



## Projet P2 - Glowing Barn

Septembre 2018 - Janvier 2019

Alexandre Bianchi - Pierre Bürki - Loïck Jeanneret

INF2dIm-b

### Superviseurs

David Grunenwald

Stephane Beurret

## Abstract

Dans le cadre du Projet 2 nous avons dû réaliser une application graphique possédant des animations, tout en portant une attention particulière à la gestion du groupe. Pour atteindre ces buts, notre groupe a décidé de développer un jeu vidéo inspiré des effets magnétiques des aimants. Les buts ont été atteints avec la réalisation de "Magn & Cie", un jeu contenant une petite histoire et 12 puzzles sous la forme de cartes 2D.



*Niveau du jeu*

# Table des matières

<b>Abstract</b>	<b>1</b>
<b>Table des matières</b>	<b>2</b>
<b>Introduction</b>	<b>4</b>
But du projet	4
Description du jeu	4
<b>Conception</b>	<b>4</b>
Choix de l'architecture	4
ECS "Officiel"	5
Notre implémentation de ECS	6
Qt et QGraphics	7
Analyse des point chauds	8
Moteur physique (collisions, aimants)	8
Gestion et chargement des fichier maps	8
Système d'animation simple	9
Système d'entité modulable	9
Répartition des tâches	10
<b>Implémentation</b>	<b>10</b>
Caméra	10
Créations des niveaux	11
Découpe d'un fichier de niveau	11
MapEntity	12
LayerEntity	12
ObjectLayerEntity	12
EntityFactory	12
Particules	13
Liste des composants	13
Déboggage avancé	13
Gestion des inputs	14
Système de hitbox	15
Fonctionnement général des aimants	16
Système de boutons	17
Parallaxe	18
Animations	19
Liste des entités	20
Player	20
Portes	21
Box	22

Magnet zipper	23
Composants:	23
Magnet gravity	24
Magnet jumper	25
Problèmes rencontrés	25
Chargement des graphismes	25
Tailles des ressources	26
QSoundEffect	26
Protocole de tests	27
Procédure de test	27
Test hub	27
Test zipper	28
Test aimants	29
Test boîtes	30
Test divers	32
Test boutons	33
Test du jeu	33
"Garde-fou" implémenté dans le code	33
Bugs connus	34
Déploiement	36
Windows / MacOS	36
Linux	36
Changement à effectuer pour de futur projets	36
Critique du planning	37
<b>Conclusion</b>	<b>37</b>
<b>Annexes</b>	<b>38</b>
<b>Bibliographie</b>	<b>38</b>

## Introduction

### But du projet

De nos jours, le monde du jeu vidéo émerge de son adolescence et se voit devenir plus qu'un média de divertissement : une forme d'art et d'expression. De ceci naît le fait que les jeux sont plus pensés, autant en amont qu'en aval, d'un point de vue unique au domaine : le *game design*. Notre groupe étant formé de personnes qui, de leur propre aveu, jouent à de tels jeux, ce point de vue nous intrigue autant qu'il nous paraît être devenu naturel. Nous avons donc utilisé ce projet comme prétexte pour expérimenter la discipline.

Toutefois, hormis l'intérêt que nous exprimons envers le *game design*, le principal exercice qu'aura présenté ce projet, est la réalisation d'un petit moteur le plus versatile et modulable possible afin de répondre à l'objectif principal imposé: réalisation d'une application possédant des animations et travail en équipe. À cette fin, nous avons étudié un *design pattern*, l'*Entity-Component System* (ECS), et avons établi d'utiliser le logiciel *Tiled* dans le but de créer nos niveaux très aisément. Le choix de *Qt* comme *framework* quant à lui, était imposé par les consignes du projet.

Ce sont ces décisions que nous discuterons dans ce document.

### Description du jeu

Magn & Cie est un jeu vidéo se présentant comme un platformer à deux dimensions, présentant quelques mécaniques permettant de petites énigmes. À l'heure actuelle, le jeu comporte une dizaine de niveaux dans lesquels ces mécaniques sont montrées et exploitées de manière intuitive.

Tous les éléments du jeu sont réunis sous le thème des aimants. Les capacités du personnage jouable, hormis le saut et le déplacement latéral sont:

- Actionner des leviers
- Passer des portes
- Activer et désactiver l'aimant qu'il porte, qui permet lorsqu'il est activé les actions suivantes
- Être attiré par des rayons tracteurs (aimants *zipper*)
- Rebondir sur certains aimants (aimants *jumper*)
- Être attiré ou repoussé par des aimants radiaux (aimants *gravity*)
- Porter des caisses magnétisées

Le joueur devra éviter les chutes dans les trous sans fond, les piques et les étincelles agressives (*sparks*), faute de quoi le niveau dans lequel il se trouve recommencera.

## Conception

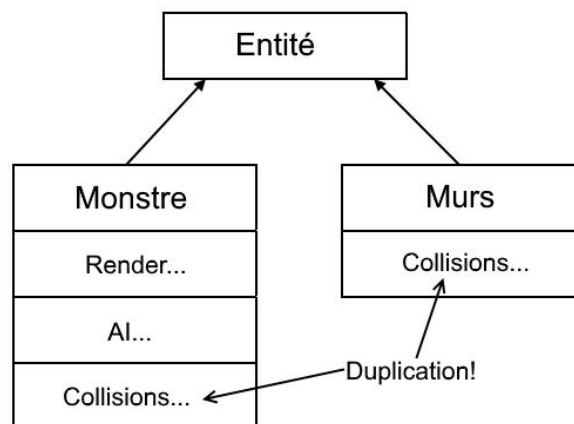
### Choix de l'architecture

Dans une tentative de ne pas foncer dans un mur, nous avons décidé d'essayer d'utiliser un *game design pattern* existant. Nous avons décidé d'utiliser ECS (*Entity Component System*) car il est utilisé par des moteurs de jeu connu tel que *Unity*, le groupe a également été influencé par le livre *Game Programming Patterns* [1].

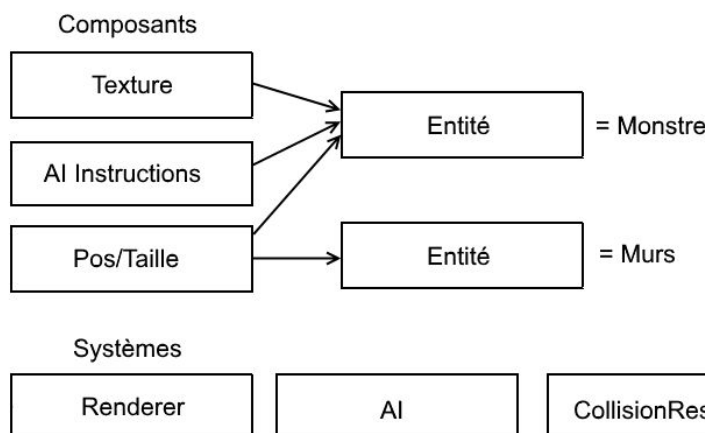
#### ECS "Officiel"

Ce *design pattern* consiste à préférer créer des entités par composition de composants, plutôt que par héritage. Ceci permet de réduire la répétition de code, de rendre les entités modulables et d'augmenter la lisibilité du code en séparant les données de leur traitement. Les *Entity* sont représentés par une liste de *Component* contenant chacun des données. Ces données sont mise à jour par des *System* qui effectuent des actions sur les entités possédant des composants similaires à ce que fait le système.

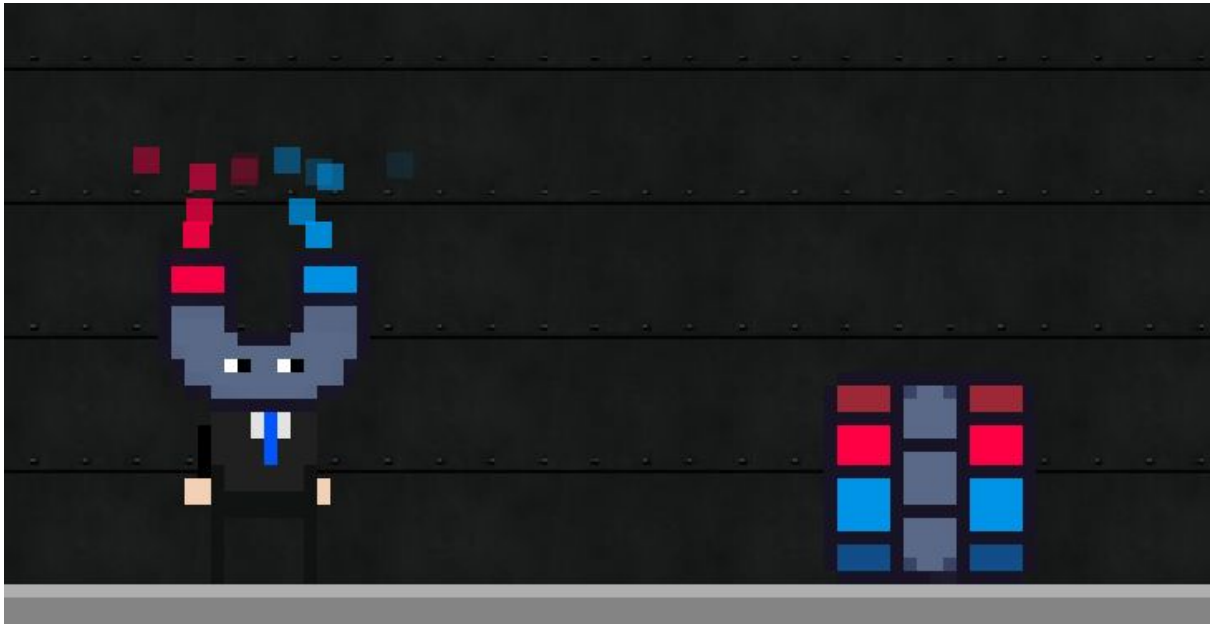
#### Architecture en utilisant de l'héritage:



#### Architecture ECS



## Exemple concret d'utilisation



*Joueur et boîte*

La caisse et le joueur possèdent de nombreux points communs :

- Affectés par la gravité
- Réactions aux *Hitboxes*
- Réactions aux différents aimants

Mais également de nombreuses différences:

- Le joueur peut être contrôlé par le clavier
- Le joueur possède des animations
- La caisse peut presser certaines plaques de pression

Les composants et systèmes pour la gravité, la réaction aux hitboxes et la réaction au boutons possèdent différents paramètres pour chaque entité mais leur code n'est pas dupliqué. Pour les autres comportements non traité, on crée d'autre composants qui peuvent éventuellement être réutilisé.

### Notre implémentation de ECS

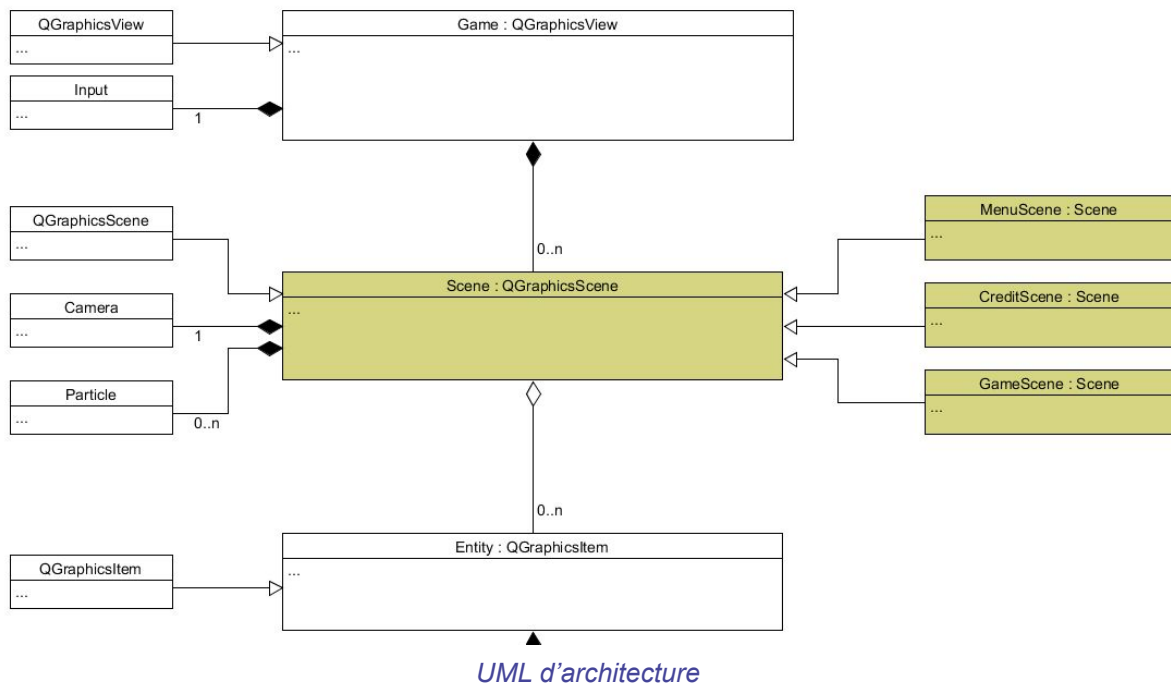
Notre implémentation s'est grandement inspirée du chapitre 14 du livre *Game Programming Patterns* [1], qui mentionne uniquement les *Components* sans faire allusion aux *Systems*. En conséquence notre implémentation n'en possède pas.

Les communications entre composants sont traités dans ceux-ci directement. Chaque composant possède une référence à son entité, et chaque entité possède la méthode publique *GetComponent* permettant d'accéder à chacun de ses composants par nom. C'est par ce biais que, par exemple, le composant des contrôles peut influencer le composant gérant la physique.

Une conséquence de l'utilisation de l'ECS est que nos classes ne se prêtent que très peu au dessin d'un diagramme UML sophistiqué. En effet, à quelques exceptions près, nos composants héritent tous directement de *Component*, ce qui donne un diagramme UML sans surprise ni saveur. Les composants dont la structure d'héritage mérite d'être mentionnée sont ceux dérivés de *HitboxComponent*, mais ceux-ci feront l'objet d'une explication plus loin. Les composants restants sont donc absents de notre diagramme UML, mais sont listés et accompagnés d'une brève description si nécessaire en annexe.

## Qt et QGraphics

Ce *framework* nous étant imposé dans les directives du P2, il nous a fallu revoir nos *Entities* en tant que *QGraphicsItems*, ce qui n'était pas la conversion la plus intuitive. Voici un UML représentant les spécialisation qu'on a faites aux classes nécessaires afin de pouvoir utiliser ECS et *QGraphicsViews* ensemble:





## Analyse des point chauds

Certains points considérés critiques ont été analysé avant l'implémentation. Ce chapitre du rapport y est consacré. Dans cette partie du document seront exposées quatre questions d'architecture logicielle et la solution qu'il aura été décidé d'apporter à chacune.

### Moteur physique (collisions, aimants)

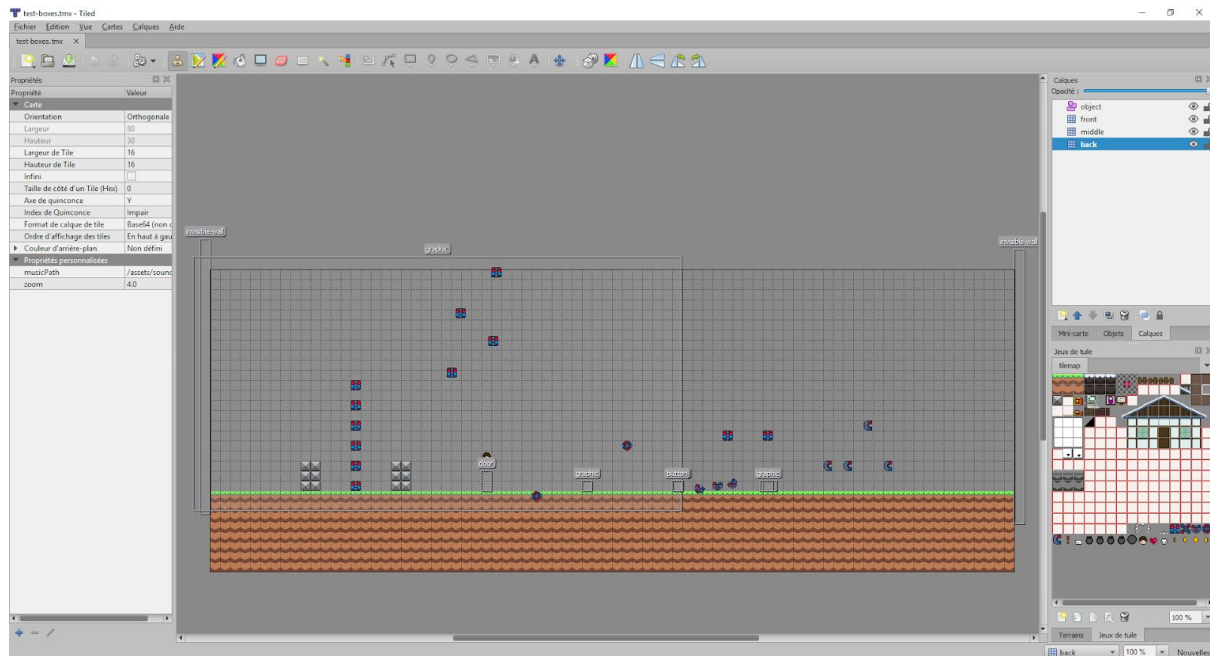
Les mécaniques fondamentales d'un *platformer* méritaient mûre réflexion avant de passer à l'implémentation. En ce qui concerne les collisions, on aura profité des connaissances acquises en amont lors de projets personnels, et décidé de réutiliser les algorithmes de détection et résolution de collisions de ceux-ci.

Les aimants quant à eux ont mérité un prototypage. Nous avons donc utilisé (de façon *quick and dirty*) un projet antérieur réalisé en javascript pour y tester des versions des aimants *gravity* et *jumper*. On peut le trouver ici: <https://orikaru.net/glowing-barn-PROTO>. Le code du prototype est accessible depuis un navigateur mais est difficilement lisible car le seul but était de tester les mécaniques de jeu au détriment de la qualité du code.

### Gestion et chargement des fichier maps

Les spécifications du jeu précisent que le jeu sera une succession de différents niveaux. Il aurait été possible de réaliser directement à la main des fichiers de données ou de réaliser le placement des éléments directement au sein du code source principal de notre programme. Cependant, avec une optique pareil, le code principal se retrouve alors entaché d'un grand nombre de code dupliqué et complique sa compréhension ainsi que sa lecture. Il rendrait également beaucoup plus difficile la création de carte.

Pour palier à ces différents problèmes il a été décidé d'utiliser Tiled pour faciliter la conception des différents niveaux qui se devaient d'être réalisés. En effet, Tiled est un programme "open source" d'édition de niveau développé en Qt lui aussi. Il permet de réaliser des niveaux utilisant des graphismes 2D ou isométriques. Le programme consiste en une interface graphique permettant de placer des éléments graphiques ou des objets auxquels sont attachés des propriétés et de répartir les éléments entre différents calques. Le programme s'occupe ensuite de transcrire les différents niveaux en fichiers de données XML, afin de pouvoir être utilisé par n'importe quel moteur de jeu, langage, etc...



Interface de tiled.

Durant l'analyse de ce point, il a principalement donc été question de savoir comment la map serait décodée à partir des fichiers XML générés par Qt, ainsi que la possibilité d'intégrer le code parsing et l'affichage à notre architecture principale de programme basé sur le design pattern ECS (précisé par la suite).

L'étude préalable consistait à implémenter une solution personnelle de parsing XML, cependant, après de plus amples recherches, il s'avère que Tiled est lui aussi développé grâce au framework Qt. Aussi, à partir de ce constat et sachant que la librairie d'affichage des maps au sein de l'éditeur de Qt est lui aussi Open Source alors notre programme l'utilise directement. Ceci implique cependant qu'une partie de l'affichage est délégué à cette librairie plutôt que notre propre méthode de rendu (N.B. la suite des différents points d'implémentation est traitée dans le chapitre du même nom).

### Système d'animation simple

Bien que présent dans notre liste de points chauds, le système d'animation n'était pas prévu dans l'application zéro confort. Ce système a été listé dans les points chauds à cause d'une méconnaissance du système de ressource de Qt et du fonctionnement des *QPixmap*. Pour faire face à ce point chaud, nous nous sommes mis au clair sur le système de ressource de Qt rapidement au début du projet. La documentation des *QPixmap* a permis de vérifier que les fonctionnalités dont nous avons besoin existaient.

Le système d'animation a été implémenté naturellement après le système de rendu d'image statique.

### Système d'entité modulable

Comme l'ECS est la base de tout notre code, on a gardé ce point chaud pour être traité en dernier, afin que notre expérimentation serve de squelette pour le début de l'implémentation si la qualité en était satisfaisante. Si l'ECS figure parmi nos points chauds, c'est qu'il

s'agissait de la première fois que nous nous y confrontons. Implémenter les classes *Entity* et *Component* aura finalement été simple, et nous avons directement pu commencer l'implémentation à partir de là.

## Répartition des tâches

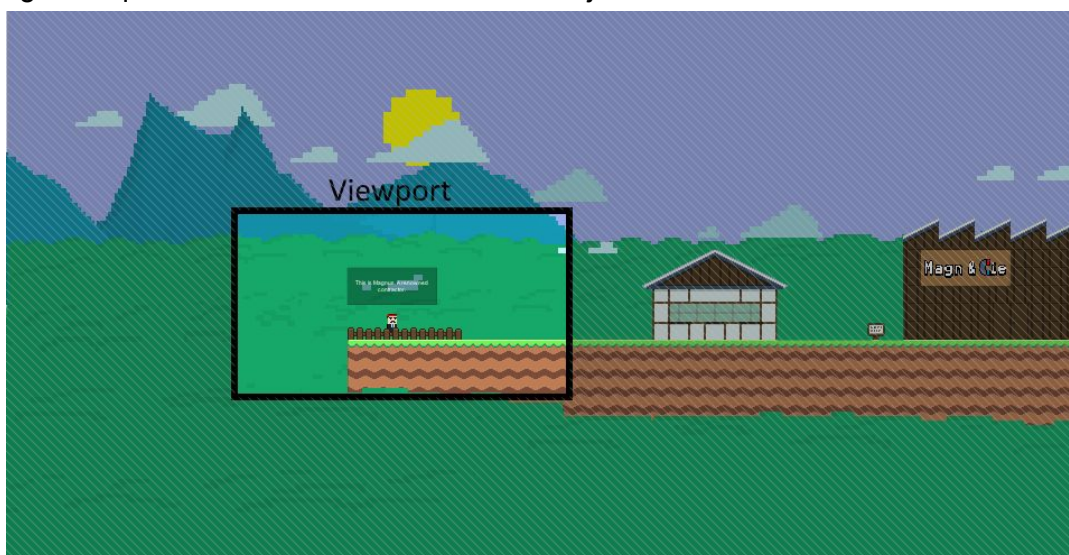
La répartition principale des tâches a largement été basé sur les expériences personnelles de chacun des membres de l'équipe. Pour pouvoir travailler en parallèle de manière optimale, on aura utilisé l'outil de *versionning* git, et avons opté pour le *workflow* qui porte son nom. Pour nos *merge request*, nous nous sommes mit d'accord que chaque membre du groupe devait faire une revue du code rapide puis éventuellement l'approuver.

## Implémentation

### Caméra

La caméra est une classe s'occupant principalement de changer le *viewport* de l'application afin de changer la zone affichée à l'écran.

Par exemple, bien que la carte soit beaucoup plus grande que ce qui est affiché, seul le rectangle indiqué en noir est visible sur l'écran du joueur.



*Représentation du fonctionnement de la caméra*

La caméra possède de nombreuses fonctionnalités:

- Centrer la vue sur une entité (si l'entité se déplace, la caméra suivra)
- Centrer la caméra sur une position quelconque
- Changement du niveau de zoom de la caméra
- Effet de tremblement de terre (non utilisé)
- Plusieurs mode de déplacement
  - Linéaire (avance en direction de l'entité de façon linéaire)
  - Ressort (avance en direction de l'entité de façon plus linéaire)
- Changement de la vitesse de déplacement

Différence entre les deux modes de déplacement de la caméra (avec un effet de ressort un peu exagéré pour la démo) : <https://youtu.be/3Lf2KmKMbZ8>

## Créations des niveaux

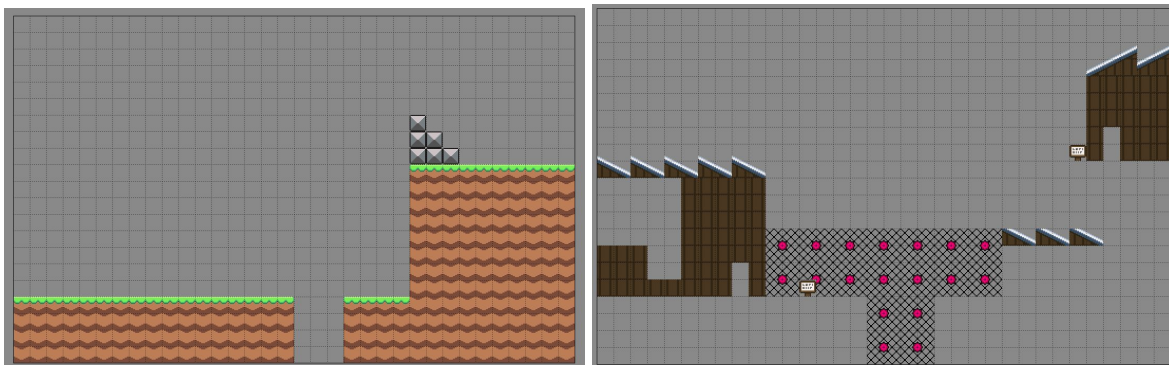
Cette partie de l'implémentation se base sur la bibliothèque de lecture et d'affichage du programme Tiled, développé en utilisant le framework Qt. Grâce à celle-ci, la lecture des fichiers XML détaillant les maps ainsi que l'affichage et les informations des différentes tuiles contenues dans les différents calques sont facilement accessibles.

Les classes concernant les niveaux font parties des seules qui héritent de l'objet *Entity*. La raison principale de ce choix dépend principalement du fonctionnement de la bibliothèque utilisée. En effet, celle-ci n'étant pas basé sur l'ECS, il aurait été compliqué et extrêmement chronophage de l'adapter à ce design d'architecture. La seconde raison étant que ces classes sont assez différentes d'une entité de base de par leur rôle et les composants qui aurait dû être développé pour celle-ci n'aurait pû être utilisés que des cas précis et presque inutile pour d'autres entités.

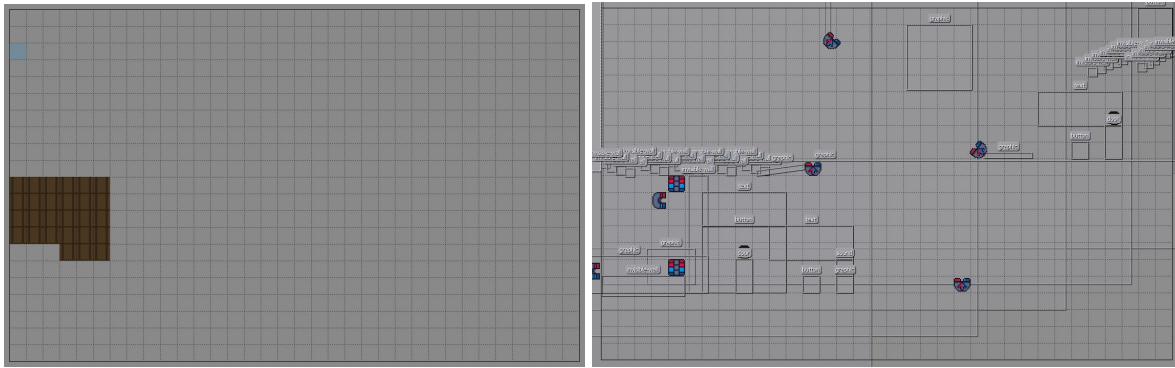
### Découpe d'un fichier de niveau

Un fichier Tiled est décomposé en plusieurs calques, dans le cadre de ce projet l'architecture de tout fichier de niveau est fixe et contient chaque fois 4 calques différents.

- Le calque de collision qui contient les tuiles à travers lesquelles une entités ne peut passer si elle est sensible aux collisions.
- Les calques de premier plan et d'arrière plan sont des calques contenant des tuiles décoratives.
- Le calque d'objet contient toutes les entités d'un niveau.



*Calques de collisions (gauche) et d'arrière plan (droite)*



*Calques de premier plan(gauche) et d'objets (droite)*

Dans le cadre de cette implémentations 4 classes ont été réalisées :

### MapEntity

Cet objet se construit grâce à l'objet permettant la lecture des fichiers Tiled ainsi que l'objet s'occupant du rendu de ceux-ci. Il agit comme un tableau contenant les différents calques de tuiles (LayerEntity) et d'objets (ObjectLayerEntity) d'un niveau.

### LayerEntity

Un LayerEntity est l'objet qui s'occupe de stocker la partie visuelle d'un niveau. Il consiste en une composition de tuiles qui sont ensuite rendues à l'écran grâce à la bibliothèque de Tiled. Les calques contiennent aussi les collisions générales d'un niveau et par conséquent il est possible pour n'importe quel LayerEntity de générer un masque de collision à partir de son contenu.

### ObjectLayerEntity

Cette partie représente les calques d'objets, les calques d'objets sont une spécificité de Tiled. En effet, ces calques contiennent généralement les différents éléments dynamiques d'un niveau. Dans notre cas, chaque niveau créé contient l'un de ces calques sur lequel sont stockées les informations qui permettent à l'EntityFactory de générer les différents entités avec leurs composants respectifs dont un niveau à besoin. Dans le cadre de notre implémentation, c'est cet objet qui va appeler les différentes méthodes de la fabrique d'entité en fonction du nom des différents objets où de l'id de certaines tuiles utilisées comme objets.

### EntityFactory

Cette classe ne contient que des méthodes statiques et permet de créer des modèles d'alliances "entité composants" pour faciliter la création de ses objets. Les objets les plus complexes à réaliser sont ensuite transformés en template Tiled, permettant une édition rapide de la carte dans l'éditeur.

## Particules

Le système de particule est composé de deux éléments:

- La classe *Particle* qui représente et dessine une particule
- Le composant *ParticleSpawnerComponent* qui est attaché à une entité et crée les particules.

Lors de la conception des particules, il a été choisi d'utiliser le système d'entité afin de pouvoir attacher des composants aux particules (afin de les faire réagir aux champs magnétiques) mais cette fonctionnalité n'a pas été utilisée au finale car trop gourmande en ressource.

Lors de la création d'un spawner, il faut lui donner trois fonctions: la fonction déterminant quand créer une particule, la fonction appliquée à chaque particule pour les mettre à jour ainsi que la fonction de rendu. Le code de ces fonctions n'est pas dupliqué (passage par référence au particules...)

Le spawner va ensuite créer des particules et les ajouter à la scène. Cette étape a causé quelques problèmes car nous ne pouvons pas ajouter ou supprimer des objets à l'intérieur de la boucle mettant à jour à jour ceux-ci. Ce qui nous amené à créer nos propres fonctions: *addEntityLater()* et *deleteLater()*

Le système de particule est actuellement uniquement utilisé sur le joueur afin d'afficher si l'aimant est activé ou non.



*Particules*

## Liste des composants

Voir annexe: "Liste des composants"

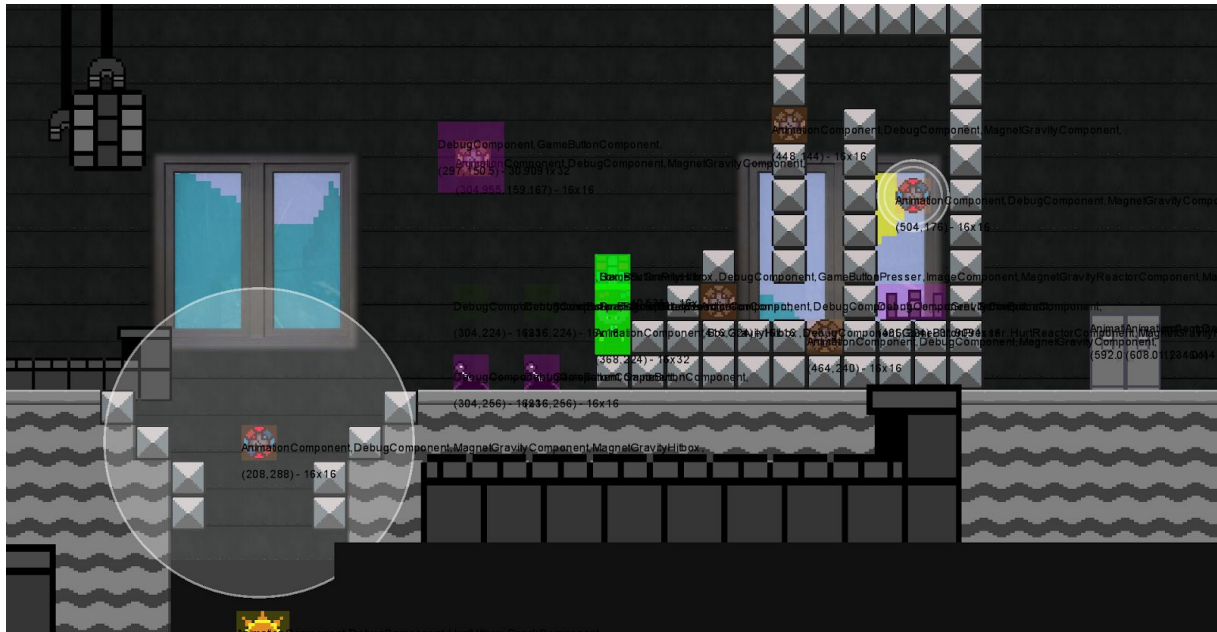
Certains composants plus complexes et/ou importants sont néanmoins expliqués ci-dessous:

### Débogage avancé

Afin de pouvoir déboguer le jeu de façon plus simple, nous avons trois composants dédiés à cette tâche: *DebugComponent*, *DebugTextComponent* et *DebugHitboxComponent*, permettant respectivement d'afficher la taille et position de l'entité, un texte personnalisé et les "hitbox" de chaque entité.



Voici à quoi ressemble le jeu lorsque ces différents composants sont ajoutés à certaines entités:



### Niveau avec éléments de débogages

## Gestion des inputs

Le composant *PlayerInput*, qu'on attache au personnage jouable, est le composant qui lie les entrées (qu'il lit depuis la classe *Input*) aux différents composants qui leur sont dépendants. Dans sa méthode *update*, il délègue les entrées à son attribut *state*. Ceci est une implémentation du *state design pattern* ([1], chapitre 7). Il permet de nous épargner des imbriquements de *ifs* interminables dûs au fait que le traitement des inputs se comporte comme une machine à états. Par exemple, la touche saut permet au personnage de sauter, mais seulement lorsqu'il est au sol. Ou encore, à l'inverse, il n'est nécessaire de vérifier que l'on a atterri que depuis un état "aérien".

De plus, chaque état a une méthode d'entrée, qui permet de modifier l'animation du joueur, jouer un son, démarrer un timer, etc. On utilise le timer pour aisément implémenter deux petites mécaniques pas nécessairement visibles mais très agréable pour le joueur. Ce sont le *jump leniency*, qui permet au joueur de sauter alors qu'il vient de tomber du bord d'une plateforme par accident, et les nuances de hauteur de saut, qui permet d'effectuer des petits bonds par l'appui bref du bouton saut, tout comme de grands saut par le maintien du même bouton.

Ces états sont instanciés statiquement pour éviter l'appel du constructeur et du destructeur à chaque changement d'état.

## Système de hitbox

Le composant Hitbox est virtuel pure, et possède l'interface nécessaire à déterminer si il intersecte une autre Hitbox, ainsi que des méthodes pour modifier ses dimensions ainsi que sa position relative à l'entité à laquelle elle appartient. Le rôle d'une Hitbox est déterminé par son attribut nom (qui correspond également au nom donné au composant), car il a été choisi qu'on puisse obtenir les instances de Hitbox d'après leurs noms dans une méthode statique.

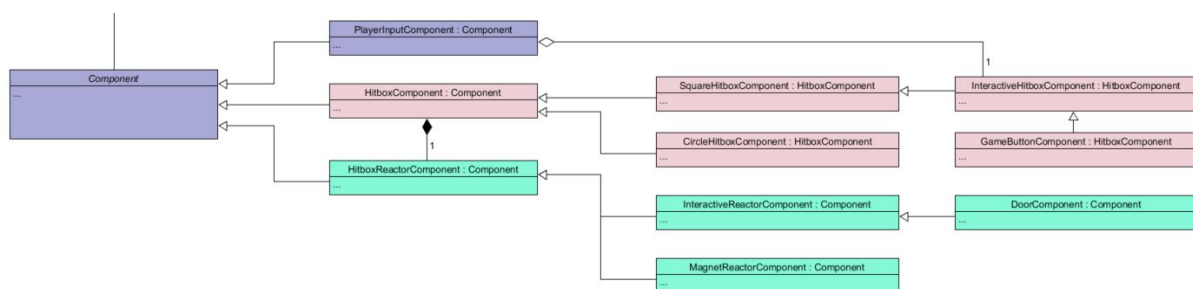
Les composants *SquareHitbox* et *CircleHitbox* sont deux implémentations de cette interface.

Le composant *HitboxReactor* est l'interface d'un composant permettant à son entité de réagir à la collision avec un type de Hitbox. Par exemple, le composant *MagnetZipperReactor*, attaché à un joueur, permet à celui-ci d'être captif du rayon d'un aimant *zipper*.

Plus particulières sont les hitboxes nécessaires à l'interaction, par exemple pour une porte ou un levier, qui nécessitent l'appui de la touche bas pour être actionnés. Pour faire ceci nous avons implémenté deux classes supplémentaires.

Le *InteractiveReactorComponent*, attaché aux entités avec lesquelles on peut interagir, hérite de *HitboxReactor*. Il est de plus attribué d'une touche que le joueur doit appuyer, et affiche une petite image indiquant de quel bouton il s'agit lorsque le joueur est à portée.




Le *InteractiveComponent*, uniquement attaché au joueur, permet à celui-ci d'interagir avec le composant ci-avant grâce à ses méthodes permettant de consulter les inputs.



UML représentant le système d'hitbox




## Fonctionnement général des aimants

Le jeu possède 3 types d'aimants: *Zipper* , *Gravity*  et *Jumper* , le comportement de chacun de ces aimants est expliqué plus loin.

Les aimants sont la mécanique principale de notre jeu, nous avons donc essayé de garder un système simple et flexible afin de pouvoir ajouter de nouveaux aimants de façon simple. Chaque aimant est composé de deux composants : un "générateur" et un "réacteur", pour créer un champ magnétique il suffit d'ajouter un générateur à une entité et pour qu'une autre entité y réagisse, il suffit de lui ajouter un "réacteur".

Le générateur et le réacteur possèdent tous deux une hitbox permettant de vérifier quand les deux entités s'intersectent.

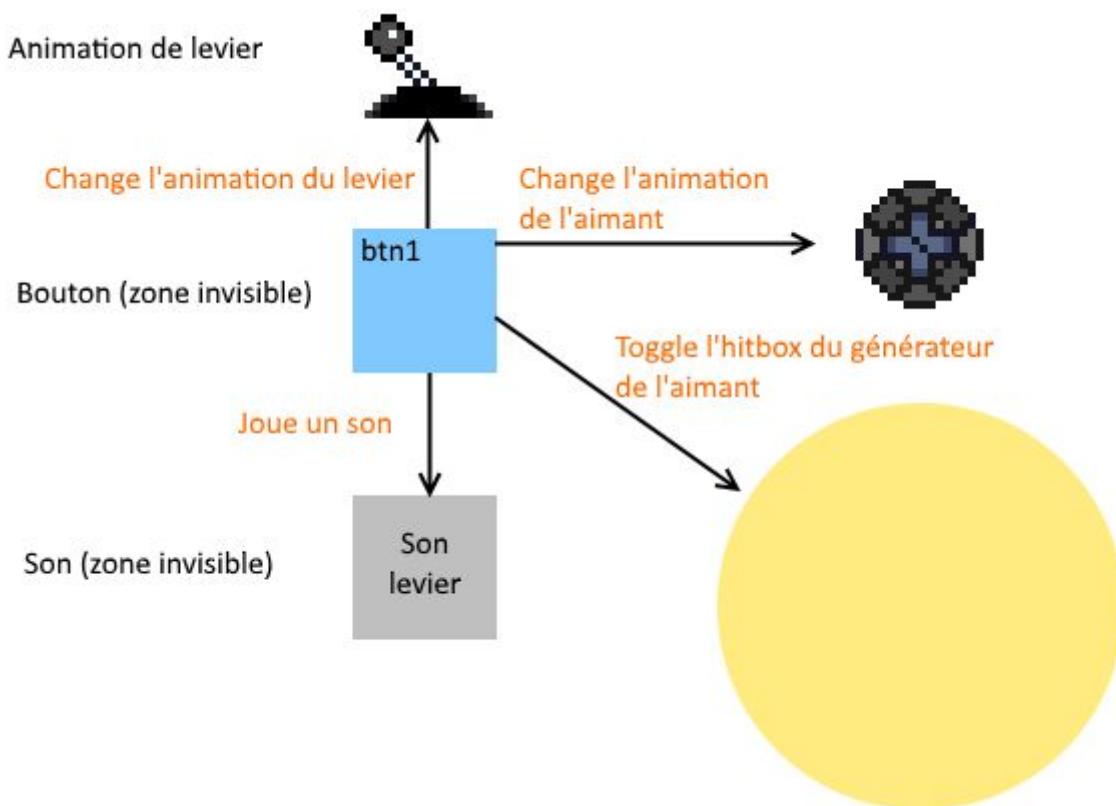
Un quatrième type d'aimant (*Grappin* ) était prévu, mais l'idée a été abandonnée car il ressemblait trop à l'aimant gravité.

## Système de boutons

Note : "bouton" ici signifie une zone présente sur une carte et non pas une touche du clavier.

Le système de bouton est essentiel à la réalisation de niveaux plus complexes, il permet à différentes entités d'interagir avec un bouton (levier, plaque de pression, réceptacle pour caisse, ...) afin d'affecter le comportement de nombreux composants.

Par exemple, les animations liées à un bouton changent d'état en fonction de l'état d'un bouton, le générateur de son joue un son quand un bouton est activé / désactivé et les hitbox des aimants sont activé / désactivée en fonction de l'état du bouton.



Les composants suivants peuvent être affecté par des boutons:

- *CameraSequence* (Démarre les séquences de caméra)
- *Door* (Empêche / autorise de rentrer dans une porte)
- *SoundButtonReactor* (Joue un son quand l'état du bouton change)
- *StoryMagnet* (Active la physique et sa hitbox)
- *Animation* (Change l'animation actuelle)
- *Image* (Cache / affiche l'image)
- *Text* (Cache / affiche le texte)
- *InteractiveReactor* (Empêche / autorise l'interaction)
- *MagnetGravity* (Active / désactive le champ magnétique)
- *MagnetJumper* (Active / désactive le champ magnétique)
- *MagnetZipper* (Active / désactive le champ magnétique)

Les boutons possèdent de nombreuses propriétés permettant de choisir les conditions nécessaires à leur activation / désactivation tel que:

- Inverser l'état de sorti du bouton
- La touche du clavier requise pour presser le bouton
- La durée durant laquelle le bouton restera pressé après interaction
- Si le bouton peut être activé / désactivé manuellement
- Les entités autorisées à toucher le bouton (permet d'autoriser seulement les caisses par exemple)
- Les boutons devant être pressé pour pouvoir presser ce bouton
- Les boutons devant être pressé pour pouvoir relâche ce bouton

N'importe quel classe du jeu peut utiliser la méthode statique

*GameButtonComponent::areButtonsPressed()* pour vérifier si tous les boutons de jeu passés en argument sont pressés.

Il est possible d'inverser l'état requis d'un bouton en ajoutant un "!" devant son nom.

## Parallaxe

Afin de donner une impression de profondeur dans nos niveaux, le *ParallaxComponent* change la position de son entité en fonction de celle de la caméra. Il change également son "z-index" afin de s'assurer que les éléments soient affichés dans le bon ordre.

Ce composant comporte possède un problème majeur : le parallaxe est affecté par la taille de l'écran impliquant que sur des écrans plus grand que 1920x1080, il y a un risque d'afficher des zones que le joueur n'est pas censé voir.

Attacher ce composant à une entité empêche également le changement de position par code.

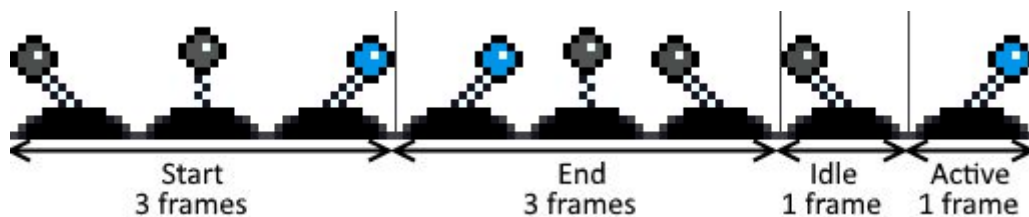
## Animations

L'*AnimationComponent* permet d'afficher des images animées de façon simple. Pour afficher une image statique, il faut plutôt utiliser *ImageComponent*, qui est plus approprié dans ce cas.

L'*AnimationComponent* utilise des *spritesheets* pour afficher différentes images les une après les autres, afin de donner l'impression que l'objet est animé. Plusieurs sous-animations peuvent coexister.

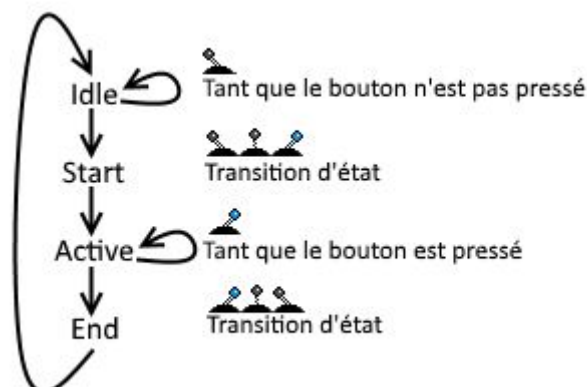
Pour créer une animation, il faut donner la largeur d'une image (commune à chaque sous-animation) puis spécifier le nom, le nombre de frame et la vitesse de chaque sous-animation.

Exemple avec le levier possédant quatre animations :



*Graphique du découpage d'un fichier d'animation*

Bien que les noms des sous-animations soient libres, "start", "end", "idle" et "active" sont des noms spéciaux car ce sont les noms utilisés pour changer l'animation en fonction de l'état des boutons selon la machine d'état suivante :



*Graphique d'état d'une animation*

Afin de faciliter la création d'animation, la classe *AnimationFactory* a été créée, elle contient une liste de définitions d'animation, et possède une méthode *GetAnimationComponent()* permettant d'obtenir une animation prête à l'emploi.

Les animations, tout comme les images, peuvent avoir différentes tailles et angles de rotation (sélectionnables sur Tiled dans les propriétés du rectangle "Graphics")

## Liste des entités

Les entités considérées comme importante sont décrites ci-dessous. Pour une liste plus exhaustive, voir annexe: "Liste des entités".

**Une entité est uniquement définie par ses composants.** Les seules propriétés sauvegardées dans une entité sont sa taille et sa position.

### Player

L'entité contrôlable par le joueur, possède une variante sans aimant utilisé dans l'histoire.



*Joueur*

### Composants (pendant l'introduction)

- *PlayerInputComponent*
- *InteractiveComponent*
- *AnimationComponent*
- *PhysicsComponent*
- *HurtReactorComponent*
- *HitboxComponent* (Pour activer les boutons poussoirs)

### Composants additionnels après l'introduction

- *MagnetGravityComponent* (Pour attraper les boîtes)
- *MagnetZipperReactorComponent*
- *MagnetJumperReactorComponent*
- *MagnetGravityReactorComponent*
- *HitboxComponent* (Pour éveiller le courroux des *Sparks*)
- *ParticleSpawnerComponent* (Pour montrer l'état de son aimant)

Le joueur est créé par un "*player spawner*" sur Tiled, les options du spawner permettent de définir l'animation et la taille utilisée pour le joueur ainsi que si le joueur possède des pouvoirs magnétiques ou non.

Si plusieurs *spawners* sont présents sur une map, plusieurs joueurs contrôlables vont apparaître, mais ce comportement n'est pas attendu lors du gameplay normal. Si aucun spawner n'est présent, le jeu plantera.

## Portes

Les portes permettent au joueur de changer de niveau et éventuellement de scène.



Portes

### Composants

- *DoorComponent*
- *AnimationComponent*
- *ParallaxComponent* (Permet de facilement mettre un objet légèrement en arrière plan)

Les portes ont leur comportement défini par leur *DoorComponent*, qui hérite de *InteractiveReactorComponent*. Il a juste fallu redéfinir la méthode *action*, c'est-à-dire celle qui est appelée lors de l'interaction à proprement parler.

Lors de l'interaction avec la porte, la porte s'ouvre, le joueur ne peut plus être contrôlé et joue une animation. La scène est notifiée qu'il faut changer de map (il n'est pas possible de changer de map depuis la porte car l'entité devrait se supprimer elle-même) à l'aide de la fonction *scheduleMapChange*

## Box

La caisse peut interagir avec les champs magnétiques, activer les boutons et est attirée par le joueur.



*Joueur portant une boîte*

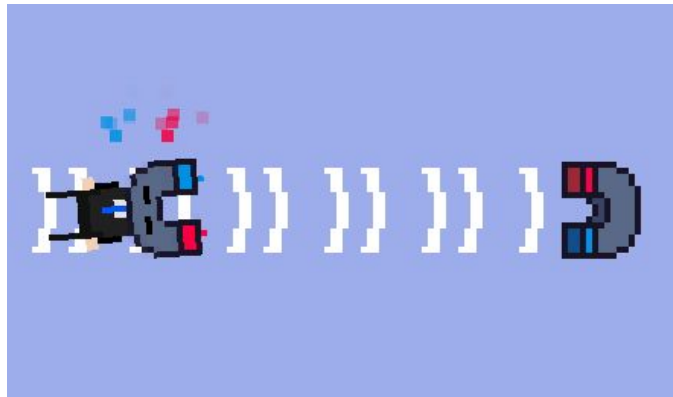
### Composants:

- *ImageComponent*
- *PhysicsComponent*
- Plusieurs *HitboxComponents* (Nécessaire pour ses collisions et interactions)
- *MagnetZipperReactorComponent*
- *MagnetJumperReactorComponent*
- Trois *MagnetGravityReactorComponents*:
  - Un pour être sensible aux aimants
  - Un pour être attrapé par le joueur
  - Un dernier pour repousser les autres caisses et éviter ainsi de s'empiler

La caisse est une des entités qui a pu être créée sans devoir coder de nouveau composants, démontrant ainsi l'utilité du *pattern* ECS. Elle ajoute de nombreuses possibilités de puzzle car elle peut interagir avec certains boutons.

## Magnet zipper

Attire le joueur et les caisses dans sa direction.



*Magnet zipper*

### Composants:

- *MagnetZipperComponent*

Cette entité ne possède qu'un composant de base, mais celui-ci s'occupe d'ajouter automatiquement les composants suivants:

- *SquareHitboxComponent* (Pour le champ magnétique)
- *AnimationComponent* (Pour l'animation du champ magnétique)
- *AnimationComponent* (Pour l'animation de l'aimant)

Cet aimant désactive la gravité et attire les entités de façon linéaire dans sa direction. Il y a comme un effet de "ressort" lors de l'entrée dans le champ afin de centrer l'entité au milieu du champ.

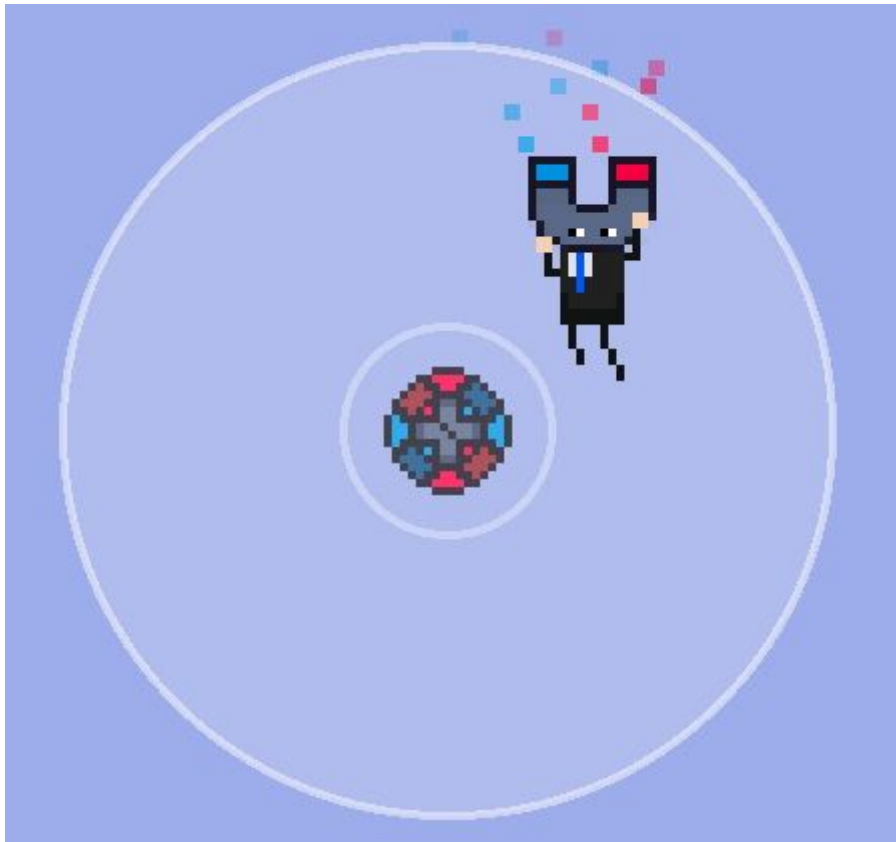
Il est possible de changer la taille, la direction et la force du champ magnétique.

Les entités gardent leur vitesse de sortie quand elles sortent de l'aimant (ne peut pas dépasser la vitesse maximale autorisée pour l'entité)



## Magnet gravity

Repousse ou attire le joueur et les caisses dans un certain rayon.



*Magnet gravity*

### Composants:

- *MagnetGravityComponent*
- *AnimationComponent*
- *GenericRenderComponent* (Pour l'animation dépendant de la force de l'aimant)

Le *MagnetGravityComponent* s'occupe de créer les *hitbox*.

Cet aimant attire ou repousse les entités (cela dépend de la force de base définie) plus ou moins fort en fonction de sa force et de la distance entre son centre et celui de l'entité.

Dans le cas où l'aimant est attractif, afin d'éviter des comportements imprévisibles (division par zéro...) l'entité est téléportée et immobilisée au centre de l'aimant quand elle est dans un rayon plus petit que cinq pixels, ce qui permet au joueur de "s'accrocher" au centre de cet aimant.

## Magnet jumper

Repousse le joueur et les caisses dans une certaine direction



*Magnet jumper*

### Composants:

- *MagnetJumperComponent*
- *ParticleSystemComponent*
- *AnimationComponent* (Pour montrer l'état d'activation et la direction du rebond)

Le *MagnetJumperComponent* s'occupe de créer sa *hitbox*.

Cet aimant donne, au contact avec toute entité y réagissant, une vitesse à celle-ci. La différence avec les aimants *gravity* répulsifs réside dans le fait que ceux-ci peuvent influencer à distance. Dans l'effet, le *jumper* est un trampoline.

On peut définir la taille, la direction et la force d'un tel aimant depuis Tiled.

## Problèmes rencontrés

Dans le cadre de l'implémentation, différents problèmes se sont présentés. Dans une idée de documentation, de partage d'information et afin de ne pas les reproduire dans de futur projet, ce chapitre les relatent ainsi que les solutions qui y furent apportées.

### Chargement des graphismes

Le premier problème bloquant rencontré dans le projet venait de la bibliothèque récupérée de Tiled. En effet la méthode qui permet le chargement des différents graphismes d'un niveau n'utilisait pas la bonne fonctionnalité de Qt quant à la résolution des chemins de fichiers.

```
QString absolutePath = QDir::cleanPath(dir.filePath(filePathOrUrl));  
if (absolutePath.startsWith(QLatin1String(":/")))  
    // return QUrl(QLatin1String("qrc") + absolutePath); //old  
    return QUrl::fromLocalFile(absolutePath);           //new  
  
return QUrl::fromLocalFile(absolutePath);
```

### Tailles des ressources

Lors de l'ajout des sons au projet, une erreur inattendue s'est produite. Le programme refusait de se compiler car les ressources incluses dans le système des ressources Qt, c.-à-d. les fichiers ajouté à l'exécutable en utilisant les *QRessources*, étaient trop volumineuses. Ceci a permis de révéler des lacunes quant à la compréhension et l'utilisation des ressources avec Qt. En effet, selon la documentation, les ressources sont censées être de petits éléments d'interfaces, icônes, petites images, petits fichiers, etc... Les ressources plus volumineuse se doivent d'être des éléments externes.

Pour palier au problème rapidement afin de continuer le reste du travail il fut décidé d'utiliser provisoirement la variable de configuration "resources\_big" destinée à qmake et permettant d'autoriser des ressources volumineuses.

Par la suite, une solution plus viable fut implémentée. Elle consiste à garder les sons comme ressources externes et en utilisant des adresses absolues pour les atteindre depuis l'intérieur du programme. Cette solution implique cependant de copier les ressources par nous même dans le futur dossier de l'application. Un script fut donc ajouté à la compilation afin d'effectuer cette copie. Cependant cette copie a révélé des différences de configuration entre les OS, et pour que la copie de Windows fonctionne de la même façon il aura fallu adapter un script par OS et enlever de façon explicite des configurations liées à la structure des dossiers créés lors de la compilation et ajouter tacitement lors d'une compilation sur Windows.

```
!win32 {  
    copydata.commands = $(COPY_DIR) $$PWD/assets $$OUT_PWD  
}  
win32 {  
    copydata.commands = $(COPY_DIR) $$shell_path($$PWD/assets) \  
    $$shell_path($$OUT_PWD/assets)  
}
```

## QSoundEffect

La décision d'apporter des effets sonores au programme a permis de soulever des différences notables entre les différents OS. Le premier choix pour l'implémentation des sons fut d'utiliser directement les fonctionnalités fournies par le framework Qt. Cependant, la classe *QSoundEffect*, censée jouer un son sans bloquer le thread principal et avec peu de délai, n'eut pas l'effet escompté dans la version du programme compilée pour Windows. En effet, à chaque fois que le joueur effectue une action, un ralentissement drastique du programme s'effectuait.

Beaucoup de recherche et de tests ont été effectués afin de résoudre ce problème, cependant peu d'informations étaient disponibles. De plus, il était compliqué de comprendre vraiment où se situaient le problème, entre l'OS, le framework et notre propre code.

La réponse aura été trouvée par essais et tâtonnements successifs. La solution consiste à créer une nouvelle classe héritant de *QThread* et d'instancier un thread par son joué. Ceci permet de réduire de façon importante les interruptions du programme. Cependant, cette solution n'est pas parfaite et dépend encore fortement de la RAM libre sur l'OS pour notre programme. Cette solution soulève aussi un problème sur MacOSX quant à la gestion des threads sur cette plateforme. En effet, sur Mac, les effets sonores ne peuvent être joués en utilisant cette technique. Une des pistes intéressantes à explorer serait de trouver un moyen afin de garder les différents effets sonores réellement déjà chargés dans la RAM au lancement du programme afin de les atteindre plus rapidement. Cependant, *QSoundEffect* ne permet pas d'avoir accès à son buffer et offre peu d'option quand à la gestion du chargement de ses ressources. À partir de ce constat, il semble intéressant d'utiliser une solution externe à Qt de gestion de fichiers médias dans le cadre d'un jeu vidéo.

## Protocole de tests

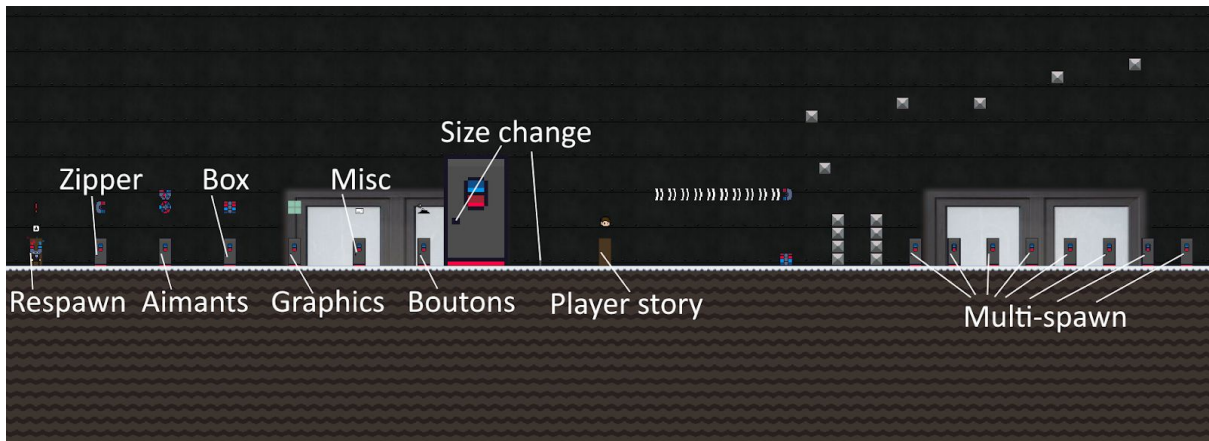
Par manque de temps, notre projet ne contient pas de test unitaires. Cependant nous possédons une procédure permettant de tester tous les éléments de gameplay ainsi que le bon déroulement du programme à haut niveau, en utilisant différentes maps de test. Les tests listés ci-dessous sont considéré comme "passés" sauf en cas d'indication contraire.

### Procédure de test

Les tests à effectuer sont expliqués par carte.

Commencer par charger le hub de test en appuyant sur **F8** pour ouvrir le dialogue de chargement d'une carte et saisir `:/maps/test-hub.tmx` afin d'ouvrir la carte principale de test.

### Test hub



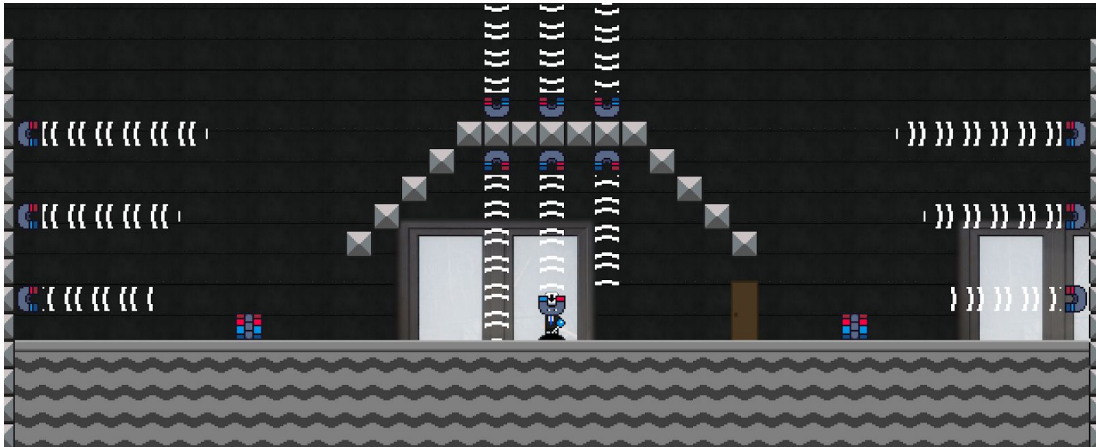
*Capture du hub de test*

Cette carte permet d'accéder aux autres cartes de test ainsi que de tester les *player spawn*. Avant d'entrer dans les portes permettant de test d'autres éléments, tester les éléments ci-dessous:

1. Entrer dans la porte "Respawn" et vérifier que la map est rechargée.
2. Vérifier que la musique est chargée et se joue.
3. Utiliser une des deux portes "Size change" et vérifier que la taille du joueur est changée.
4. Utiliser la porte "Player story" et vérifier que le joueur sans l'aimant soit chargé.
5. Le joueur sans l'aimant ne doit pas pouvoir activer / désactiver l'aimant et ne doit pas être affecté par les champs magnétiques.

Note: le "Multi-spawn" n'a pas d'utilisation pratique et peut crash le programme si un des joueurs essaie d'entrer dans une porte, il n'est donc pas conseillé d'interagir avec ces portes avant la fin des tests. De plus, le système de gestion des inputs n'étant pas créé pour ce type d'application, on se retrouve avec des comportements inattendus, comme un saut qui peut être prolongé indéfiniment. Ceci est dû aux PlayerStates, dont tous les différents PlayerInputComponents se partagent les instances statiques. Le multi-spawn était donc juste une expérimentation des limites de notre ECS, mais ne porte aucune importance en tant que test.

## Test zipper

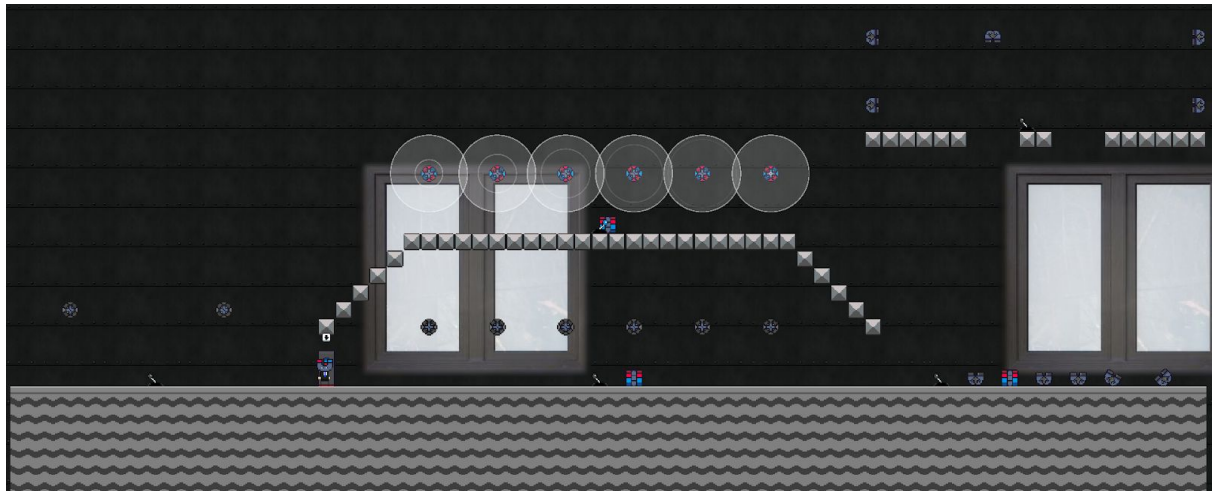


*Niveau de tests des zippers*

Cette carte permet de tester le bon fonctionnement des *aimants zipper*

1. Interagir avec le levier, vérifier l'activation / désactivation des aimants et de leur champs magnétiques.
2. Vérifier que les animations fonctionnent (désactivée, activée, activation et désactivation)
3. Les animations des champs magnétiques doivent posséder des vitesses différentes.
4. Le joueur doit être déplacé dans la direction des champs avec des vitesses plus ou moins grande, seulement si son aimant est activé.
5. Hormis pour les aimants possédant une vitesse maximale, le joueur ne doit pas passer à travers les murs

## Test aimants

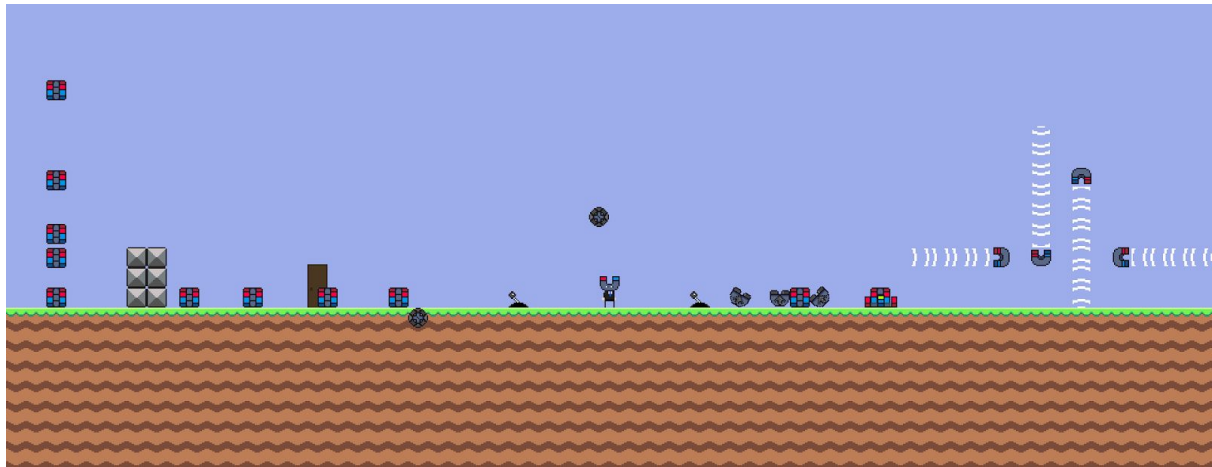


*Niveau de tests des aimants*

Cette carte permet de tester le bon fonctionnement des aimants *gravité* et *jumpeurs*

1. Interagir avec les leviers, vérifier l'activation / désactivation des aimants et de leur champs magnétiques.
2. Vérifier que les animations fonctionnent (désactivées, activées, activation et désactivation)
3. Des particules doivent apparaître au dessus des jumpeurs quand ils sont activés (avec une direction appropriée)
4. Les vitesse des animations des champs magnétiques des aimants gravité doit changer en fonction de la force du champ magnétique
5. Le joueur doit être attiré / repoussé plus ou moins fort selon la puissance du champ de l'aimant.
6. Le joueur doit se "coller" au centre des aimants gravités ayant une attraction en son centre.
7. Le joueur doit être propulsé dans la bonne direction lors d'un contact avec un aimant jumper.

## Test boîtes



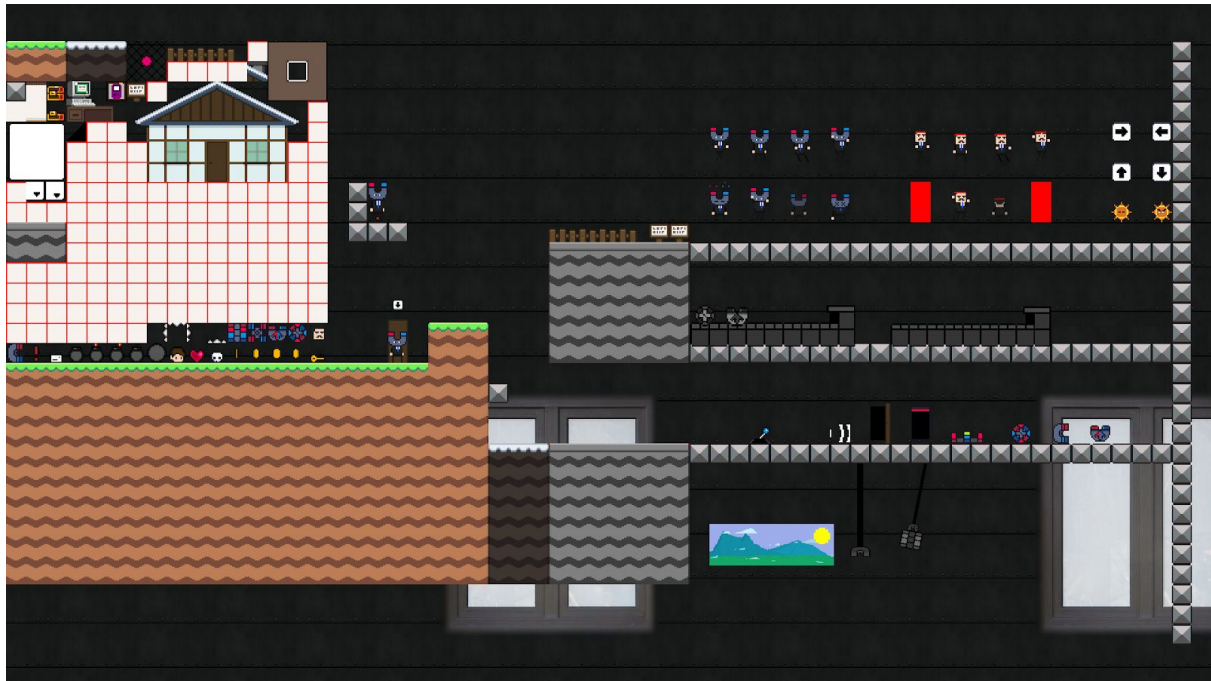
*Niveau de tests des boîtes*

Permet de tester les boîtes et leur interactions avec d'autres aimants

1. Les boîtes ne doivent pas pouvoir se superposer entre elle. (**Test échoué**)
2. Les boîtes doivent se repousser entre elle.
3. Les boîtes réagissent de la même façon que le joueur aux champs magnétiques.
4. Le joueur ne doit pas pouvoir activer le "réceptacle de caisse" et la caisse doit pouvoir l'activer.
5. Les boîtes sont attirées par le joueur uniquement quand son aimant est activé.
6. Les boîtes se "collent" au dessus de la tête du joueur quand elles y sont assez proche.



## Test graphiques

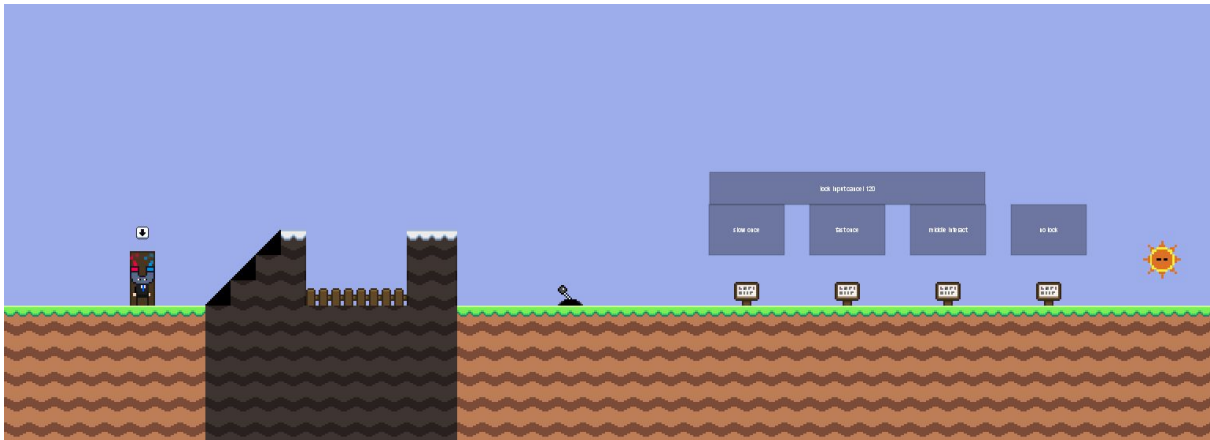


*Niveau de tests des graphismes*

Permet de tester l'affichage des images et le fonctionnement des animations.

1. Le tileset doit être entier et contenir les même tiles que celui dans les ressources.
2. Les animations affichées en haut à droite de la map doivent être animée proprement.  
(Les rectangles rouges sont des animations normales pour le player story)
3. Vérifier que les animations activées par des leviers fonctionnent (désactivées, activées, activation et désactivation)
4. Vérifier que le décor soit bien affiché (montagnes, nuages, aimants suspendus, ...)  
**(Test échoué : le culling ne fonctionne pas correctement pour l'aimant avec une rotation)**
5. Vérifier que le deuxième aimants suspendu possède une rotation.
6. Vérifier le parallax avec le bouton invisible placé sur le point d'exclamation au dessus du spawn.

## Test divers

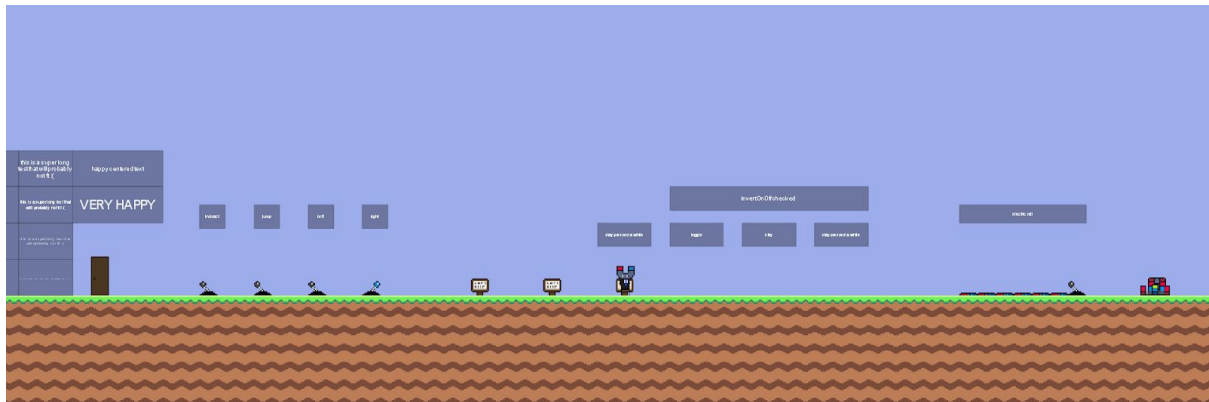


*Niveau de tests divers*

Test diverses fonctionnalités, certaines n'étant pas utilisé dans le reste du jeu.

1. Vérifier qu'on puisse grimper la pente (test des obstacles invisibles).
2. Un contact avec les barrières en bois doit tuer le joueur et recharger la carte.
3. Interagir avec le levier devrait jouer un son.
4. Vérifier que la caméra se déplace au dessus des pics en cas d'interaction avec les différentes zone de *camera sequence*
5. Vérifier que la spark se dirige vers le personnage et le tue lorsqu'il est suffisamment proche

## Test boutons



*Niveau de tests des boutons*

Permet de tester le fonctionnement des boutons de jeu et l'affichage de texte

1. Les textes affichés doivent avoir différentes tailles et ne doivent pas sortir de leur cadre.
2. Vérifier que les inputs affichés au dessus des boutons changent l'état des leviers.
3. Vérifier que les différents textes suivants s'affichent en fonction de l'état correspondant des leviers : (!interact !jump !left !right, interact jump !left !right et interact jump left right)
4. Vérifier les modes d'activation des boutons (toggle, stay, stay a while)
5. Vérifier les mêmes, avec inversion on/off
6. Vérifier le rail électrique (boutons presseurs suivis du levier, activable uniquement en séquence de gauche à droite)
7. Vérifier le fonctionnement du bouton activable par boîte

## Test du jeu

En plus du test individuel des fonctionnalités ci-dessus, il est important de vérifier les points suivants:

- Passage du menu au jeu
- Tester que toutes les cartes soient possibles
- Tester que les portes amènent bien au niveau suivant / précédent, sur le spawn correct.
- Passage du jeu aux crédits (en terminant le dernier niveau)
- Passage des crédits au menu

## "Garde-fou" implémenté dans le code

En plus des tests effectués ci-dessus, le jeu peut détecter certaines erreurs, notamment liées à la conception des cartes:

- Affichage d'une erreur dans la console lors de l'utilisation d'un rectangle / tile non supporté (Unknown object "<object name>")

- Assert lors de l'utilisation d'une animation inexistante
- Assert lors de l'utilisation d'une hitbox nommée avec un nom d'aimant, sans que l'entité possède le composant générant le champ magnétique.

## Bugs connus

Les bugs répertoriés ci-dessous ne sont pour le moment pas associés à des issues sur Gitlab car nous souhaitons déjà terminé une première version du projet, sans être tentés de corriger toutes les erreurs.

Bug	Criticité
Le culling ne marche pas correctement pour les animations et images possédant une rotation	Moyenne
Le jeu ne s'affiche pas correctement quand il est lancé sur un écran avec une résolution supérieure à 1920x1080	Haute
Le menu de pause ne s'affiche pas toujours à la même taille	Moyenne
Le jeu "lag" lors du chargement de nouvelle musique (au début de niveau)	Moyenne
La touche pour rendre le son muet sur Linux ne fonctionne pas comme attendu après sa première activation / désactivation	Moyenne
Très très rare occurrence d'un crash dû au thread gérant l'audio.	Basse
Déclencher plusieurs séquences caméras en même temps bloque la caméra.	Très basse
Mourir / charger un nouveau niveau pendant une séquence caméra bloque la caméra avec une vitesse incorrecte.	Très basse
Entrer dans plusieurs portes en même temps avec plusieurs joueurs peut crash le jeu	Très basse
La vitesse de saut du joueur est cumulée avec celle des jumpers, permettant d'effectuer des super sauts.	Moyenne
Sauter contre un plafond avec une caisse téléporte la caisse à gauche du plafond.	Moyenne
Une des cartes possède un aimant avec la mauvaise animation. (Causé par soucis de gameplay)	Basse
Le jeu s'appelle "Glowing barn" au lieu de "Magn & Cie" dans la barre des tâches	Basse
Les effets sonores ne se jouent pas sur MacOS	Haute

L'utilisation d'outils tel que le memcheck de Valgrind a révélé que notre application possède un memory leak:

```

==13638== HEAP SUMMARY:
==13638==       in use at exit: 5,875,051 bytes in 72,029 blocks
==13638==   total heap usage: 2,038,317 allocs, 1,966,288 frees, 13,448,608,415 bytes allocated
==13638== LEAK SUMMARY:
==13638==    definitely lost: 60,137 bytes in 311 blocks
==13638==    indirectly lost: 270,925 bytes in 1,620 blocks
==13638==    possibly lost: 12,290 bytes in 192 blocks
==13638==    still reachable: 5,341,131 bytes in 68,788 blocks
==13638==               of which reachable via heuristic:
==13638==                 length64      : 12,120 bytes in 270 blocks
==13638==                 newarray      : 2,208 bytes in 58 blocks
==13638==                 multipleinheritance: 43,632 bytes in 54 blocks
==13638==    suppressed: 0 bytes in 0 blocks
  
```

*Sortie de Valgrind sur notre programme*

Bien que nous n'ayons pour le moment pas le temps nécessaire d'investiguer davantage cette erreur, ainsi que d'apprendre à utiliser correctement les outils mis à disposition (profiler, memcheck, ...), nous avons retenu plusieurs leçons essentielles pour nos prochains projets en C++

- Absolument utiliser les smart pointers
- Passer les arguments par références

De plus, memcheck a trouvé des problèmes causé par des valeurs que nous avons oublié d'initialiser. Ces erreurs seraient corrigées très rapidement si nous n'avions pas décidé de ne pas refaire une nouvelle release avant le rendu de ce rapport.

Memcheck	
Issue	Location
▶ Conditional jump or move depends on uninitialised value(s)	<a href="#">playerinputcomponent.cpp:116</a>
▶ Conditional jump or move depends on uninitialised value(s)	<a href="#">input.cpp:37</a>
▶ Use of uninitialised value of size 8	<a href="#">input.cpp:37</a>
▶ Use of uninitialised value of size 8	<a href="#">input.cpp:37</a>
▶ Use of uninitialised value of size 8	<a href="#">input.cpp:37</a>
▶ Use of uninitialised value of size 8	<a href="#">input.cpp:37</a>
▶ Conditional jump or move depends on uninitialised value(s)	<a href="#">input.cpp:37</a>
▶ Conditional jump or move depends on uninitialised value(s)	<a href="#">input.cpp:37</a>
▶ Use of uninitialised value of size 8	<a href="#">input.cpp:37</a>
▶ Use of uninitialised value of size 8	<a href="#">input.cpp:37</a>
▶ Conditional jump or move depends on uninitialised value(s)	<a href="#">playerinputcomponent.cpp:116</a>
▶ Use of uninitialised value of size 8	<a href="#">input.cpp:37</a>

*Sortie de memcheck sur notre programme*

## Déploiement

### Windows / MacOS

Dans le cadre de MacOS et de Windows les programmes de rassemblement des dépendances fournis par Qt ont été utilisés. Cependant le programme de génération d'installateur n'aura été utilisé que pour Windows. Dans le cadre de MacOS on lui préférera un simple .dmg comprenant le programmes et un lien vers le dossier des applications car les utilisateurs de cet OS sont habitués à ce système ci.

### Linux

Le déploiement sur Linux aura été plus compliqué, car l'outil rassemblant les librairies nécessaires au projet nécessitait une (très) vieille version de libc et cette librairie est utilisée par la majorité du système sur Linux, changer cette librairie est un bon moyen d'assurer la mort du système. Par manque de temps, les librairies ont été rassemblées à la main (à l'aide de la commande *ldd* ainsi que des comparaisons avec ce qui était généré sur Windows). L'installateur n'aura pas été proposé non plus car il est plus usuel sur linux d'avoir simplement un programme que l'on peut lancer. Le programme se lance à l'aide d'un script shell afin de définir le bon chemin de recherche pour les librairies partagées.

## Changement à effectuer pour de futur projets

Bien que nous soyons globalement content du résultat, il y a beaucoup d'éléments de code que nous aimerions changer dans notre prochain projet :

- Utilisation du "vrai" ECS
- Utilisation de *smart pointer*
- Plus grand utilisation du passage par référence
- Apporter une attention plus particulière à ne pas oublier les mots clés *override*, *const*
- Documenter le code au fur et à mesure
- Système de Hitbox séparé du système de composant
- Affiner l'estimation de la durée de chaque tâche en jour et non en nombre de semaine



## Critique du planning

Bien que les objectifs du cahier des charges aient été remplis (ou justifiablement abandonnés), on peut néanmoins se pencher sur le respect tout relatif du planning au fil du développement. En effet, grisés que nous étions par l'avancement, nous ne pouvions nous empêcher de travailler sur quelque autre fonctionnalité moins pressante. On peut voir par exemple que l'*issue* "menu" devait être entamée le 9 décembre, mais ça n'a pas été le cas avant plus d'un mois plus tard. Il faut toutefois noter que ceci est vraisemblablement aussi en partie dû à un mauvais agencement temporel de nos tâches. Le planning intercalant des mécaniques de haut niveau par des fonctionnalités de base, et ces aller-retours nous poussaient à les éviter.

Hormis cela, on peut globalement se féliciter d'avoir su estimer les durées de chaque tâche (utiliser comme unité la semaine y a contribué...) Il est dit "globalement" car il y a des exceptions, comme en particulier "sons et bruitages", qui, comme l'explique la section qui lui est dédiée, aura pris considérablement plus d'une semaine avant d'être considérée comme satisfaisante pour le rendu final.

Une autre tâche dont la durée s'est avérée différer grandement de son estimation est celle concernant l'ECS. En effet, comme il s'agissait d'une des premières tâches, il y avait très peu à faire, puisque le gros du travail pour ce système résidait dans la réalisation de composants spécifiques à d'autres tâches.

## Conclusion

Si nous considérons ce programme un succès, malgré les fonctionnalités qu'on a choisi de ne pas garder, et le suivi tout relatif du planning, c'est parce qu'il nous aura donné la satisfaction de voir marcher mieux qu'escompté notre jeu. Toutefois, en prenant le recul nécessaire à l'autocritique, on ne peut que regretter le manque de clarté dans notre vision partagée au début du projet. C'est celui-ci qui nous aura fait nous fixer ces objectifs que l'on aura par la suite abandonnés. Ceci nous motive à l'avenir, pour ce genre de projet, à prototyper très tôt pour mieux saisir, tous ensemble, l'objectif final. Puisque la jeune discipline du *game design* manque encore de vocabulaire pour partager ces notions, rien ne vaut encore l'exemple.

## Annexes

1. Cahier-des-charges.pdf
2. Spécifications.pdf
3. Liste-des-entites-disponibles-sur-tiled.pdf
4. Liste-des-composants-disponibles.pdf

## Bibliographie

[1] Game Programming Patterns, Robert Nystrom,  
<http://gameprogrammingpatterns.com/contents.html>