# 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 craintif de l’eau, il est impossible pour lui de mouiller son nœud papillon qui lui fût donné par son maitre Poissonnier.

Un jour, alors que notre héro pêchait, il reçu par une fusée glaçon un nouvel ordre de mission…

Contenu du message glacé :

**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-fondera. Nous nierons tout lien avec cette mission. Vous êtes maintenant seul. Bonne chance.

Le pingouin décide d’accepter la mission et par aussi tôt 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 atterrie à quelque mettre de l’entrée de le 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 pêcher.

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

# Conclusion

Notre cahier des charges fut assez gourmant en fonctionnalités. Mais au final, nous avons obtenu un jeu fonctionnel, certaines fonctionnalités n’ont pas été implémentées par manque de temps et certains mécanismes du jeu ont évolué lors du développement vers des solutions plus mûres et réfléchies.

Les tâches les plus importantes pour le fonctionnement du jeu ont été priorisées, les mécanismes les plus importants du jeu ont été implémentés et sont fonctionnels. Nous avons aujourd’hui un jeu jouable pour les premiers niveaux. Il sera également facile par la suite de générer et ajouter de nouveau grâce à notre population automatique des maps.

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 à obligé se poser des questions de compatibilités plus avancées que lors d’un développement sur une seule plateforme.

Au fil de l’avancement dans projet de nouvelles et meilleures solutions ont été apporté, cependant la plupart des anciennes solutions déjà implémentées n’ont pas eu le temps d’être modifiées par manque de temps. On peut donc remarquer que lors du développement du projet, le niveau et les solutions apportées sont de plus en plus propres et mûrs.

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 peu être implémenté dans « Player » comme nous l’avons 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 mis en suspend et finir les niveaux manquants.

# 

# Conception

## Planning et suivi

On a utilisé la forge BITBUCKET avec la technologie GIT. Nos commits sont datés et considéré comme le journal de bord le l’équipe (commentés et contenant les modifications et ajouts).

Nous étions au début partie sur une méthodologie de développement en cascade, cependant il s’est avérée que nous travaillions plus avec une méthodologie agile sans pour autant le faire exprès (avant de prendre connaissance de cette methode lors du cours de Génie Logiciel). La structure de notre projet était en effet composée d’un Product Backlog avec toute nos fonctionnalité du cahier des charges. D’un Sprint Backlog qu’on se donnait tous lundi pendant les heures du projet. Les sprints en eux même qui durait une semaine, du lundi au lundi. Et finalement tout les lundis 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 a été la seule à travailler sur une classe, les cas où ça s’est produit ont été par manque de temps.

Au final le travaille à été reparti ainsi :

|  |  |
| --- | --- |
| **Nom** | **Classes / Autres** |
| Margaux | Gameboard, S\_ViewTransition, Profil, M\_MenuStart, Object, Level, W\_Object, S\_Dialog, W\_Dialog, W\_Life, ItemPopulation, Enimes, Design |
| 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, B\_Movable, B\_Wall, S\_Snow, P\_Penguin, B\_Water, Ennemi, S\_Ice, E\_Renard, E\_Loup, Enigmes, Design |

On trouve que le travaille a été reparti équitablement, que tout le monde à pu toucher à tout, et que l’expérience acquise lors de ce projet a été profitable à toute l’équipe, ce qui est le plus important.

# Développement

## 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.

Nous avons dû également ajouter des touches supplémentaires pour d’autres fonctions nécessaires au bon déroulement d’une partie. La touche ESC permet de mettre le jeu en pause, la touche ESPACE permet de valider la lecture des dialogues dans le jeu.

Le clavier est aussi 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 dans notre cas) 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 intercepté 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 profile et lance ainsi une partie.

username = new QLineEdit(this);

username->grabKeyboard();

username->releaseKeyboard();

Lors d’une partie (classe Gameboard), on donne la lecture du clavier lors de la construction du Gameborad. 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 que des méthodes appelant un attribut d’une classe spécifique, car beaucoup d’interactions sont internes au Gameboard et demandaient un contrôle entre plusieurs classes.

## 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 avec un QGraphicsScene). 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’avoir d’interaction avec la partie à l’aide de son clavier. L’utilisation d’un toggle pour ce contre évènement a été nécessaire. Ce toggle est utilisée 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.

En retourner au dernier check point, le personnage recommence au début de l’énigme sur laquelle il se trouve en ce moment, et toutes ses actions sur l’environnent (bouger les blocs) sont annulés. Tous les QGraphicsItems sur la scène sont enlevés et de nouveaux items sont initialisés et placés. Cette solution est acceptable, car le joueur ne retournera jamais sur ses pas dans une énigme déjà résolue.

Le redémarrage du niveau positionne le joueur dans la première énigme du niveau sur lequel il se trouve. Tout comme quand il retourne au dernier check point, toute la scène est réinitialisée avec de nouveau items.

Lorsque le joueur choisit de quitter la partie en cours il a le choix de 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 récupéré, 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 lui gère ces évènements. En fin de compte, le menu pause est juste une interface (Proxy) posée sur la scène du jeu.

## 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, les QWidgets des vies et de la sacoche sont des proxys de la QGraphicsScene du QWidget principal.

Il nous 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 de la QWidget. Nos méthodes setPositionTop(QWidget), setPositionCenter(QWidget) et setPositionBottom(QWidget) font setGeometry() sur le QWidget envoyé en argument pour le positionnement respectivement Haut, Centre et Bas.

La mécanique des QGraphicsScene impose l’utilisation des proxys lorsqu’on utilise un QGraphicsScene en tant que QWidget principal, d’ailleurs, les proxys sont une méthode de cette classe.

## 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 voulu, et motivée par la 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 à notre jeu d’avoir le 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 depuis et dans le même 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.

## Le Personnage

### Réflexion

Le personnage a été créé avec l’obligation de pouvoir interagir avec d’autre QGraphicsItems. Lors des prototypes dans les Play Grounds nous sommes arrivés à une solution pour le déplacement et la détection des collisions du personnage avec une disposition en croix (voir la partie Penguin). Le personnage (Player) est donc celui que le joueur contrôle, et un écosystème de QGraphicsItems autour de lui informe le Gameboard de ce qui se passe. Avec l’expérience acquise, nous avons ainsi créé la classe Ennemi qui intègre le QGraphicsItem central (ici le personnage de l’ennemi).

#### Implémentation

C’est un QGraphicsItem qui contient les informations graphiques, l’orientation et le positionnement sur le Gameboard du joueur.

Voir le Penguin pour plus d’informations.

## Création des Maps (Scènes)

### Réflexion

La recherche pour la création des maps 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 ; nous pouvons donc les facilement les insérer pour créer une map entière à partir de petits morceaux. Notre choix s’est particulièrement porté sur Tiled car il permet de générer des fichiers images PNG ainsi que des fichiers texte contenant la composition précise de tous les éléments de la map (pour pouvoir faire une population de map avec les QGraphicsItems dont on avait besoin).