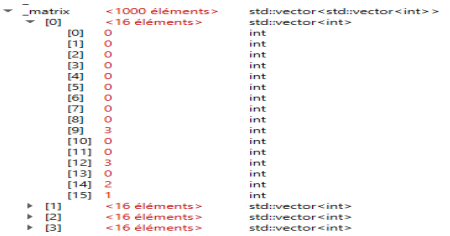
(c) End Window

FIGURE 19 – test sides and end

La première colonne de la matrice est comparée au fichier txt, et il se trouve que les résultats sont les mêmes ce qui vient appuyé la vérification du test (figure Fig : 20).



(a) txt with the number between 0 and 3.



(b) Matrix in Game column 1.

FIGURE 20 – Correct matrix generate

Nombre de bots

Il faut aussi s'assurer qu'il y ait bien le nombre de bots voulue. La figure Fig : 21a montre le nombre de bots voulue pour la difficulté *easy* c'est à dire 9 bots (boucle *for*) +1 = 10 bots, et la figure Fig : 21b montre qu'il y a bien 9 bots.

```
for (int i(1000);i<9001;i+=distBetweenBots){
    QPoint p(i,200);
    MiniBot* b = new MiniBot(p,_element);
    b->setIsOnFloor(true);
    _bots->push_back(b);
}
QPoint p(500,200);
MiniBot* b = new MiniBot(p,_element);
b->setIsOnFloor(true);
_bots->push_back(b);
```

(a) Bots Codes including in Data

(b) effective number of Bots

FIGURE 21 – Bot number correct

5 Conclusion

Tous les tests effectués se sont bien déroulés, il a été important de les exécuter au fur et à mesure du codage pour ne pas affecter une partie qui viendrai après. La grande difficultés a été le codage des collisions qui a demandé beaucoup d'effort mais qui au final fonctionne très bien.

Tous les objectifs n'ont pas été atteints comme le montre la liste suivante :

- V0.0 Le personnage peut marcher en avant et en arrière.
- V0.5 Le personnage peut sauter et courir, des trous sont présents au sol.
- V1.0 Obstacles et différents étages
- V1.5 Degrés de difficultés différents
- V2.0 bots présents
- V2.X améliorations graphiques

  Newton n'a pas pu être codé comme ennemi dynamique mais simplement comme un miniBot plus grand par manque de temps, mais sa classe a tout de même était préparé pour un futur codage plus complexe. Néanmoins la plupart des objectifs ont été accomplies et le jeu est jouable.