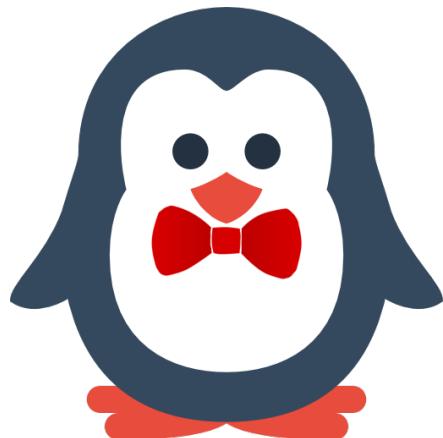


# James Gouin

## et la Banane Sacrée

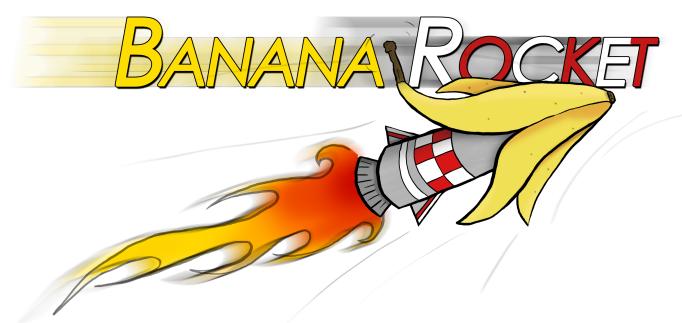


Romain Claret-Yakovenko  
Margaux Divernois  
Steve Visinand

haute école  
neuchâtel berne jura  
**arc** ingénierie  
saint-imier le locle delémont

Professeur : Monsieur Grunenwald  
Classe : INF2 DLM  
27/01/2015

James Gouin et la banane sacrée



présente

# James Gouin et la banane sacrée

## Abstract

James Gouin et la banane sacrée est un jeu de réflexion en 2D. Le joueur doit diriger un pingouin à travers des énigmes dans un univers au thème glacial où la résolution d'une énigme permet de passer à l'énigme suivante. Le but du joueur est de terminer tous les niveaux en résolvant les énigmes les unes après les autres pour finalement récupérer l'objet ultime et résoudre le mystère derrière celui-ci.

Le déroulement du projet est décrit en détail dans ce document, les choix qui ont été faits, les différentes possibilités commentées ainsi que les explications techniques des éléments clef du projet.

## Table des matières

<b>Abstract .....</b>	<b>3</b>
<b>Introduction.....</b>	<b>1</b>
La longue histoire de James Gouin.....	1
<b>Conception .....</b>	<b>2</b>
Planning et suivi .....	2
<b>Développement .....</b>	<b>3</b>
Création des Maps (Scènes) .....	3
Chargement automatisé des niveaux.....	3
<i>Obtention des informations .....</i>	<i>3</i>
<i>Population de la scène .....</i>	<i>4</i>
<i>Population des ennemis .....</i>	<i>5</i>
<i>Obtention des dialogues .....</i>	<i>6</i>
Surfaces .....	7
Gestion des scènes et des vues .....	8
<i>Positionnement de la vue.....</i>	<i>8</i>
<i>Changement de vue / de niveau .....</i>	<i>9</i>
<i>Cases de transitions .....</i>	<i>9</i>
<i>Vérification de la position du pingouin et des blocs.....</i>	<i>10</i>
Gestion du Clavier .....	10
Le personnage .....	11
Collisions .....	11
<i>Joueur en croix .....</i>	<i>12</i>
Evénements.....	14
<i>Gestion des événements sur des cases : .....</i>	<i>14</i>
<i>Objets.....</i>	<i>14</i>
Glisse .....	15
Bloc déplaçables.....	17
Ennemis.....	21
<i>Champ de vision .....</i>	<i>21</i>
<i>Déplacement.....</i>	<i>22</i>
<i>Positionnement .....</i>	<i>24</i>
<i>Déplacement .....</i>	<i>25</i>
<i>Détection du pingouin, activation du champ de vue.....</i>	<i>28</i>
Positionner les Proxys .....	28
Menu Principal .....	28
Profil et Système de sauvegarde .....	29
<i>Classe profil.....</i>	<i>29</i>
<i>Classe MenuStart .....</i>	<i>29</i>
Menu Pause.....	30
Recommencer l'énigme, recommencer le niveau et checkpoints .....	32
<i>Checkpoints.....</i>	<i>32</i>
<i>Recommencer le niveau et le jeu .....</i>	<i>32</i>
<b>Conclusion .....</b>	<b>34</b>
<b>Bibliographie .....</b>	<b>35</b>
<b>Annexes.....</b>	<b>35</b>

## Introduction

James Gouin et la Banane Sacrée est jeu d'infiltration en 2D sans combats sous forme de puzzle game. Le joueur manipule un pingouin agent-secret envoyé sur un iceberg pour récupérer la « banane sacrée » volée par un singe. Son personnage peut marcher et glisser dans les quatre directions. Il fera face à un niveau de tutoriel, 6 niveaux normaux et un niveau final.

Il a pour objectif de récupérer des blocs de glace auprès du boss normal de chaque niveau afin de se créer un passage jusqu'à l'iceberg central abritant l'igloo du singe. Pour cela, le joueur devra résoudre des problèmes logiques en se frayant un chemin tout en évitant d'entrer dans le champ de vision des ennemis.

## La longue histoire de James Gouin

James Gouin est un Jeune Pingouin avec beaucoup de potentiel. Il fut recruté à la fin de ses études de poissonnerie par une organisation secrète capitaliste. Ces compétences, hors du commun, en esquive de queues de poissons enragés le propulsèrent au titre d'espion pour les projets spéciaux. Il est cependant un grand crsaintif de l'eau, il est impossible pour lui de mouiller son noeud papillon qui lui fût donné par son maître Poissonnier.

Un jour, alors que notre héro pêchait, il reçut par une fusée glaçon un nouvel ordre de mission... Le message glacé contenait ces quelques mots :

### Ordre de mission pour James Gouin

Docteur Douceur, le méchant Singe, a volé la banane Sacrée ! Il est de ton devoir de la récupérer. Pour cela, nous t'envoyons en mission sur le très dangereux ... Iceberg glacé, la base du monstrueux singe Docteur Douceur! Tu as deux jours avant qu'il n'utilise la banane sacrée pour faire un délicieux banana split. Empêche-le ! Dans 1min, ce message glacé s'auto-fondra. Nous nierons tout lien avec cette mission. Vous êtes maintenant seul. Bonne chance.

Le pingouin décide d'accepter la mission et part aussitôt pour rejoindre le QG de l'organisation. Il lui est mis à disposition une Banana Rocket pour rejoindre l'antre du Docteur Douceur. Une fois le voyage terminé, il atterrit à quelques mètres de l'entrée de la forteresse de l'iceberg glacé. Alors que le pingouin tente d'entrer sur l'iceberg, il se voit bloquer par une grosse loutre et un portail. Celle-ci lui explique qu'elle est la gardienne du portail et est chargée de la garder fermée à tout envahisseur. Cependant, elle trouve qu'elle n'est pas assez bien payée pour faire ce travail. La loutre propose de fermer les yeux et laisser le pingouin passer si celui-ci lui donne du poisson. En effet, vu son poids, elle ne peut plus aller pêcher.

C'est ainsi que le joueur commence son aventure à la quête de la banane sacrée.

## Conception

### Planning et suivi

La forge Bitbucket avec la technologie GIT a été utilisée. Les commits sont datés et considérés comme le journal de bord de l'équipe. Ils sont commentés et contiennent les modifications et ajouts.

La méthodologie de développement en cascade s'est imposée au début du projet, cependant il s'est avéré que celle utilisée était principalement la méthodologie agile.

La structure du projet était en effet composée d'un « Product Backlog » avec toutes les fonctionnalités du cahier des charges. D'un « Sprint Backlog » qu'on se donnait tous les lundi pendant les heures du projet. Les « sprints » en eux même qui duraient une semaine, du lundi au lundi suivant. Et chaque début de semaine on avait une itération du jeu avec de nouvelles fonctionnalités.

La répartition du travail a aussi été adaptée à chaque sprint, il est rare qu'une personne ait été la seule à travailler sur une classe. Au final le travail selon les classes a été reparti ainsi :

Nom	Classes / Autres
Margaux	Gameboard, S_ViewTransition, Profil, M_MenuStart, Object, Level, W_Object, S_Dialog, W_Dialog, W_Life, ItemPopulation, Enigmes, Design, Système de sauvegarde
Romain	Player, Surface, Gameboard, S_ViewTransition, P_Penguin, M_MenuStart, M_Pause, Level, MainGame, Images, ItemsPopulation, Icons, Licenses, Maps, Enigmes, Deploy, Installer
Steve	Surface, Gameboard, Système de déplacements (Timers), B_Movable, B_Wall, S_Snow, P_Penguin, B_Water, Ennemi, S_Ice, E_Renard, E_Loup, Enigmes, Design

Le travail a été reparti équitablement. Tout le monde a pu toucher à tout, et l'expérience acquise lors de ce projet a été profitable à toute l'équipe, ce qui est le plus important. Pour voir la répartition des tâches selon les fonctionnalités et non selon les classes, se référer au diagramme de Gantt joint en annexe.

## Développement

### Création des Maps (Scènes)

Le choix de logiciel pour la création des cartes s'est portée sur l'outil open source « Tiled », qui permet de découper un Tile (qui est une image contenant des morceaux d'images) en partie 32x32 pixels. Cela facilite la tâche lors de la création de niveaux. Tiled a été le logiciel plébiscité car il permet de générer des fichiers images PNG ainsi que des fichiers texte contenant la composition précise de tous les éléments. Ce fichier texte est ensuite utilisé pour effectuer la population du jeu.

### Chargement automatisé des niveaux

#### Réflexion

Dès le départ, l'objectif était d'avoir un chargement automatisé des niveaux. Ainsi, en mentionnant le numéro du niveau, on doit pouvoir charger complètement une scène contenant le niveau et ses éléments.

Les informations suivantes devront être contenues dans le fichier :

- Taille du niveau (Nombre de cases en format 20\*15)
- Point de départ du pingouin
- Vue de départ
- Case débloquant l'ouverture de la sortie
- Les messages des dialogs

#### Implémentation

C'est la classe « Level » qui s'occupe complètement de l'automatisation des niveaux. Ci-dessous se trouve les détails d'implémentation de la classe.

#### Obtention des informations

Certains éléments sont indispensables avant la population. C'est pourquoi la fonction `void getSceneSize()` lit le fichier `<numeroDuNiveau>header.txt`. Par exemple pour le niveau tutoriel, il contient les éléments suivants:

```
width=60 //Largeur en pixel (20pixels par vue)
height=30 //Hauteur en pixel (15pixels par vue)
viewRequestedX=1 //Première position de la vue x
viewRequestedY=2 //Première position de la vue y
levelStartX=7 //Position de début x
levelStartY=21 //Position de début y
unlockEndX=4 //Case qui débloque la fin du niveau x
unlockEndY=3 //Case qui débloque la fin du niveau y
```

### Population de la scène

La fonction `QGraphicsScene*` `populateScene()` crée une scène et lui ajoute tous les éléments qui constituent le niveau. Pour cela, elle doit lire le fichier nommé `<numeroDuNiveau>.txt`. Tout d'abord, elle définit le fond d'écran de la scène avec l'image nommée `<numeroDuNiveau>.png`.

On utilise des tableaux à deux dimensions qui vont présenter le contenu de chaque matrice. Dans le fichier texte, chaque matrice est présentée sous la forme ci-dessous (exemple 5x5) :

```
[layer]
type=<nomDuType>
data=
0,0,0,0,0
0,1,0,1,0
0,0,0,0,0
0,0,0,1,0
0,0,0,0,0
```

Il est donc nécessaire d'utiliser deux boucles, une pour la lecture des lignes et une pour la lecture des colonnes.

```
if(line[line_count].contains("type=<nomDuType>"))
{
    line_count++;
    line[line_count]=t.readLine();
    line_count++;
    line[line_count]=t.readLine();

    for (matY = 0; matY < maxBlocksHeight; matY++)
    {
        QStringList values = line[line_count].split(",");
        for (matX = 0; matX < maxBlocksWidth; matX++)
        {
            <nomDeLaMatrice>[matX][matY] +=
                values.at(matX).toInt();
        }
        line_count++;
        [line_count]=t.readLine();
    }
}
```

Cette implémentation demande beaucoup d'itérations. Cela signifie qu'à moins d'avoir beaucoup de cases à remplir, il est préférable de ne pas utiliser de matrices (exemple : case de début du niveau).

Ensuite, on effectue une dernière fois deux boucles qui font le parcours des matrices. On teste chaque bloc. Si `<nomDeLaMatrice>[i][j]` c'est pas = 0 (appelé plus loin valeur), alors on crée un élément qu'on ajoute à la scène.

Exemple :

```
if (<nomDeLaMatrice>[i][j] != 0)
{
    <classeDeL'Element> *item = new <classeDeL'Element>();
    item->setPos(i,j);
    scene->addItem(item);
}
```

Pour certains cas particuliers, d'autres activités sont faites si la valeur n'est pas égal à 0.

S'il s'agit de:

- Une case de dialogue, alors la valeur est ajoutée à l'attribut `int dialogNumber` de l'élément.
- Une case de transition, alors la valeur est testée. Elle est définie comme suit : 1<sup>er</sup> chiffre = Objet, 2<sup>ème</sup> chiffre = Quantité. Pour une valeur située entre 20 et 30, la définition se fait comme ci-dessous :

```
item->setLevelEnd(false);
item->setNbItem(Mat_Doors[i][j]%20);
item->setNeededItem("Poisson");
```

- Une case de fin de niveau, alors l'attribut `bool levelEnd` de la case de transition est modifié à `true`.
- Un ennemi, qui est présenté plus en détail au point suivant.

## Population des ennemis

Tout d'abord, les ennemis possèdent une matrice, définie durant les parcours précédents. Cependant, un ennemi nécessite un parcours sous la forme d'une `QList` de `QPoint` définissant son parcours. Ces informations sont définies dans le fichier sous la forme suivante :

**Exemple du niveau tutoriel :**

8-1,8-7//8-9,5-9,5-12,8-12,8-9,11-9,11-12,8-12//15-6

**Définition :**

// Sépare deux ennemis  
, Sépare deux points  
- Sépare les coordonnées x et y du point

L'ordre des ennemis est très important. Ils devront être définis de haut en bas et de gauche à droite. Lors de la lecture du fichier, une `QList<QList<QPoint>>` `ennemi` est créée. Elle contient les `QList` qui seront utilisées lors de la définition des ennemis.

## James Gouin et la banane sacrée

```
QStringList listEnnemi = line[line_count].split("//");

for(int j = 0; j < listEnnemi.size(); j++)
{
    QStringList listPoint = listEnnemi.at(j).split(",");
    QList<QPoint> listeDePoints;
    for(int i = 0; i < listPoint.size(); i++)
    {
        QStringList point = listPoint.at(i).split("-");
        listeDePoints.append(QPoint(point.at(0).toInt(),
                                    point.at(1).toInt())));
    }
    ennemi.append(listeDePoints);
}
```

Dans la deuxième phase, lors de la création des éléments, on vérifie à l'aide d'un Switch de quel type est l'ennemi afin de déclarer le bon objet. Cette décision a été prise afin de limiter le nombre de matrices à parcourir lors de la phase précédente. Ensuite, on lui attribue la QList à l'emplacement « k » de la liste de listes nommée « ennemi ».

```
int k = 0;
if (Mat_Enemies[i][j] != 0)
{
    switch(Mat_Enemies[i][j])
    {
        case 1: {
            E_Renard *item2 = new E_Renard(ennemi.at(k), game);
            item2->addToScene(scene);
            break;
        }
        case 2: {
            E_Loup *item2 = new E_Loup(ennemi.at(k), game);
            item2->addToScene(scene);
            break;
        }
        default: break;
    }
    k++;
}
```

## Obtention des dialogues

Il s'agit de la fonction `void getSceneDialog()`. Les dialogues sont obtenus depuis un fichier nommé <numeroDuNiveau>texte.txt. Il contient les textes des dialogues (un texte par ligne).

A la lecture du fichier, les lignes sont insérées dans une liste qui contient tous les dialogues. Ils seront ensuite appelés par les cases de dialogues.

## Surfaces

### Réflexion

La solution qui a été retenue afin de répondre aux besoins du jeu est la création d'une surface de jeu composée d'une multitude de petits blocs qui représentent des surfaces différentes sur lesquelles le pingouin va réagir différemment suivant leur type.

Par décision commune et grâce à la quantité de ressource à disposition sur internet, le jeu est défini en carrés d'une taille de 32px\*32px.

Deux choix possibles pour créer la surface de jeu avec des blocs :

- Chaque bloc dispose d'une image : Cela nécessite la mise en place d'un algorithme pour que les blocs dans les angles aient une apparence différente. Cette méthode est plus flexible pour la modification des niveaux et de leur design.
- Les blocs de surfaces sont invisibles et une image de fond est placée en « background » donnant la représentation visuelle des murs et surfaces : Cette méthode nécessite de créer les images de « background »

Le deuxième choix a été retenu, car, comme présenté précédemment, le logiciel Tiled répondait aux besoins de créations d'un fond. L'implémentation d'un algorithme pour la première solution semblait donc une perte de temps au vu de l'efficacité du programme trouvé. De ce fait, tous les blocs de surfaces sont invisibles par défaut.

**Types nécessaires au jeu :**

- S\_Ice : Glace sur laquelle le joueur glisse (S'il n'a pas de bottes)
- S\_Snow : Neige sur laquelle le joueur peu marcher
- S\_ViewTransition : Case gérant le déplacement entre les vues/niveaux
- S\_Dialog : Affiche un message au passage du joueur
- B\_Water : Eau, si le joueur tombe dedans il perd une vie
- B\_Wall : Mur infranchissable par le joueur
- B\_Movable : Bloc que le joueur peu pousser. Il peut glisser ou couler.

### Implémentation

La classe principale créée pour de l'implémentation des surfaces est la « Surface ». Elle hérite de [QGraphicsRectItem](#).

Plusieurs méthodes de [QGraphicsRectItem](#) ont été réimplémentées pour fonctionner avec le système choisi de positionnement des blocs par case de 32px. A noter que l'obtention de la tailles des blocs est obtenue par le biais de la méthode statique « `getGameSquares()` » de la classe « `GameSquare` » ainsi il est facile de modifier la dimension des blocs dans le jeux.

### Les méthodes implémentées :

- void setPos(int x, int y) : Positionner la surface
- QPoint getPos() : Obtenir la position de la surface
- void setColor(QString) : Assigne la couleur à la surface (Facultatif)

Les classes « S\_Ice », « S\_Snow », « B\_Water » et « B\_Wall » héritent simplement de « Surface » sans ajouter de méthodes spécifiques. Elles sont créées pour différencier le type de surface lorsque le joueur marche dessus.

Les classes « S\_Dialog » et « S\_ViewTransition » implémentent d'autres fonctions. Pour « S\_Dialog » et « S\_ViewTransition » se référer plus loin dans le rapport. Quant à « B\_Movable », une section lui est également dédiée dans le rapport.

## Gestion des scènes et des vues

### Positionnement de la vue

#### Réflexion

Cette fonctionnalité est indispensable pour le fonctionnement du jeu. Dès le départ, le concept était que le joueur se déplace sur une scène (représentant un niveau). Un niveau étant composé de plusieurs énigmes, il doit donc pouvoir se déplacer entre plusieurs vues, chacune représentant une énigme.

Le jeu comporte de nombreux systèmes de coordonnées :

- Les coordonnées réelles dans la scène
- Les coordonnées par rapport aux cases (Rapport de 32pixels)
- Les coordonnées de la position de la vue sur la scène

(1,1)	(1,2)	(1,3)
(1,2)	(2,2)	(3,2)

Il est important de pouvoir déplacer non seulement le joueur, mais également tous les widgets s'affichant au-dessus de la scène, c'est pourquoi un troisième set de coordonnée est ajouté : Les coordonnées de la position de la vue.

Pour plus de lisibilité, elle est représentée par deux coordonnées représentant sa position x et y par rapport à la scène (cases de vue de 20\*15 blocs). Ci-contre un exemple avec le menu tutoriel.

#### Implémentation

La fonction « setviewPosition() » de la class Gameboard gère la position de la scène sur la vue. Pour son implémentation, un QPoint (nommé « viewRequested ») contenant la position souhaitée a été utilisé. Il convertit la valeur de ce QPoint en positions réelles sur la scène en modifiant les valeurs de deux variables : viewPositionX et viewPositionY.

## Changement de vue / de niveau

### Réflexion

Afin de séparer correctement le jeu en énigmes, il est nécessaire d'avoir des vues fixes et délimitées par une entrée et une sortie. Tant que le joueur reste sur ces cases, la vue ne doit pas bouger. Cependant, la vue doit être modifiée s'il arrive sur les cases de début et de fin.

La solution trouvée à ce point est l'utilisation de la collision entre le pingouin et une case de transition. On sait que si par exemple le pingouin arrive sur la case de transition en avançant dans la direction gauche, la vue peut effectuer une translation sur la gauche.

Le point fort de cette solution est son adaptabilité rapide avec tout format de niveau. Peu importe la direction suivante, la vue sera déplacée correctement. Le point faible est qu'il est nécessaire de bloquer l'arrivée sur une case de transition par d'autres directions. Elle doit donc être soit entourée d'autre cases de transitions, soit de murs/eau.

### Implémentation

La fonction « checkChangeView(char sens) » de la classe Gameboard est appelée à tous les déplacements du pingouin. Si le pingouin est sur une case de transition, elle vérifie les propriétés de cette case et effectue les actions adéquates parmi : Changement de niveau, Déplacement de la vue, Blocage de l'accès.

Lorsque l'utilisateur change de vue, la fonction appelle la fonction « changeView(char sens) ». Celui-ci gère le déplacement de la vue d'une case grâce à la fonction « setViewPosition() » présentée plus haut. Elle gère également le repositionnement des différents widgets superposant la scène grâce à des QGraphicsProxyWidget ainsi que la sacoche temporaire du pingouin.

## Cases de transitions

### Réflexion

La question suivante s'est posée : Est-ce que le pingouin peut de toute façon passer et changer de scène ? Non, car il est important qu'il résolve l'énigme. Par moments, il est nécessaire qu'il obtienne un objet avant de continuer (comme c'est le cas dans le niveau tutoriel). De plus, les objets en question sont différents en type et quantité. Pour cela, il faut effectuer des vérifications supplémentaires avant le déplacement du pingouin dans l'énigme suivante.

### Implémentation

Les cases de transitions sont des surfaces. Elles en héritent donc toutes les propriétés. Pour répondre aux points présentés précédemment, il faut lui ajouter **les 5 attributs** :

**bool levelEnd :** Prends la valeur True s'il s'agit d'une case "Fin de niveau"  
**int nextLevel :** Numéro du niveau suivant (s'il s'agit d'une case « Fin »)

**bool needItem :** Prends la valeur True si la case doit vérifier les objets récupérés

**QString\* neededItem :** Contient le nom de l'objet à récupérer  
**int nblItem :** Contient la quantité qu'il doit récupérer

### Vérification de la position du pingouin et des blocs

#### Réflexion

Le pingouin ne doit sortir de la vue qu'à des moments précis (l'utilisation des cases de transitions). Pour éviter tout débordement et sorties de vue du pingouin, il est nécessaire de contrôler sa position et l'empêcher de bouger s'il sort de la scène.

De même, les blocs déplaçables ne devraient jamais sortir de la vue. Il faut donc vérifier que chaque bloc déplaçable reste bloqué s'il arrive au bord de la vue.

#### Implémentation

La fonction « checkPosition() » de la classe Gameboard reçoit un QGraphicsItem en paramètre et vérifie que celui-ci est dans la vue (retourne true) ou non (false).

Dans la pratique, l'objet envoyé à cette fonction n'est pas l'objet lui-même (pingouin ou bloc) mais le bloc de collision de la direction souhaitée. Ainsi, la vérification est faite avant déplacement.

### Gestion du Clavier

#### Réflexion

Le joueur a besoin de pouvoir diriger son personnage à l'aide de touches directionnelles sur son clavier. Les touches choisies sont W A S D et les flèches Haut Gauche Bas Droite : ces touches du clavier sont le plus souvent utilisées pour les jeux vidéo.

Afin de couvrir les autres fonctionnalités du jeu, des touches supplémentaires ont étées implémentées. La touche ESC permet de mettre le jeu en pause, la touche ESPACE permet de valider la lecture des dialogues dans le jeu et la touche 0 permet de recommencer rapidement l'éénigme. Le clavier est également utilisé lorsque le joueur crée un profil, en inscrivant son nom de profil.

#### Implémentation

Le contrôle du clavier entre les différents widgets (le menu principal et le gameboard) est donné à l'aide de la fonction grabKeyboard() et est retiré à l'aide de releaseKeyboard(), toutes deux appartenant à la classe QWidget. C'est ainsi que les évènements du clavier ne sont pas interceptés par un widget différent que celui avec lequel le joueur interagit.

Dans le menu principal (classe MenuStart), on donne la lecture du clavier au QLineEdit lorsque l'utilisateur souhaite créer un nouveau profil. La lecture du clavier est enlevée lorsque le joueur valide son nouveau profil et lance ainsi une partie.

## James Gouin et la banane sacrée

```
username = new QLineEdit(this);
username->grabKeyboard();

...
username->releaseKeyboard();
```

Lors d'une partie, on donne la lecture du clavier lors de la construction du Gameboard. On utilise la méthode appartenant à QWidget : keyPressEvent(QKeyEvent), pour réagir aux touches utilisées par le joueur.

À chaque évènement de clavier, un contrôle de l'état du jeu est effectué pour éviter des interactions parasites (par exemple le personnage qui continue à glisser alors que le menu pause est affiché).

```
if(!toggleMenuPause && !isSliding)
{
    if(!dialogToogle)
```

L'utilisation des toggles s'est avérée plus efficace car beaucoup d'interactions sont internes au Gameboard et demandaient une vérification partagée entre plusieurs classes.

## Le personnage

### Réflexion

Le personnage a été créé avec l'obligation de pouvoir interagir avec d'autres QGraphicsItems. Lors des prototypes, la solution pour le déplacement et la détection des collisions du personnage avec une disposition en croix a été sélectionnée (présentée dans la section Penguin). Le personnage (Player) est donc celui que le joueur contrôle, et un groupe de QGraphicsItems autour de lui informe le Gameboard de ce qu'il se passe. L'expérience acquise a permis de créer la classe Ennemi qui intègre le QGraphicsItem central (ici le personnage de l'ennemi).

### Implémentation

Le personnage est défini dans une classe « Player » héritant de QGraphicsItem. La classe contient les informations graphiques, l'orientation et le positionnement sur le Gameboard du joueur.

## Collisions

### Réflexion

Plusieurs solutions permettaient de gérer les collisions de James Gouin afin que ce dernier puisse connaître la joie de se payer un mur ou de tomber dans l'eau glacée, mais également les plaisirs de la glisse et de pousser des blocs de glaces.

Lors des réflexions permettant la détection des collisions entre deux éléments dans le GVF, la fonction « collidingItems » s'est imposée. Cette dernière renvoie une QList contenant les objets en collision.

## James Gouin et la banane sacrée

Pour détecter la surface sur laquelle se trouve le pingouin, il suffit de tester les types des surfaces en collision. La fonction collidingItems ne renvoie pas que les blocs en dessous du pingouin mais également ceux adjacents. Pour régler le problème il a été décidé de réduire la taille du pingouin de 1px de chaque côté. Cela a semblé la solution la plus évidente. La largeur du pingouin est donc de 30.

Pour ce qui est de pouvoir déplacer un bloc, ou de ne pas pouvoir aller sur certains blocs interdits (mur) il en a été tout autre. Le problème est qu'il faut réussir à détecter la collision entre le joueur et le bloc en question AVANT que le joueur passe dessus.

### Plusieurs solutions :

1. Garder une largeur de 32 pour le pingouin, détecter les collisions sur les bords aussi et travailler avec la position des blocs en collision par rapport à la position du pingouin pour définir sur quel flanc est le bloc.
2. Ne pas prévoir les collisions, permettre le déplacement sur les « murs » et quand le pingouin est en collision avec ce type, faire qu'il revienne rapidement en arrière d'une case.
3. Implémenter un joueur en croix : Des blocs invisibles sont placés de chaque côté du joueur, ceux-ci servent à détecter la collision avec un mur et à bloquer cette direction pour le joueur tant que la collision est vraie.

Après réflexion, la troisième solution a été décidée. La première semblait compliquée et la deuxième ne donnait pas un bon résultat ; sur les machines les moins puissantes, on peut voir le pingouin légèrement clignoter lorsque l'on essaye de passer sur un mur.

### *Implémentation*

#### Joueur en croix

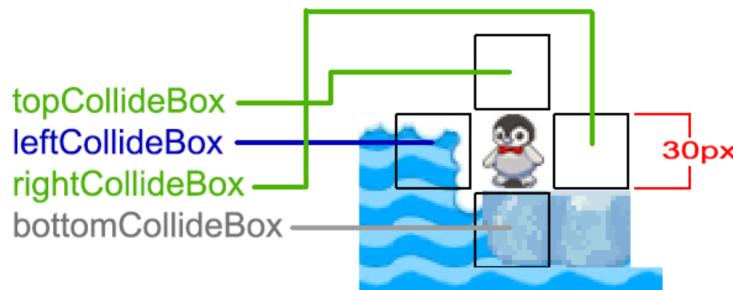
Pour obtenir un joueur en croix, une classe qui hérite de la classe précédemment implémentée : « Player » a été développée. Dans cette nouvelle classe « P\_Penguin » ont été ajoutés 4 QGraphicsRectItem qui se placent autour du « Player » et qui le collent littéralement au moindre de ses déplacements.

Dans un premier temps, il était prévu d'ajouter les 4 QGraphicsRectItem directement dans la classe « Player » qui hérite de « QGraphicsItem », Mais plusieurs bugs graphiques sont rapidement apparus, on a conclu que l'intégration de « QGraphicsRectItems» dans une classe héritant de « QGraphicsItem » était délicat à implémenter, d'où le choix de créer une autre classe héritant de « Player ».

Donc, dans la classe « P\_Penguin », les méthodes « setPos » et « moveBy » sont réimplémentées. Ainsi, en plus de déplacer le joueur, elles déplacent également les 4 blocs de collisions. Pour les mêmes raisons que celles énoncées précédemment les blocs et le Player sont des carrés de 30px de largeur.

La méthode `void addToScene(QGraphicsScene*)` permet de facilement ajouter un `P_Penguin` à la scène. Elle s'occupe d'ajouter le Player ainsi que les 4 `QGraphicsRectItem` directement en un appel.

**Positionnement :**



Sur cet exemple les blocs « `topCollideBox` » et « `rightCollideBox` » détecteront la collision avec une surface de type « `snow` » et « `leftCollideBox` » détectera une collision avec une surface de type « `water` », le joueur pourra se diriger dans ces directions. Mais le bloc « `bottomCollideBox` » détectera une collision avec un type « `Wall` » et le joueur ne pourra pas se déplacer dans cette direction.

**Détection des collisions « simples » (refus de passer)**

La détection des collisions se fait dans la classe « `Gameboard` » afin de pouvoir implémenter les blocs déplaçables qui glissent par la suite.

5 méthodes permettent de récupérer les listes des blocs en collision :

```
QList<QGraphicsItem *> CollidesRight();
QList<QGraphicsItem *> CollidesLeft();
QList<QGraphicsItem *> CollidesTop();
QList<QGraphicsItem *> CollidesBottom();
QList<QGraphicsItem *> CollidesCenter();
```

Le test de collision est effectué à chaque fois que l'utilisateur demande de déplacer le pingouin.

4 méthodes ont étées créées dans `Gameboard` pour détecter les collisions du pingouin. En réalité les 4 redirigent vers une même méthode en spécifiant le sens par une variable de type « `char` » et en fournissant la bonne liste de collision à tester. Ces 4 méthodes ont étées développées afin de simplifier l'utilisation dans le code. Ce sont « `MovePingouintoTop`, ..`ToBottom`, ..`ToLeft`, ..`ToRight` et elles se basent sur « `MovePingouin` ».

« `MovePingouin` » parcourt donc l'ensemble des blocs en collisions, retourne « `vrai` » si le pingouin peut se diriger dans cette direction et « `false` » dans le cas contraire. Elle vérifie également en dernier que le pingouin ne sort pas de la vue.

**Cas particulier :** La gestion des blocs « déplaçables ». La méthode va d'abord voir si le bloc en collision est « poussable » dans la direction voulue par le joueur (avec l'appel de 4 méthodes implémentées dans « `B_Movable` » (`IsMovableTo...`)). Si c'est le cas, on autorise le déplacement du pingouin et on enregistre l'adresse du bloc que l'on doit déplacer dans un

## James Gouin et la banane sacrée

attribut du Gameboard « moveBloc ». Le déplacement du bloc déplaçable ne se fait pas dans « MovePingouin » car il faut gérer le cas du bloc qui glisse sur la glace et du pingouin qui ne glisse pas avec. Plus d'informations dans la section dédiée aux blocs déplaçable.

## Événements

### Gestion des événements sur des cases :

#### Réflexion

Lorsque le joueur déplace le pingouin sur la carte, il active différents événements. Les événements possibles sont les suivants :

- Ramasser un objet
- Ramasser une vie (par définition, une vie est un objet)
- Activer un dialogue
- Tomber dans l'eau
- Entrer en collision avec un ennemi
- Débloquer la sortie

#### Implémentation

C'est la fonction « checkPositionEvents() » de la classe Gameboard qui gère cela en fonction de la case sur laquelle se trouve le Pingouin. Elle est appelée à chaque déplacement du pingouin et effectue les vérifications nécessaires.

## Objets

Les objets sont les items positionnés dans le niveau que le joueur peut ramasser. Ils possèdent deux attributs :

```
QString* nom;  
QBrush* objectSkin;
```

Le nom permet de définir la texture car chaque image d'objet est appelée exactement la même chose. Ainsi, on définit uniquement son nom à son instanciation.

### A ce jour, il y a 3 types d'objets :

- Chaussures : Permettent de ne pas glisser sur la glace. Elles modifient l'attribut **slideAble** de l'objet pingouin en false. Cet objet est supprimé automatiquement au changement d'énigme.
- Œuf : Il s'agit des vies du pingouin. Chaque œuf ramassé ajoute une vie.
- Poisson : Il s'agit des éléments indispensables durant le niveau tutoriel.

## Glisse

### Réflexion

Si le pingouin est sur de la glace, il doit glisser jusqu'à rencontrer un obstacle qu'il ne peut pas déplacer. Il doit également glisser après un déplacement depuis de la neige sur la glace. Cependant, lors du cas exceptionnel où le pingouin est équipé de bottes à crampons, il ne doit pas glisser.

### Lorsque l'utilisateur active un déplacement :

1. On vérifie que le pingouin arrive à se déplacer dans cette direction (Partie « Détection des collisions « simples » (refus de passer) » du rapport).
2. OK : On déplace le pingouin.
3. Si le pingouin est sur une surface glissante : Il continue à se déplacer dans la même direction (on recommence au point 1). Le joueur ne peut plus rien faire jusqu'à ce que le pingouin ait fini de se déplacer.

La méthode « bool isSlide() » est utilisée afin de détecter si le pingouin est sur de la glace. L'attribut « char cSensPingouinSlide » de la classe Gameboard est utilisé pour mémoriser le sens de glisse du pingouin. Enfin, l'attribut «bool isSliding » est utilisé pour bloquer les activités du joueur jusqu'à la fin du déplacement.

Dans un premier temps la méthode pour implémenter le point « 3 » était une boucle « while() ». Le problème avec cette technique est que l'on ne voit pas le pingouin se déplacer à l'écran, placer des « sleep » dans la boucle n'est pas une méthode viable, théoriquement cela bloquerait le programme à chaque occurrence mais visuellement on constate que le programme bloque jusqu'à ce que la boucle termine sans mettre à jour la position du pingouin progressivement. La solution retenue a été d'ajouter un timer dans la classe Gameboard qui appelle un slot qui se charge de déplacer le pingouin toutes les x secondes.

### Implémentation

Le slot utilisé est « SlidePingouin()» et le timer connecté : « **timerPingouinSlide** ». Le démarrage du timer se produit après le déplacement du pingouin seulement si ce dernier glisse («isSlide() = true »).

S'il glisse, on modifie l'attribut « isSliding » à vrai pour bloquer les touches du clavier, on enregistre le sens de déplacement du pingouin dans « cSensPingouinSlide » et on démarre le timer.

Voici le comportement du slot qui se charge de la glissade à proprement parler :

```
void Gameboard::SlidePingouin()
{
    bool endSlide = true;

    switch (cSensPingouinSlide)
    {
        case 't':
            if(MovePingouinToTop() && pingouin->isSlide())
```

```

{
    pingouin->moveBy(0, -1);           // Déplacement du pingouin
    checkChangeView(cSensPingouinSlide); // Changement éventuel de vue
    checkPositionEvents();             // Vérification des événements

    if(moveBloc != NULL) //Déplacement du bloc que le Pingouin pousse
    {
        moveBloc->moveBy(0,-1);
        SinkMovable(moveBloc);
        moveBloc = NULL;
    }
    endSlide = false;
}
break;

... // Les autres directions
}

if(endSlide)
{
    checkItem();
    checkChangeView(cSensPingouinSlide);
    timerPingouinSlide->stop();          // Arrêt du timer
    isSliding=false;                      // Déblocage des touches
    moveBloc=NULL;
}
}

```

**EndSlide** est une variable utilisée pour voir quand le pingouin a terminé sa glisse. L'utilisation de cette variable évite l'ajout d'un « else » supplémentaire pour chaque direction traitée.

À chaque fois que le timer se termine, on déplace le pingouin dans la bonne direction jusqu'à ce que la variable « **endSlide** » arrive à « **false** » (quand le pingouin n'est plus sur une surface glissante ou s'il ne peut plus avancer). À ce moment-là, on arrête le timer et on débloque les touches du clavier.

A chaque déplacement (comme dans la partie où le pingouin ne glisse pas), on effectue plusieurs vérifications comme « `checkPositionEvents()` ». Cette méthode est décrite en détail plus loin dans le rapport.

Il est important de définir `moveBloc` comme « `NULL` » à la fin de la glissade du pingouin. Sans cela, on constate un bug : Lorsque le pingouin glisse en poussant un bloc et que les deux arrivent sur de la neige, au prochain déplacement du pingouin dans une autre direction, le bloc se déplacera dans la même direction ! Le fonctionnement des blocs déplaçables est décrit dans la partie « `blocs` » du rapport.

La répétition de l'appel de ces méthodes est obligatoire. Certes on aurait pu démarrer directement le timer à chaque déplacement et donc éviter de se répéter dans le code. Mais d'un point de vue réactivité, cela ne semble pas la meilleures des idées. On a donc fait le choix de répéter un peu de code dans le slot appelé par le timer.

## Bloc déplaçables

### Réflexion

Avec les blocs déplaçables est venu les mêmes réflexions que celles effectuées dans la problématique des collisions du pingouin. C'est à dire celle d'une implémentation de bloc de détection des collisions en croix.

Fonctionnement des blocs déplaçables :

- Lorsque le pingouin pousse le bloc, ce dernier doit se déplacer dans la même direction que le pingouin.
- Un bloc déplaçable ne peut pas être tiré par le pingouin.
- Si un bloc déplaçable est poussé contre un mur il ne bouge pas.
- Si un bloc déplaçable est poussé contre un autre bloc déplaçable, ils ne bougent pas.
- Si un bloc déplaçable est poussé dans l'eau, ce dernier coule et se transforme en neige pour que le pingouin puisse marcher dessus.
- Si un bloc déplaçable est sur de la glace et qu'il est poussé, il glissera jusqu'à ce qu'il rencontre un obstacle ou qu'il ne soit plus sur la glace.

Comme le pingouin, le bloc déplaçable dispose de 4 blocs invisibles placés en croix autour pour détecter dans chaque direction la présence d'éléments qui empêcheraient son déplacement.

### Implémentation

Les blocs déplaçables héritent de « Surface », ils sont implémentés dans la classe « B\_Movable ».

Le bloc utilise 4 QGraphicsRectItem pour détecter les collisions environnantes :

```
QGraphicsRectItem *leftCollideBox;  
QGraphicsRectItem *rightCollideBox;  
QGraphicsRectItem *bottomCollideBox;  
QGraphicsRectItem *topCollideBox;
```

A chaque déplacement de ce bloc par les méthodes « moveBy » ou « setPos », l'on a réimplémentées. Ces blocs de détection des collisions vont se déplacer comme le bloc principal. Tout comme le pingouin, une méthode « addToScene » s'est imposée pour ajouter facilement le bloc et ces blocs de collision à la scène.

Afin de savoir si le bloc est en mesure de se déplacer dans une direction, les méthodes publiques suivantes sont implémentées :

```
bool IsMovableToLeft();  
bool IsMovableToRight();  
bool IsMovableToBottom();  
bool IsMovableToTop();
```

Elles utilisent la méthode privée :

```
bool IsMovable(QList<QGraphicsItem *>);
```

On lui passe la liste des éléments en collision avec le bloc de détection des collisions dans le sens demandé.

Le fonctionnement de cette méthode est relativement simple. On lui passe une liste de `QGraphicsItem*`. Si dans cette liste un élément est d'un type sur lequel le bloc ne peut pas se déplacer, elle retourne « false ». Dans le cas contraire, elle retourne « true ». Les types « interdits » sont : « `B_Wall` », « `B_Movable` », « `E_Renard` », « `E_Loup` ».

### Déplacement et glisse du bloc

Le déplacement des blocs déplaçables se fait avec celui du pingouin, dans la classe « `Gameboard` ».

Il est d'abord nécessaire de savoir si un bloc déplaçable est sur de la glace. La méthode « `bool isSlide()` » est implémentée dans « `B_Movable` » et retourne true si le bloc est sur de la glace.

Le fait que les blocs puissent glisser seuls sans être en collision avec le pingouin introduit une autre problématique. De plus plusieurs blocs peuvent se déplacer en même temps dans des directions différentes.

Il faut donc plusieurs attributs pour enregistrer ces blocs et leurs sens de déplacement pour les déplacer dans un slot appelé par un timer.

- Un attribut pour enregistrer le bloc en collision qui ne glisse pas : `B_Movable *moveBloc;`
- Une structure de données qui représente un bloc qui glisse :

```
struct slideBloc{    B_Movable *slidingMovable;    char sens; //l, r, t, b};
```
- Une liste de blocs qui glissent :

```
QList<slideBloc> listsLindingBlocs;
```

Il faut dans un premier temps enregistrer dans un attribut quel bloc déplaçable il faut déplacer. Ceci se fait dans la méthode « `MovePingouin` » (introduite précédemment). Dans cette méthode on retourne vrai ou faux pour donner ou non l'autorisation au pingouin de se déplacer suivant les blocs en collision.

On ajoute la condition suivante :

```
bool Gameboard::MovePingouin(QList<QGraphicsItem *> CollidingItems, char sensDepl)
{
...
    else if(typeid(*CollidingItems.at(i)).name() == typeid(B_Movable).name())
    {
        B_Movable *b;
        b = dynamic_cast<B_Movable*>(CollidingItems.at(i));

        if(sensDepl == 'l' && b->IsMovableToLeft())
        {
            moveBloc = b;
            bMove = true;
        }
    }
}
```

```

        }
        else if(sensDepl == 'r' && b->IsMovableToRight()){
            moveBloc = b;
            bMove = true;
        }
        ...
    ...
}

```

Si le bloc déplaçable en collision peut se déplacer dans la même direction que le pingouin on l'ajoute dans l'attribut de Gamebord « moveBloc » et on autorise le pingouin à se déplacer.

Ensuite il faut déplacer le bloc en question juste après avoir déplacé le pingouin.

Ceci se fait par l'appel de la méthode « void MoveBloc(char sens) » dans la méthode « keyPressEvent » (où sont gérés les déplacements du pingouin) toujours dans Gameboard.

Cette méthode procède en plusieurs étapes :

1. Elle déplace le bloc en collision (« moveBloc ») dans le sens demandé (le même que le déplacement du pingouin). Le sens est donné par l'intermédiaire du paramètre « char sens ».

```

void Gameboard::MoveBloc(char sens)
{
    switch(sens)
    {
        case 't':
            moveBloc->moveBy(0,-1);
        break;
        case 'b':
            moveBloc->moveBy(0,1);
        break;
        case 'l':
            moveBloc->moveBy(-1,0);
        break;
        case 'r':
            moveBloc->moveBy(1,0);
        break;
    }
    ...
}

```

2. Appel de la méthode « fixeMovable(B\_Movable \*b) » qui coule le bloc si il se trouve dans l'eau ou qui le transforme en mur si il se place sur une case de transition.
3. Si le bloc est sur de la glace (isSlide()==true) on crée un élément de type « SlideBloc » et on l'ajoute à la liste de blocs qui glissent (listSlindingBlocs).

```

if(moveBloc->isSlide())
{
    slideBloc sb;
    sb.slidingMovable = moveBloc;
    sb.sens = sens;
    listSlindingBlocs.append(sb);
}

```

4. Enfin on démarre le timer qui se charge du déplacement des blocs qui glissent et on supprime le pointeur sur le bloc en déplacement « moveBloc ».

```
    timerBlocDeplSlide->start(SLIDE_SPEED);
}

moveBloc = NULL;
}
```

Maintenant il reste à aborder le déplacement des blocs en glissement à proprement dit. Tout se passe dans le slot « SlideBloc() » qui est connecté au timer « `timerBlocDeplSlide` ».

On parcourt l'ensemble des blocs qui sont en train de glisser. La variable « `removeBloc` » est utilisée pour déterminer si un bloc doit être retiré de la liste des blocs en glissement.

```
void Gameboard::SlideBloc()
{
    for(int i=0; i<listSlindingBlocs.size(); i++)
    {
        B_Movable* SlidingBloc = listSlindingBlocs.at(i).slidingMovable;
        bool removeBloc = true;
```

Si le bloc est toujours sur de la glace (`isSlide`) on continue dans nos investigations pour savoir si on va retirer le bloc de la liste. Suivant le sens de déplacement du bloc on teste s'il peut se diriger dans la direction voulue. Si il peut, on le déplace, on vérifie si il n'est pas tombé dans l'eau ou autre (`fixemovable`) et on passe la variable « `removeBloc` » à false pour que le bloc ne soit pas retiré de la liste.

```
if(SlidingBloc->isSlide())
{
    switch (listSlindingBlocs.at(i).sens)
    {
        case 't':
            if(SlidingBloc->IsMovableToTop())
            {
                SlidingBloc->moveBy(0,-1);
                fixemovable(SlidingBloc);
                removeBloc = false;
            }

            break;

        case 'b':
        ...
    }
}
```

(Mêmes opérations pour les différentes directions)

On termine notre boucle de parcours en retirant les blocs qui ne sont plus sur la glace ou qui ne peuvent plus bouger.

```
    if(removeBloc)
    {
        listSlindingBlocs.removeAt(i);
    }
}
```

On termine la méthode en stoppant le timer si la liste de bloc en déplacement est vide :

```
if(listSlindingBlocs.size() == 0)
{
    timerBlocDepSlide->stop();
}
}
```

### fixeMovable

« void fixeMovable(B\_Movable \*b) » détecte avec quoi est en collision un bloc déplaçable.

- Collision avec de l'eau : Le bloc en collision sera supprimé et remplacé par un bloc de neige au même endroit (Le bloc déplaçable coule dans l'eau).
- Collision avec une case de transition : Le bloc sera remplacé par un bloc de type « B\_Wall » et un message sera affiché.
- Collision avec un objet : L'objet sera supprimé. S'il s'agit d'un objet indispensable, un pop-up est affiché et l'éénigme recommencée.

## Ennemis

### Réflexion

Un ennemi est une sentinelle qui suit un chemin préconfiguré. Un ennemi dispose d'un champ de vision et s'il détecte le joueur dans ces champs ou si le joueur entre en collision avec cette dernière, il est détecté, perd une vie et recommence le niveau.

Le champ de vision est adaptable, c'est-à-dire qu'un ennemi ne doit pas pouvoir voir au-delà d'un mur ou d'un bloc déplaçable, ils doivent obstruer sa vue.

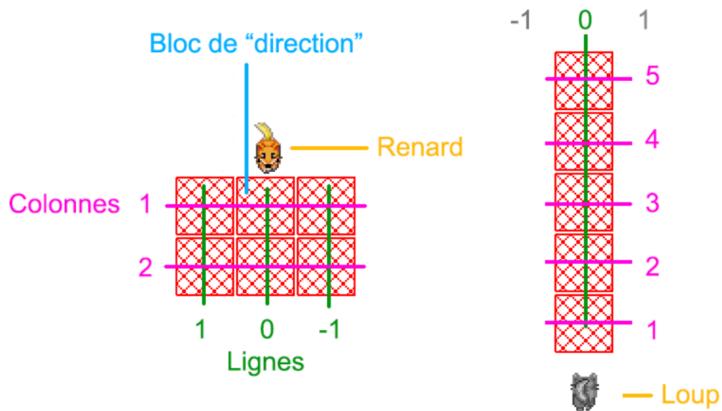
Il y a plusieurs types d'ennemis, ils ne varient pas beaucoup d'un à l'autre. Leur apparence change, ainsi que la forme de leur champ de vision et également leur vitesse de déplacement.

### Champ de vision

Dans un premier temps il était prévu d'implémenter le champ de vision d'un ennemi en forme de demi-cercle d'un certain rayon. L'angle d'ouverture du demi-cercle changerait en fonction des blocs obstruant le champ de vision.

Cette solution nous semblait être la plus parfaite, cependant on a très vite compris que cette idée était très complexe à mettre en place et que très peu de temps restait pour cette dernière.

La solution retenue est de décomposer le champ de vision en une multitude de blocs disposés devant l'ennemi. Les blocs de détection sont assignés à une colonne et une ligne dont le numéro est établi comme sur la capture suivante :

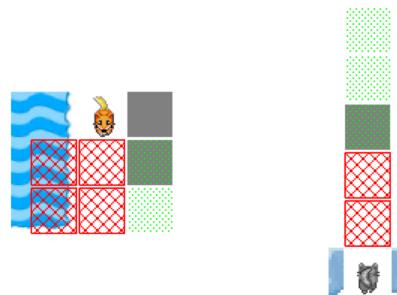


Ces nombres ont été choisis pour faciliter le positionnement des blocs et pour permettre une désactivation des blocs si un élément vient obstruer le champ de vision.

Au départ il était prévu que le système permette des champs de visions larges (plus de 3 lignes) mais finalement, pour le gameplay un champ de vue supérieur en largeur n'apporte rien, cela n'a donc pas été implémenté.

L'idée est la suivante : Si un bloc de type « mur » ou « déplaçable » est en collision avec le bloc du champs de vision B(colonne, ligne) de l'ennemi, tous les blocs :

- $B(>\text{colonne}, \text{ligne})$  seront désactivés. (implémenté)
- Si le bloc de direction est obstrué, tous les autres blocs sont désactivés ! (implémenté)
- Si  $\text{ligne} > 0$  :  $B(>\text{colonne}, \geq \text{ligne})$  seront désactivés. (non implémenté)
- Si  $\text{ligne} < 0$  :  $B(>\text{colonne}, \leq \text{ligne})$  seront désactivés. (non implémenté)



*Les blocs verts sont désactivés, le pingouin peut s'y aventurer sans risque. Par contre attention aux blocs rouges !*

## Déplacement

L'ennemi essaye de rejoindre les points attribués. Ils sont codés sous forme de QPoint enregistrés dans une QList. L'ennemi apparaît toujours sur son point de départ. Ensuite il va rejoindre le point suivant de sa liste jusqu'au dernier, puis il va recommencer en rejoignant le point 0.

Pour se faire, il commence par rallier la position en X du QPoint de destination. Si le point.x est plus grand que sa position en x, il va se déplacer contre la droite. Dans le cas contraire, il se déplacera contre la gauche. Dès que sa position en x est égale à celle du point de destination, il fait la même démarche mais avec les coordonnées en y. Dès que sa position == à la position du point de destination, il passe au point suivant.

Mais si l'ennemi rencontre un obstacle sur son chemin, il va s'arrêter et commencer à faire son chemin dans le sens inverse. Il ne va en aucun cas essayer de contourner l'obstacle.

Le bloc de « direction » est celui utilisé pour détecter une collision et empêcher l'ennemi de continuer son chemin. Il est toujours positionné en colonne = 1 et ligne = 0.

### *Implémentation*

La classe Ennemi hérite de « QGraphicsRectItem ». La première idée était de créer des timers internes à la classe Ennemi, mais cette possibilité a rapidement été abandonnée car l'intégration de timer, de slot et de connect dans une classe nécessite l'héritage de « QObject », chose compliquée à mettre en place en plus de l'héritage de « QGraphicsRectItem » obligatoire.

On s'est donc inspirés de l'exemple de Qt « CollidingMouse ». Il connecte le slot « advance » de la scène à un timer. À la fin du timer le slot appelle toutes les méthodes « advance » implémentées dans les QGraphicsItem se trouvant sur la scène.

Nous avons donc décidé d'utiliser cette technique. La classe Ennemi hérite alors de « QGraphicsItem ». Les méthodes propres à cet héritage ont été implémentées tout comme dans la classe « Player » (boundingrect, paint).

Structures de données et attributs nécessaires :

- Chaque bloc du champ de vision est associé à une ligne et une colonne ainsi qu'une variable boolean qui définit son état :

```
struct ViewBloc{
    QGraphicsRectItem *bloc;
    int ligne;      //.. -2, -1, 0, 1, 2 ..
    int colonne;   // 0,1,2..
    bool actif;
};
```

- Une liste contenant tous les blocs constituant le champ de vue : QList<ViewBloc> **champVue**;
- Une liste contenant le chemin de points que l'ennemi tente de rejoindre : QList<QPoint> **path**;

- Un attribut contenant dans quel objet de la classe Gameboard se trouve l'ennemi. Ceci est indispensable pour que le pingouin perde une vie si l'ennemi le repère.  
Gameboard \*game;
- Un attribut donnant la vitesse de l'ennemi ainsi qu'un autre que l'on va incrémenter jusqu'à obtenir un multiple de la vitesse et déplacer l'ennemi

```
int speed;
int time;
```

L'attribut time est initialisé dans le constructeur avec un chiffre aléatoire entre 0 et speed (par un rand). Ceci afin d'obtenir des mouvements différents pour chaque ennemi (qu'ils ne bougent pas tous en même temps).

Un ennemi ne peut pas être instancié seul, les classes héritant de cette dernière règlent les paramètres de vitesse, d'apparence et crée le champ de vision de l'ennemi.

### Positionnement

Les méthodes « moveBy » et « SetPos » ont été implémentées. Ces deux sont confrontées à la problématique de la position du champ de vision.

Par contre, elles ne doivent pas placer les blocs du champ de vision correctement en fonction de la ligne et de la colonne, ni prendre en compte l'orientation du méchant. Ceci est fait dans les méthodes « set\_Orientation\_--- »

Pour se faire une méthode pour le positionnement des blocs a été créée :

```
void Ennemi::setPosViewBloc(QGraphicsRectItem* bloc, QPoint p)
{
    int gs = Gameboard::getGameSquares();
    QPoint posEnnemi = convertPosPoint(this->pos());

    bloc->setPos(posEnnemi.x()*gs + p.x()*gs+1, posEnnemi.y()*gs + p.y()*gs+1);
}
```

En fonction de la ligne et de la colonne du bloc passées en paramètre dans « QPoint p », le bloc se positionne correctement.

Exemple, pour positionner le champ de vision à gauche de l'ennemi :

```
void Ennemi::setOrientation_right()
{
    orientation = 'r';
    foreach (ViewBloc vb, champVue)
    {
        setPosViewBloc(vb.bloc, QPoint(vb.colonne, vb.ligne));
    }
    update();
}
```

Dans cette méthode on met à jour l'attribut « orientation » qui définit dans quel sens est l'ennemi et on fait une mise à jour de l'apparence du GraphicsItem avec update qui appelle la méthode paint() et change l'image de l'ennemi suivant son orientation.

On voit que l'on parcourt tous les blocs du champ de vue et on les positionne selon leur ligne et colonne.

Mais la ligne et la colonne doivent être entrées suivant l'orientation de l'ennemi. Pour positionner l'ennemi dans une direction il faudra appliquer le changement suivant à la position des blocs du champ de vue :

- droite : setPosViewBloc(vb.bloc, QPoint(vb.colonne, vb.ligne)); (Défaut)
- gauche : setPosViewBloc(vb.bloc, QPoint(-vb.colonne, -vb.ligne));
- haut : setPosViewBloc(vb.bloc, QPoint(vb.ligne, -vb.colonne));
- bas : setPosViewBloc(vb.bloc, QPoint(-vb.ligne, vb.colonne));

### Déplacement

Les déplacements de l'ennemi se font dans la méthode « advance() » qui est appelée 10\*2 fois par seconde (timer connecté dessus à 100ms).

La méthode advance(int) est appelée deux fois toutes les 100ms, une fois avec en paramètre 1 et une fois avec 0. On va donc commencer à utiliser qu'un seul des deux appels :

```
void Ennemi::advance(int step)
{
    if(step == 1)
    {
        viewBlocActif();
        pinguinDetection();
```

10 fois par seconde on appelle les méthodes « viewBlocActif() » qui désactive les blocs obstrués par un mur et on teste la détection du pingouin avec « pinguinDetection() ». La vitesse de mise à jour n'a pas été choisie par hasard, des tests ont été effectués et cela semble être la vitesse la plus lente possible pour que le pingouin soit détecté dans toutes les situations.

Nous voulions la vitesse la plus lente pour réduire au mieux l'utilisation du CPU. (Ces méthodes sont traitées dans la partie suivante)

```
if(time % speed==0 && !detectPinguin)
{
    time = 0;
    QPoint posEnnemi = convertPosPoint(this->pos());
```

Ensuite on regarde si l'attribut time est un multiple de la vitesse de cet ennemi, si c'est le cas on va procéder à un mouvement de l'ennemi ! On réinitialise alors time à 0. La méthode « convertPosPoint » convertit en coordonné par case de 32px toutes positions par pixel de la scène.

## James Gouin et la banane sacrée

Si le pingouin est détecté, l'ennemi ne bouge plus.

Maintenant on détermine dans quelle direction devrait être le méchant, elle est enregistrée dans « direction ». La logique est la même que celle décrite dans la partie « réflexion » de ce chapitre.

```
char direction = orientation;
if(path.at(iDestPoint).x() > posEnnemi.x())
{
    direction = 'r'; // la direction voulu avant de marcher
}
else if(path.at(iDestPoint).x() < posEnnemi.x())
{
    direction = 'l';
}
else if(path.at(iDestPoint).y() > posEnnemi.y())
{
    direction = 'b';
}
else if(path.at(iDestPoint).y() < posEnnemi.y())
{
    direction = 't';
}
```

Si la direction idéale n'est pas la même que l'orientation actuelle du joueur il va falloir tourner l'ennemi durant cette occurrence (et ne pas le déplacer). Le problème ici est de prendre en compte l'orientation de l'ennemi pour qu'il tourne dans la bonne direction.

```
if(direction != orientation) //l'orientation n'est pas bonne
{
    // Le point est à ma gauche ou à ma droite ?
    if(orientation == 't')
    {
        if(path.at(iDestPoint).x() > posEnnemi.x())
        {
            //tourne à SA droite
            setOrientation_right();
        }
        else if(path.at(iDestPoint).x() < posEnnemi.x())
        {
            //tourne à SA gauche
            setOrientation_left();
        }
        else if(path.at(iDestPoint).y() > posEnnemi.y())
        {
            //On se retourne (Toujours par SA droite )
            setOrientation_right();
        }
    }
    // ... Même raisonnement pour les autres orientations
}
```

Si la direction est égale à l'orientation, alors on va procéder à un déplacement de l'ennemi. On procède comme décrit dans la partie « Réflexion » de ce chapitre.

## James Gouin et la banane sacrée

La méthode « collide » teste si le bloc de « direction » de l'ennemi est en collision avec un mur ou un bloc déplaçable (L'ennemi n'est pas capable de le déplacer). Si c'est le cas on change de sens (attribut « bool sens » initialisé à true dans le constructeur).

L'attribut iDestPoint donne l'id du point que l'ennemi doit rejoindre.

La méthode « nextPoint() » renvoie l'id du point suivant, en fonction de l'attribut « sens » et du « iDestPoint » actuel.

```
else
{
    //déplacement en x en premier puis en y
    if(path.at(iDestPoint).x() > posEnnemi.x())
    {
        if(!collide())
        {
            this->moveBy(1,0);
        }
        else
        {
            sens = !sens;
            iDestPoint = nextPoint();
        }
    }
    else if(path.at(iDestPoint).x() < posEnnemi.x())
    {
        if(!collide())
        {
            this->moveBy(-1,0);
        }
        else
        {
            sens = !sens;
            iDestPoint = nextPoint();
        }
    }
    else if(path.at(iDestPoint).y() > posEnnemi.y())
    {
        ...
    }
    else if(path.at(iDestPoint).y() < posEnnemi.y())
    {
        ...
    }
    else //on est arrivé sur le point de destination
    {
        iDestPoint = nextPoint();
    }
}
```

On termine par rappeler les deux méthodes nécessaires à la détection pour que le joueur soit détecté directement après le déplacement de l'ennemi. Et incrémentation de « time »

```
viewBlocActif();
pinguinDetection();
```

```
        }
        time++;
    }
}
```

### Détection du pingouin, activation du champ de vue

Avant toute détection du pingouin il faut effectuer une mise à jour des blocs actifs du champ de vision. Cela se fait dans la méthode « viewBlocActif() ». Cette méthode a un comportement fidèle aux attentes décrites dans la partie « Réflexion ».

L'optimisation de cette méthode a demandé beaucoup d'importance car est appelée 10 fois pas seconde pour chaque ennemi. La meilleure solution a été l'utilisation d'itérateurs au lieu de « at() » pour parcourir les QList. Les parcours de boucle inutiles ont été réduites au minimum.

Malgré tout, cette méthode exécutée 10 fois par seconde est relativement lourde. En partie en raison de la structure de donnée choisie (List) et des boucles qu'elle nécessite pour la parcourir.

Pour la méthode « pinguinDetection() », on parcourt la liste de bloc du champs de vue et on vérifie si un bloc actif est en collision avec le pingouin.

### Positionner les Proxys

#### Réflexion

L'interface utilisateur devait afficher les éléments importants du jeu (vies, items dans la sacoche, et la scène). La superposition de QWidgets a été retenue. Le Gameboard est donc le QWidget principal et les QWidgets des vies et de la sacoche sont des proxys appartenant à la scène. Il a semblé judicieux de placer la vie en haut à gauche de la scène et la sacoche en bas à droite (commun dans les jeux vidéo).

#### Implémentation

Le positionnement des deux proxys (vie et sacoches) est calculé automatiquement par rapport à la taille du QWidget. Les méthodes setPositionTop(QWidget), setPositionCenter(QWidget) et setPositionBottom(QWidget) font un setGeometry() sur le QWidget envoyé en argument pour le positionnement respectivement Haut, Centre et Bas.

La mécanique des QGraphicsScene impose l'utilisation des QGraphicsWidgetProxy lorsqu'on utilise un QGraphicsScene en tant que QWidget principal.

### Menu Principal

#### Réflexion

Ceci est la première interface avec laquelle l'utilisateur interagit, elle doit donc être la plus simple et compréhensible possible.

Le joueur peut choisir un profil existant ou créer un nouveau profil. Il a été choisi arbitrairement de limiter le nombre de profils maximum à 5. Cependant la décision de ne pas permettre une suppression de profil est voulue, et motivée par l'argument qu'un utilisateur peut toujours revenir au niveau de l'île et ainsi recommencer les énigmes à l'infini. Il y a peu d'intérêt dans l'état du jeu actuel de supprimer son profil.

### **Implémentation**

Le menu principal est composé de QWidgets et d'un QGraphicsScene. Le QWidget principal (MainGame) implémente un QGraphicsScene pour permettre au jeu d'avoir un mode plein écran.

Un QWidget placé au-dessus du QWidget principal s'occupe de la gestion des profils (MenuStart). Le chargement et les sauvegardes sont effectués en utilisant un fichier JSON. Une fois le profil chargé, MainGame va construire le Gameboard et le placer au-dessus de lui avec le focus. La partie peut ainsi commencer.

## **Profil et Système de sauvegarde**

### **Classe profil**

Un profil correspond aux données de l'utilisateur chargées depuis la sauvegarde. Il contient les attributs suivants :

```
QString username;
QString startDate;
QString saveDate;
QString gameTime;
QString loadDate;
int level;
QList<int> power;
int nbLive;
int difficulty;
```

A ce stade du jeu, les éléments comme la difficulté et les pouvoirs sont préparés dans le système de sauvegarde mais ne sont pas utilisés dans le jeu.

L'attribut power est défini comme une liste de chiffres établi à 1 ou 0, où 1 définit un pouvoir acquis et 0 un pouvoir non acquis.

Il contient également les fonctions `void read(const QJsonObject &json)` et `void write(QJsonObject &json)`. La première utilisée pour modifier les éléments de l'objet courant à partir d'un QJsonObject tandis que le second modifie l'objet QJsonobject reçu en paramètre en lui donnant les paramètres de l'objet courant.

### **Classe MenuStart**

Il s'agit du menu principal du jeu qui permet de gérer la liste des joueurs, l'enregistrement et le chargement des parties.

### *Implémentation*

#### **Affichage des parties existantes :**

Pour afficher les boutons de lancement, il est nécessaire de lire une première fois le fichier de sauvegarde et de récupérer la liste des utilisateurs enregistrés. La fonction `bool MenuStart::getProfil()` s'occupe de cela. Ensuite, on utilise un `QSignalMapper` pour lier les boutons au slot `void loadGame(QString value)`, car le nombre de boutons est variable.

#### **Charger une partie existante :**

Lorsque l'utilisateur clique sur un bouton pour charger son compte, on fait appel au slot `void loadGame(QString value)`. Celui-ci lit le fichier, puis utilise la fonction « `read` » de `Profil`. Enfin, il émet le signal `void startGame(Profil* user)` de la classe `MainGame` avec en paramètre l'utilisateur chargé.

```
Profil* user = new Profil();
user->read(object[value].toObject());
emit startGame(user);
```

#### **Création d'une nouvelle partie :**

Lorsque l'utilisateur clique sur le bouton « Nouvelle partie », il peut insérer son surnom et créer une partie. Cela fait appel à la fonction `void newGame()`.

Il lit le fichier, charge toutes les anciennes parties (pour ne pas les perdre), puis écrit à nouveau dans le fichier. La lecture préalable du fichier est obligatoire, car écrire à la suite ne fonctionne pas selon les principes d'écriture JSON.

Une fois l'enregistrement terminé, il charge la partie avec l'émission du signal `void startGame(Profil* user)` de la classe `MainGame`.

#### **Sauvegarde d'une partie**

La sauvegarde agit comme la création d'une nouvelle partie, à la différence que l'on reçoit un utilisateur déjà créé en paramètre. On modifie la durée de jeu, puis enregistre les éléments dans le fichier.

## **Menu Pause**

### *Réflexion*

Traditionnellement, le joueur doit être capable de pouvoir prendre une pause dans un jeu vidéo, il était donc important d'intégrer cette option, et également proposer d'autres options importantes pour une expérience joueur la plus agréable possible pour un jeu de type réflexion/puzzle.

Il était indispensable de permettre au joueur de recommencer un niveau, car il est possible de ruiner toutes les chances de finir le niveau (par exemple, en collant contre un mur un bloc indispensable).

Il a été choisi de permettre au joueur de retourner sur l'île principale (où le joueur peut choisir quel niveau jouer) depuis n'importe quel niveau, sauf celui du tutoriel. Dans un jeu de réflexion, le joueur peut perdre plus facilement patience et vouloir essayer de faire autre chose.

On peut quitter la partie en cours à tout moment, et ainsi retourner au menu principal et choisir un autre profil, ou quitter le jeu.

### *Implémentation*

Ce menu est un QWidget posé en tant que Proxy sur le Gameboard (QWidget principal contenant la scène). Les proxys sont gérés par le QGraphicsFramework pour permettre la superposition de QWidget sur une scène.

Comme son nom l'indique, ce menu apparaît lorsque le jeu est en pause, ce qui veut dire que le jeu doit se geler lors de la présence de ce menu. Le timer pour le glissement du personnage est mis en pause.

```
timerPingouinSlide->stop();
timerPingouinSlide->start(SLIDE_SPEED);
```

Le joueur ne doit pas être capable d'interagir avec la partie à l'aide de son clavier. L'utilisation d'un toggle pour ce contre évènement a été nécessaire. Ce toggle est utilisé dans les évènements du clavier (keyPressEvent(QKeyEvent)) pour bloquer la lecture des touches. Il aurait été sûrement possible d'utiliser grabKeyboard() et releaseKeyboard() de la classe QWidget à la place du toggle, mais ces méthodes ont été découvertes après l'implémentation des toggles (manque de temps pour faire les tests nécessaires).

```
toggleMenuPause = true;
```

Le design du menu est important pour ne pas faire un choc à l'utilisateur lorsqu'il l'appelle. Étant un QWidget la customisation est faite aisément avec la méthode lui appartenant.

```
this->setStyleSheet();
```

Quand on reprend la partie, on dégèle la partie. Le timer est redémarré et le clavier est libéré. Ainsi notre personnage peut continuer son aventure.

Recommencer l'énigme et recommencer le niveau sont expliqués au point suivant.

Lorsque le joueur choisit de quitter la partie en cours, il a le choix entre sauvegarder son avancement dans le jeu ou non. S'il décide de sauvegarder, son profil sera mis à jour avec son avancement dans le jeu actuel (nombre de vies, nombre de morceaux du pont récupérés, et le niveau sur lequel il se trouve), sinon le profil n'est pas mis à jour. Après le choix, il retourne sur le menu principal.

Les fonctions pour reprendre la partie, redémarrer la partie, redémarrer le niveau, retourner sur l'île principale, et quitter la partie sont toutes connectées au parent (le Gameboard) qui

## James Gouin et la banane sacrée

gère ces évènements. En fin de compte, le menu pause est juste une interface (Proxy) posée sur la scène du jeu.

### Recommencer l'énigme, recommencer le niveau et checkpoints

#### Checkpoints

##### Réflexion

Le joueur doit pouvoir recommencer l'énigme à tout moment. Afin qu'il ne recommence pas le niveau entier à chaque fois, il était nécessaire d'implémenter des checkpoints. Ils enregistrent la position du pingouin. Les checkpoints sont sauvés à chaque changement de vue. Ainsi, une énigme équivaut à un checkpoint.

##### Implémentation

Le checkpoint est représenté sous la forme d'un QPoint. Deux fonctions sont liées :

```
void saveCheckpoint();
void loadCheckpoint();
```

La première permet de sauver dans le QPoint la position du pingouin. La seconde de la recharger.

#### Recommencer le niveau et le jeu

##### Réflexion

Le joueur peut recommencer l'énigme ou le niveau à tout moment. Cependant, cela lui coûte une vie. Il peut le faire depuis le menu escape, ou depuis la touche clavier « 0 ».

Lorsque le joueur recommence l'énigme, il doit recommencer au checkpoint précédent. Cela signifie qu'il perd les objets obtenus entre l'enregistrement du checkpoint et le redémarrage tout en gardant ceux des énigmes précédentes du niveau.

Dans le cas d'un redémarrage du niveau, il retrouve tous les paramètres du fichier de sauvegarde.

##### Implémentation

Deux slots sont utilisés. Ils sont utilisés comme fonction par certaines méthodes et sont connectés aux boutons du menu pause.

```
public slots:
    void restartEnigma();
    void restartLevel();
```

En recommençant l'énigme, le personnage est repositionné au dernier checkpoint et toutes ses actions sur l'environnement (bouger les blocs) sont annulés. La fonction vérifie que le joueur a encore des vies. Si ce n'est pas le cas, il appelle la fonction restartLevel() en attribuant 3 nouvelles vies au joueur.

## James Gouin et la banane sacrée

Le redémarrage du niveau positionne le joueur dans la première énigme du niveau sur lequel il se trouve. A nouveau, toutes ses actions sont annulées.

Pour annuler les actions du joueur, la solution appliquée est la suppression de tous les QGraphicsItems de la scène et la repopulation d'une nouvelle scène. Cette solution est acceptable, car le joueur ne retournera jamais sur ses pas dans une énigme déjà résolue.

Comme une nouvelle scène est recréée, il est nécessaire de :

- Supprimer tous les items de la scène : removeAllItems()
- Déconnecter le timer de l'ancienne scène : disconnectTimer()
- Définir les proxys sur la scène et les positionner : setProxy()

## Conclusion

Notre cahier des charges fut assez gourmand en fonctionnalités. Mais au final, le jeu obtenu est fonctionnel. Certaines fonctionnalités n'ont pas été implémentées par manque de temps et certains mécanismes du jeu ont évolués lors du développement vers des solutions plus mûres et réfléchies.

Les tâches indispensables pour le fonctionnement du jeu ont été priorisées. Les mécanismes les plus importants ont été implémentés et sont fonctionnels. Le jeu, à l'état actuel est jouable pour les premiers niveaux. Il sera également facile par la suite de générer et ajouter de nouveaux niveaux grâce à la population automatique des cartes de jeu.

De nombreux problèmes de compatibilité inter-plateformes sont apparus durant le développement, mais des solutions ont été trouvées et au final l'application est compatible Mac et Windows. On peut avancer que la diversité des OS lors du développement a été profitable et a obligé de se poser des questions de compatibilités plus avancées que lors d'un développement sur une seule plateforme.

Au fil de l'avancement dans le projet de nouvelles et meilleures solutions ont été trouvées aux fonctionnalités précédentes. Cependant, elles n'ont pas toutes pu être ré implémentées par manque de temps. On peut cependant remarquer que lors du développement du projet, les solutions apportées sont de plus en plus propres et mûres.

Par exemple, l'héritage de la classe « Player » dans la classe « P\_Penguin » n'était pas indispensable, les méthodes et les éléments (QGraphicsRectItem) de « P\_Penguin » auraient pu être implémentés dans « Player » comme fait plus tard avec « Ennemi ».

Une solution moins gourmande en mémoire et CPU aurait également pu être trouvée pour les champs de détection des ennemis.

L'espoir de pouvoir continuer ce projet anime l'équipe de Banana Rocket, implémenter les fonctionnalités qui ont été mises en suspend et finir les niveaux manquants.

## Bibliographie

GHORBEL Hatem, CHATELAIN Pierre & GRUNENWALD David, Programmation événementielle avec Qt 5, HES-SO, 2013.

Communauté Qt, Documentation, Qt-Project, qt-project.org

Communauté, Stack Overflow, stackoverflow.com

Communauté, OpenClassrooms, openclassrooms.com

## Annexes

- Cahier des charges
- Spécifications détaillées prévues
- Spécifications détaillées implémentées
- Tâches
- Diagramme de Gantt