Rapport de stage
Programmeur Gameplay

MOULARD Valentin
Du 29 Juillet 2019 au 29 Novembre 2019

Tuteur de stage : LEPRINCE Cyril
Enseignant Référent : PUZENAT Didier

Établissement de formation : Université Lumière Lyon 2 – M2 Informatique Gamagora
Entreprise d’accueil : JLA Games – 7 Rue des Bretons, 93210 Saint-Denis

Table des matières

Tous les termes en **gras**, lors de la lecture, sont définis dans le glossaire.

Introduction

Dans le cadre de mon année de Master 2 Informatique spécialisé en jeux vidéo à Gamagora, j'ai dû réaliser un stage de quatre mois. Dans ce rapport, je vais exposer le contexte dans lequel j'ai réalisé mon stage en présentant l'entreprise, les projets sur lesquels j'ai travaillé et les problématiques majeures que j'ai rencontré. Nous allons ensuite développer ces différentes problématiques, présenter les réflexions et les solutions apportées. Enfin, je donnerai une conclusion sur mon stage dans sa globalité.

1 - Contexte

A - Le jeu hyper casual

Les Hyper-Casual Games sont des jeux vidéo dérivés des Casual Games développés pour un large public sur mobile.

Ces jeux vidéo s'adressent à tous les joueurs occasionnels qui recherchent des sessions de gameplay hyper courtes : entre quelques secondes et une minute de jeu.

Les codes des jeux hyper casual :

- Téléchargement gratuit ;
- Gameplay simpliste et très court ;
- Mécaniques accessibles ;
- Contrôles intuitifs ;
- Procurent un plaisir immédiat ;
- Graphismes et sons minimalistes ;
- Modèle économique basé sur la publicité et les achats en jeu (InAppPurchase) ;
- Rejouabilité élevé.

B - Historique du jeu hyper casual

Il y a quelques années, nous identifions trois genres de jeux différents qui ciblaient trois publics différentes : Le public Casual jouant moins de deux heures par semaine, le public Mid-Core jouant moins d'une heure par jour et le public Hardcore jouant environ deux heures par jour.

Candy Crush Saga est un exemple parfait de Casual Game : des sessions de jeu très brèves de 2 à 5 minutes avec des règles rapides à assimiler, une faible courbe de difficulté et des graphismes travaillés. Depuis environ 2016, un quatrième genre est venu s'ajouter à la liste en se plaçant avant les Casual Games, les Hyper-Casual Games.

Au alentour de 2005, on pouvait déjà considérer que ce genre de jeux existaient avec les jeux web/jeux Flash. On peut considérer que certains de ces jeux avaient déjà les codes des Hyper-Casual Games : des parties très courtes, des graphismes minimalistes, des mécaniques qui offrent un plaisir immédiat avec des contrôles simplifiés et jouable gratuitement avec des publicités.

Avec l'arrivée de l'iPhone et des jeux mobiles pour smartphone en 2008, nous retrouvons ce genre de jeux. Mais le modèle économique est différent de celui que nous connaissons aujourd'hui. La monétisation du jeu n'est pas faite par les publicités que le joueur voit mais par un achat au téléchargement dit 'achat Premium'. Ces jeux sont vendus à un prix très bas, souvent autour de 1$.

Depuis quelques années, Voodoo, un autre éditeur français de jeux Hyper-Casual, arrive à placer systématiquement leurs jeux dans les tops de Google Play et de l'AppStore.
Ces éditeurs achètent des utilisateurs via des publicités pour générer un maximum de téléchargements et rester dans le haut des classements. Le tout va générer des revenus grâce aux nombreuses publicités diffusées à l'intérieur du jeu. Ensuite une partie de ces bénéfices est distribuée aux développeurs des applications.

Aujourd'hui, pour pouvoir sortir un jeu hyper casual, il est indispensable d'avoir un éditeur pour faire la promotion du jeu. Si l'application n'est jamais téléchargée, elle ne peut pas générer de revenus puisque la monétisation est basée sur le visionnage de publicités et d'achats en jeu.

C - JLA Groupe

Créé le 1er juillet 1999 par Jean-Luc Azoulay, JLA GROUPE est né de la fusion des activités de productions des sociétés AB et Hamster Productions et développe depuis lors des programmes destinés aux chaînes historiques, mais aussi aux chaînes de la TNT.

Au travers de ses différents labels (JLA Productions, Exilène Films, VAB, Carma Films) le groupe JLA propose une production large et variée de fictions et de flux.

JLA GROUPE est une société de production indépendante regroupant des filiales dédiées à la fiction, au flux, au jeu vidéo, à la jeunesse, à la musique, et qui détient également une chaîne de télévision.
Depuis sa création, JLA GROUPE a livré plus de 1500 heures de programmes à l'ensemble des chaînes françaises dont certains des derniers plus gros succès d'audiences avec *Une chance de trop* (VAB), *Camping Paradis* (JLA Productions), *La Vengeance aux yeux clairs* (JLA Productions), *Munch* (JLA Productions / Exilène Films) sur TF1, *La Forêt* (Carma Films), *Commissaire Magellan* (JLA Productions) ainsi que *Le prix de la vérité* (Episode Productions) sur France 3.

D - JLA Games

Crée en octobre 2014 par Jean Luc Azoulay, et pris en charge par Guillaume Nasi, JLA GAMES répond à une volonté de diversification des activités du groupe JLA. JLA GAMES s'occupe de la création de jeux vidéo et applications en relation avec les productions du groupe JLA, comme :

- Les Mystères de l'Amour, Le Jeu ;
- Camping Paradis, Le Jeu ;
- L'Application Karaoké Elsa Esnoult.

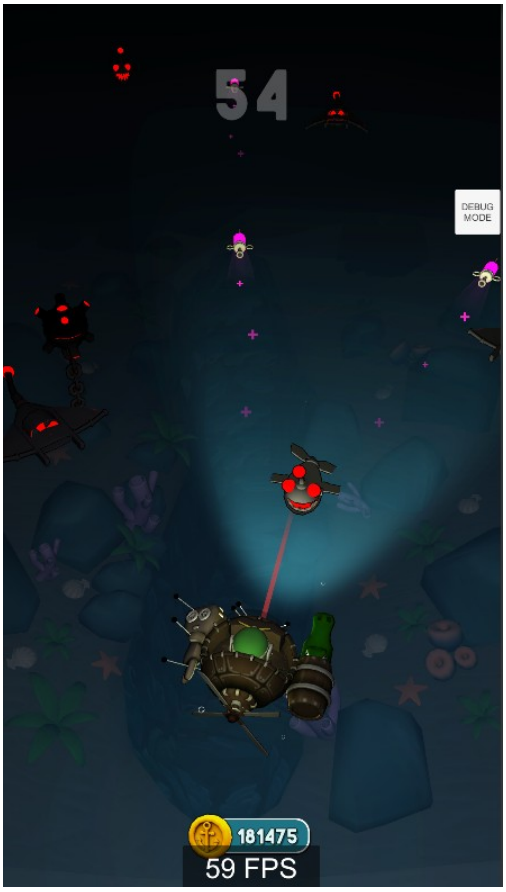
E - Mission au sein de JLA Games

La production de jeux hyper casual étant très tendance ces dernières années, JLA games s'est tourné vers ce secteur. Dans le cadre du stage, j'ai participé au développement de trois jeux hyper casual.

JLA Games est nouveau dans la production de jeux hyper casual. D'une part, l'équipe de développement a une volonté de créer des jeux hyper casual qui sont plus attrayants que ceux sur le marché dans l'optique de se démarquer par des visuels plus peaufinés. Cela implique plus de temps de production au niveau graphique. Nous voulons donner une identité visuelle propre à chaque jeux. D'autre part, les projets de jeux hyper casual menés par JLA Games doivent être facilement maintenables et reprenables au niveau technique. Cela implique un temps de réflexion plus long pour concevoir les éléments de gameplay et architecturer le code. Étant donné que les jeux hyper casual présentent souvent les mêmes briques de gameplay (Menu, Contrôles, interfaces utilisateur, etc.), la production de fonctionnalités génériques représente un gain de temps sur les futurs projets hyper casual. Les temps de production sont plus long que ce qui se fait dans le domaine de l'hyper casual car l'équipe veut produire des jeux qui se distingue des autres par une meilleure qualité de produit.

Voici la présentation des trois jeux présentant leurs contrôles, gameplay en général et fonctionnalités qui incitent le joueur à revenir sur le jeu :

- Aquabyss ;
- Slashing Machine ;
- Smashine.



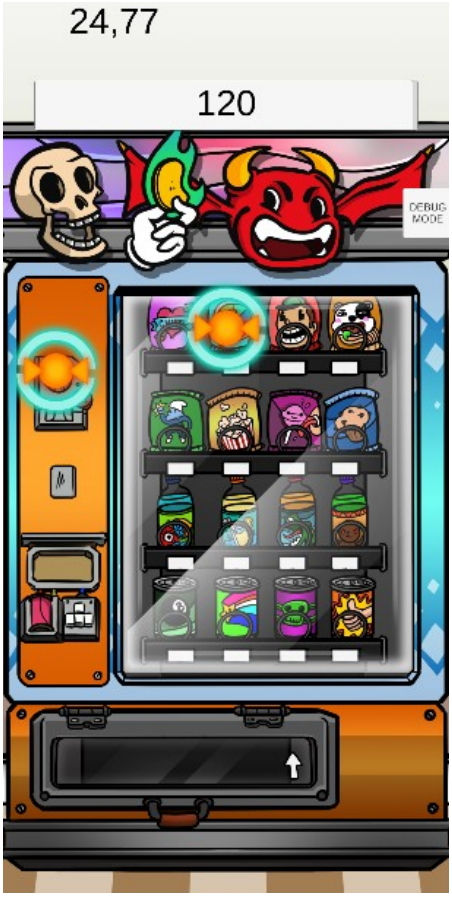
Aquabyss est un jeu où l'on contrôle un sous-marin, tirant sur des ennemis mécaniques, pouvant être assimilé au jeu d'arcade Space Invaders. Le joueur contrôle son sous-marin et tente de survivre le plus longtemps possible aux vagues d'ennemis en les détruisant avec le canon du vaisseau.
Le jeu est en mode **endless,** plus le joueur tient longtemps dans le niveau, plus la difficulté augmente. La difficulté est gérée en manipulant la vitesse d'apparition des ennemis en fonction du temps.

Le joueur peut uniquement pivoter son sous-marin à l'aide d'un **joystick virtuel**. Lorsque le joueur ne touche pas le joystick, le sous-marin tire automatiquement avec une vitesse dépendant de l'arme équipée par le joueur.
Il existe trois types d'ennemis, les raies, les piranhas et les mines. Les raies ont une vitesse normale et n'ont qu'un seul point de vie. Les piranhas ont un point de vie et sont plus rapides et les mines ont une vitesse normale mais disposent de deux points de vie.

Lorsque le joueur détruit un ennemi, il gagne un nombre de pièces en fonction du type d'ennemi abattu. Ces pièces permettent d'acheter des nouvelles armes ayant chacune un effet différent et des apparences de sous-marin. Le joueur peut choisir l'arme à équiper et l'apparence du sous marin avant le début d'une partie au menu principal.
La sensation de progression dans le jeu est amené par le fait d'acheter des nouvelles armes qui sont de plus en plus puissantes et amusantes. Plus le joueur a une arme puissante, plus il survivra longtemps dans le niveau, plus il générera de pièces et plus il aura les moyens de débloquer les armes suivantes. La rétention des joueurs se fait sur la progression par l'achat de nouvelles armes et sur la personnalisation du sous-marin.

Slashing Machine est semblable à *Aquabyss*. On contrôle un personnage dans un niveau qui défile à l'infini. L'objectif est de marquer le plus de points possible en éliminant des ennemis qui arrivent sur le joueur et en allant le plus loin possible. Les ennemis arrivent toujours plus rapidement en plus grand nombre, au fur et à mesure que le temps passe.
Le joueur peut taper sur l'écran pour effectuer une attaque. Si l'ennemi se trouve à portée de l'attaque, l'ennemi est éliminé. Cette portée est représentée par une zone de frappe de taille variable, posée devant le personnage. Si un ennemi est dans cette zone et que le joueur déclenche une frappe, l'ennemi est détruit. Si le joueur détruit un ennemi, il peut taper à nouveau, sinon le joueur entre en phase de récupération pour un court délai.
Il existe trois types de mercenaires, ceux en skateboards, ceux en scooters et ceux en voitures. Les ennemis ont des vitesses variables et augmentent au cours du temps. Ils avancent sur trois voies parallèles, une pour chaque type d'ennemi et se dirigent vers le joueur. Si un ennemi touche le personnage, le joueur perd la partie.
Le joueur est soumis à deux statistiques qui vont influencer son gameplay. Le temps de récupération (le délai après lequel le joueur peut frapper à nouveau s'il manque son coup) et la portée (la taille effective de la zone de frappe prise en compte pour éliminer un ennemi).
Le joueur peut acheter des vies supplémentaires pour une certaine somme de monnaie réelle. Une vie supplémentaire permet d'encaisser un coup en plus lors d'une partie.

Le joueur gagne de la monnaie virtuelle en fonction de ses performances dans une partie. Il peut débloquer des nouvelles armes qui sont des objets cosmétiques et améliorer ses statistiques. Le joueur peut soit réduire son temps de récupération ou augmenter sa portée. Dans la même logique que *Aquabyss*, plus le joueur améliore ses statistiques, plus il pourra aller loin, plus il détruira d'ennemis, plus il gagnera de la monnaie virtuelle et améliorer ses statistiques à nouveau.



Smashine est un jeu dans lequel le joueur est amené à taper le plus rapidement possible sur son écran pour détruire un distributeur dans le temps imparti.
Le joueur avance dans les niveaux en affrontant des machines de plus en plus difficiles à battre.
Une machine possède des points de vie et à chaque fois que le joueur effectue une tape sur l'écran, ses points de vie diminuent.
Une machine possède aussi des points sensibles sur lesquels le joueur peut taper pour infliger plus de dégâts. Ces points apparaissent à tour de rôle à l'écran et ne restent que quelques secondes. Il peut y avoir plusieurs points affiché simultanément à l'écran sur lesquels le joueur peut taper afin de détruire la machine encore plus rapidement.

Si le joueur tape un point sensible, il incrémente son combo de un. S'il tape la machine sans toucher de point sensible, la valeur de combo est réinitialisé.
Le joueur remporte la victoire s'il arrive a réduire les points de vie de la machine à zéro dans le temps imparti et passe au niveau suivant. Le joueur perd la partie sinon. Après avoir détruit un certain nombre de machines, une machine boss apparaît. Celle-ci possède plus de points de vie et est plus difficile a détruire que les machines standards. Pour briser la monotonie en détruisant des machines à la chaîne, les machines normales sont créées de manière aléatoire. Elles ont des couleurs différentes, les composants d'une machine changent de place et les machines boss ont des visuels uniques qui abordent des thèmes visuels divers et variés.

Un score est calculé à chaque fin de partie. Le calcul prend en compte le combo maximal réalisé dans le niveau et le temps restant du compte à rebours. Plus le joueur fait un combo élevé tout en terminant le niveau rapidement, plus son score sera élevé.
Une fois que le joueur a complété tous les niveaux, il peut revenir aux niveaux précédents pour essayer de battre son record. La rejouabilité du jeu est entièrement basée sur du **scoring** et la complétion de niveau.

F - Problématiques

Les principales problématiques auxquelles j'ai été confronté sont les suivantes :

- Concevoir des éléments de gameplay génériques (concernant les éléments reprenables de projet en projet) ;
- Élaborer des architectures de code maintenable et donner un maximum de marge au Game Designer pour paramétrer le jeu ;
- Produire du code efficace en terme de performance sur plateformes mobiles (produire des fonctionnalités dans les limites des ressources disponibles des smartphones et tablettes, toutes gammes confondues).

Les trois jeux hyper casual sur lesquels j'ai travaillé ont été réalisé en C# sous Unity 2019, et les **principes SOLID** devaient être respectés au maximum. Le terme SOLID dans le développement représente cinq principes importants qui imposent certaines règles dans la programmation orientée objet afin produire un code propre, bien architecturé, reprenable, flexible et facilement extensible.

2 - Production de fonctionnalités génériques

A - Contexte

Le Lead programmeur a adopté la programmation par événements pour architecturer le code mais aussi pour pouvoir créer des éléments de jeu génériques. Un programme est principalement défini par ses réactions aux différents événements qui peuvent se produire, c'est-à-dire des changements d'état de variable, l'incrémentation d'une liste, un mouvement de souris ou de clavier, etc.
La programmation événementielle peut être réalisée dans n'importe quel langage de programmation, bien que la tâche soit plus aisée dans les langages de haut niveau tel que C#.

L'objectif est de créer des éléments de gameplay qui se « branchent » les uns aux autres sans avoir besoin d'implémenter du code supplémentaire. Sur les futurs projets, le gain de temps de production au niveau de la programmation est notable en réutilisant des éléments de gameplay qui réapparaissent souvent dans les jeux hyper casual. Cela permet, à plus long terme, d'investir plus de temps sur l'implémentation de mécaniques de gameplay propre à un projet, et potentiellement passer plus de temps à élaborer ou tester des éléments de gameplay plus complexes et plus originales. Chaque brique de gameplay doit être, dans la limite du possible, indépendantes les unes des autres.

Dans un premier temps, nous allons voir comment les éléments de gameplay sont implémentés dans nos jeux grâce aux événements et dans un second temps comment ont ils été rendus génériques et indépendants.

B - Fonctionnement des événements

Dans notre cadre, l'envoi d'événement sert à signaler le changement d'un état de jeu. Il peut s'agir de la mort d'un joueur, d'un butin collecté ou même le début d'une partie.
Les événements peuvent indiquer n'importe quel changement d'état et permettent aux autres éléments du jeu de réagir de manière spécifique à chaque événement donné. Par exemple, lorsque l'événement de début de jeu est lancé, les contrôles du personnage s'activent, la barre de vie du joueur s'affiche, son score est mis à zéro, etc. Ces événements peuvent aussi diffuser des informations lorsqu'ils sont déclenchés, ce qui permet aux différents objets de communiquer entre eux en se donnant des informations. Supposons qu'un événement de début de partie est lancé et que cet événement envoie en paramètre le numéro du niveau : le joueur reçoit l'information du début d'une partie, ainsi que le numéro, et voit que ce dernier correspond à un niveau spécial dans lequel le joueur ne peut pas faire de saut.

Pour présenter la manière dont les éléments du jeu sont conçus, imaginons un jeu dans lequel nous souhaitons que le joueur puisse marquer des points lorsqu'il réalise une certaine action. Nous aurons alors un module permettant de stocker ces points et de les augmenter lorsqu'il exécute l'action en question, module que nous appellerons 'ScoreModule'. Dans l'optique d'expliquer cet exemple dans la partie suivante, nous souhaitons avoir le comportement suivant : augmenter le score d'un certain montant à chaque fois qu'un ennemi est tué.

C - Implémentation

Au démarrage du jeu, nous abonnons la méthode d'augmentation de score dans ScoreModule à l'événement de mort d'un ennemi. On dit que le ScoreModule « écoute » les événements liés à la destruction d'un ennemi.

```
private void OnEnable()
{
    // note : Abonnement de la méthode IncreaseScore à l'évenement OnKilled de l'objet Enemy
    Enemy.OnKilled += IncreaseScore;
}
```

Cette opération est réalisée dans le bloc 'OnEnable' de Unity. C'est à dire que dès que l'objet ScoreModule est activé, celui ci se mettra à réagir aux événement OnKilled venant d'un objet de type Enemy. Lorsque l'événement correspondant à la mort d'un ennemi est déclenché, le ScoreModule va lancer la méthode d'incrémentation de score 'IncreaseScore'.

```
// note : Si l'ennemi est touché par une arme de joueur, l'ennemi est tué
0 références
private void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("PlayerWeapon"))
        Kill();
}


// note : Méthode qui se charge d'exécuter la mort de l'ennemi.
1 référence
private void Kill()
{
    BroadcastEnemyKilled();
    Destroy(gameObject);
}


/* note :
Méthode qui envoie l'événement de la mort de l'ennemi
avec en paramètre sa valeur de score
Valeur de score qui sera utilisé dans le ScoreModule
pour incrémenter le score.
*/
1 référence
private void BroadcastEnemyKilled()
{
    if (OnKilled != null)
        OnKilled(m_enemyScoreValue);
}
```

OnTriggerEnter est une méthode d'Unity chargée de détecter les collisions. Lorsqu'une collision a lieu avec l'objet courant, s'agissant d'un ennemi dans notre cas, cette dernière renvoie un objet de type Collider contenant les informations sur la collision. Dans notre cas, on cherche juste savoir si l'ennemi a été touché par l'arme du joueur.

Si l'ennemi est touché par une arme du joueur (PlayerWeapon) dans la méthode OnTriggerEnter, celui-ci exécute la méthode Kill. Cette méthode envoie l'événement OnKilled correspondant à la mort de l'ennemi et ayant en paramètre, la valeur de score de cet ennemi.

Notre ScoreModule, étant abonné à l'événement OnKilled, réagit et par conséquent incrémente le score du joueur du même montant que la valeur de score reçue en paramètre.

```
// note : Méthode qui incrémente la valeur du score stockée par le montant donné en paramettre
2 références
private void IncreaseScore(float scoreValue)
{
    m_score += scoreValue;
    BroadcastScoreValue();
}
```

D - Généricité et réutilisation

Cet élément de gameplay, qu'est ScoreModule, doit être flexible et réutilisable de projet en projet. Grâce à cette implémentation par événements, le ScoreModule peut être réutilisé dans d'autres projets. Le module n'est chargé que d'incrémenter un nombre **flottant** avec une valeur qui est donnée en paramètre. Si un nouveau projet présente une notion de score, il suffit réimporter le ScoreModule et de changer l'abonnement en question dans le bloc OnEnable de Unity.

Si dans ce nouveau projet on doit incrémenter le score en collectant des pièces, nous aurons dans le ScoreModule le changement suivant :

```
private void OnEnable()
{
    // note : Abonnement de la méthode IncreaseScore à l'évenement OnCollected de l'objet Coin
    Coin.OnCollected += IncreaseScore;
}
```

La méthode IncreaseScore est exécutée que si une pièce est collectée.
Et voici comment serait définie une pièce, la classe Coin:

```
public class Coin : MonoBehaviour
{
    public static Action<float> OnCollected;

    [SerializeField]
    private float m_coinValue = 10.0f;

    // note : si la pièce touche un joueur, déclanche l'envoie de l'événement
    0 références
    private void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Player"))
            BroadcastCoinCollected();
    }


    /* note :
    Méthode qui envoie l'événement associé à la collecte de la pièce
    */
    1 référence
    private void BroadcastCoinCollected()
    {
        if (OnCollected != null)
            OnCollected(m_coinValue);
    }
}
```

Si un joueur touche la pièce, un événement correspondant à la collecte de cette dernière OnCollected est envoyée avec sa valeur en paramètre. Le ScoreModule réagira à cet événement et incrémentera le score d'un montant égal à la valeur reçue.

On voit donc que le ScoreModule peut se « brancher » à n'importe quel objet pour gérer le score du joueur tant que l'événement auquel le module est abonné reçoit une valeur flottante. Ce module est complètement indépendant et peut être implémenté dans n'importe quel projet.

Le comportement du ScoreModule est flexible et peut être étendu en implémentant, par exemple, une méthode diminuant le score ou alors, augmenter la valeur de l'incrémentation en fonction d'un autre facteur tel qu'un **combo** (comme c'est le cas dans le projet Slashing Machine).

E - Réduction des dépendances entre les objets

La programmation par événements permet de réduire un maximum les dépendances entre les objets. C'est-à-dire qu'un objet ne doit pas dépendre d'un autre de manière explicite. Si dans un projet, nous avons deux objets A et B qui sont implémentés et que A dépend de B ; lorsqu'on passe à un autre projet où nous réutilisons uniquement l'objet A (car nous n'avons pas besoin d'implémenter l'objet B dans le nouveau projet), le code plantera car l'objet B n'est pas défini dans ce dernier.

Dans notre exemple présenté plus haut, dans le premier projet, l'incrémentation se fait lorsqu'un ennemi est tué. Si cet élément de gameplay avait été implémenté en introduisant des dépendances entre les objets, nous aurions le code suivant pour l'objet Enemy :

```
/* note : Introduction d'une dépendance ! Si le ScoreModule
n'est pas référé sur TOUS les ennemis, le code plantera.
*/
[SerializeField]
private ScoreModule m_scoreModuleReference;


/* note :
flottant qui stock la valeur de score d'un 'ennemi'.
[SerializeField] rend la variable accessible dans
l'inspecteur d'Unity (paramettrable par le Game Designer)
*/
[SerializeField]
private float m_enemyScoreValue = 10.0f;


// note : Si l'ennemi est touché par une arme de joueur, l'ennemi est tué
0 références
private void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("PlayerWeapon"))
        Kill();
}


// note : Méthode qui se charge d'exécuter la mort de l'ennemi.
1 référence
private void Kill()
{
    m_scoreModuleReference.IncreaseScore(m_enemyScoreValue);
    Destroy(gameObject);
}
```

La classe Enemy est dépendante du ScoreModule car elle a besoin de la référence au ScoreModule pour incrémenter le score. Dans un projet où on réutilise ce script Enemy, le code plantera s'il n'y a pas d'implémentation de ScoreModule.

Et pour l'objet ScoreModule :

```
/* note : Méthode qui incrémente la valeur du score
stockée par le montant donné en paramètre
*/
3 références
public void IncreaseScore(float scoreValue)
{
    m_score += scoreValue;
    BroadcastScoreValue();
}
```

Premièrement, dans l'objet Enemy, si nous ne concevons pas le code de façon générique, il est nécessaire d'introduire une référence au ScoreModule. Cette derrnière permet d'accéder à la méthode d'incrémentation de score lorsqu'un ennemi est tué, comme on peut le voir dans la méthode Kill. Si cette référence est manquante, le code plante et demande une modification. Si tous les éléments d'un jeu présente des dépendances comme celles-ci, cela demanderait un temps conséquent pour faire les modifications de scripts.

Deuxièmement, dans le ScoreModule, l'implémentation est pratiquement la même qu'avec l'implémentation par événement, sauf que la méthode d'incrémentation du score a été rendue publique pour être accessible depuis d'autres objets. Si un objet possède la référence au ScoreModule, il est capable d'incrémenter le score comme il le souhaite, alors que ce n'est pas un effet voulu : nous ne voulons pas qu'il soit possible d'incrémenter ce dernier à partir d'objet n'ayant pas de rapport avec la gestion du score. En rendant la méthode d'incrémentation de score privée et en faisant réagir l'objet à un événement précis, on s'assure de la sécurité du code par une bonne encapsulation.

Il aurait été possible d'incrémenter le score par l'utilisation d'un **singleton** dans la classe ScoreModule, c'est-à-dire avoir accès à l'instance unique de la classe ScoreModule et lui ordonner d'exécuter la méthode IncreaseScore. Voici comment le singleton aurait été implémenté dans la classe ScoreModule : Cette instance unique est rendue accessible depuis l'extérieur via un champ **static** nommé « instance ». Cette opération, réalisée dans la méthode Awake d'Unity, empêche toute autre instanciation de cette classe.

```
public static ScoreModule instance;

/* note : implémentation du singleton
dans la classe ScoreModule
*/
0 références
private void Awake()
{
    if (instance == null)
        instance = this;
    else if (instance != this)
        Destroy(gameObject);
}
```

La présence d'un singleton n'est justifiable que lorsque l'on constate un besoin d'un objet devant être commun à toutes les classes. Or dans notre cas, le score ne doit être incrémenté que lorsqu'un ennemi est tué, les autres classes du projet n'ayant pas besoin d'accéder au ScoreModule.

De plus, on introduit une dépendance dans la classe Enemy qui dépendra alors du ScoreModule, puisque lorsqu'un ennemi meurt, il doit accéder à l'instance du ScoreModule. Si on démarre un nouveau projet et qu'on souhaite réutiliser le script Enemy, la dépendance de la classe Enemy envers la classe ScoreModule rendra l'opération impossible. Cela impliquera de changer le comportement de la méthode Kill dans la classe Enemy ce qui va à l'encontre de la direction prise par l'équipe pour la production de fonctionnalités réutilisable.

La capture d'écran suivante illustre la classe Enemy ayant accès au ScoreModule. Dans le cas où le ScoreModule n'existe pas, le code plantera.

```
// note : Méthode qui se charge d'exécuter la mort de l'ennemi.
1 référence
private void Kill()
{
    ScoreModule.instance.IncreaseScore(m_enemyScoreValue);
    Destroy(gameObject);
}
```

F - Conclusion

L'implémentation des briques de gameplay par événements nous permet dans un premier temps de concevoir des éléments génériques réutilisable de projet en projet sans modifier les scripts, et dans un second temps d'éviter l'introduction des dépendances entre les objets, pouvant compliquer la gestion des objets si ces derniers sont utilisés dans d'autres projets.

Voici quelques exemples d'éléments génériques ayant été réutilisés au fil des projets grâce à une implémentation par événements :

- Les scripts de contrôles, permetant de réagir aux **inputs** du joueur (sur l'écran du smartphone). Les inputs de Tap/Clique, Swipe/Balayage et Hold/Maintient envoient chacun des événements avec les informations nécessaires. Le Tap enverra un événement OnTap avec ses coordonnées (à l'endroit tapé par l'utilisateur), le Swipe enverra son **vecteur directeur** (pour savoir dans quelle direction l'utilisateur a glisser son doigt) et le Hold envoie un événement avec les coordonnées du point maintenu à l'écran ;
- Le score manager, gérant le score du joueur présenté dans cette partie ;
- Le game manager qui permet de suivre l'état du jeu actuel ;
- Des scripts utilitaires responsables de mouvements d'objet ;
- Des modules d'UI ne nécessitant qu'un changement d'apparence.

3 - Élaboration d’architectures de code maintenables

A - Contexte

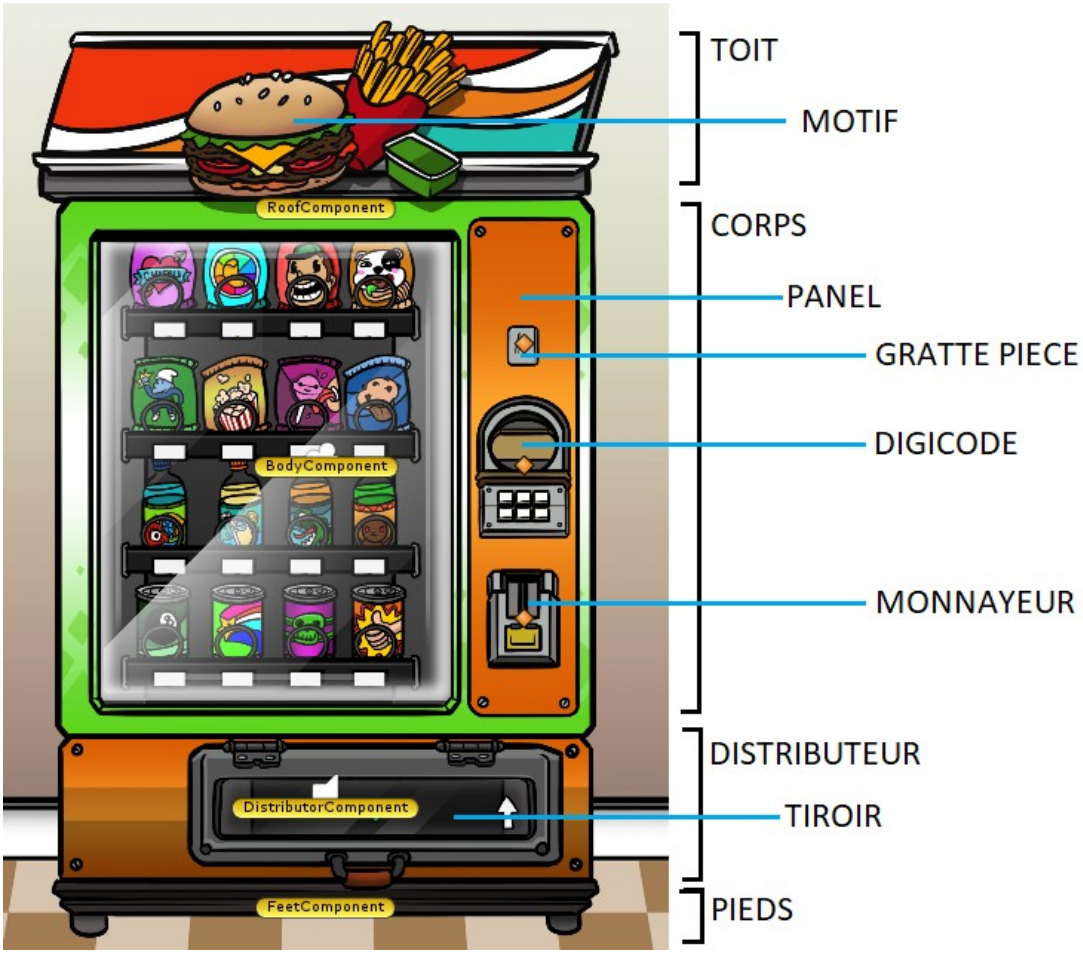
Lorsque l’équipe de production décide de lancer un nouveau projet, il est très important au niveau de la programmation de bien concevoir l’architecture global du jeu. Cette phase de réflexion est très importante car elle permet de construire le jeu sur des bases solides et d’éviter des régressions. Elle permet également d'être plus flexible, de maintenir et faire évoluer le code beaucoup plus facilement, d'être plus efficace dans le développement et aussi, d'éviter ce que l'on appelle le « **Development Hell »** qui coûterait cher à la production. C’est à ce moment là que nous concevons la manière dont les briques de gameplay vont être implémentées et que nous anticipons au maximum les problèmes potentiels auxquels nous pourrons être confronté. Ainsi, nous pouvons penser à une architecture permettant d’éviter certaines erreurs comme se retrouver dans des impasses et faire des retours en arrière.

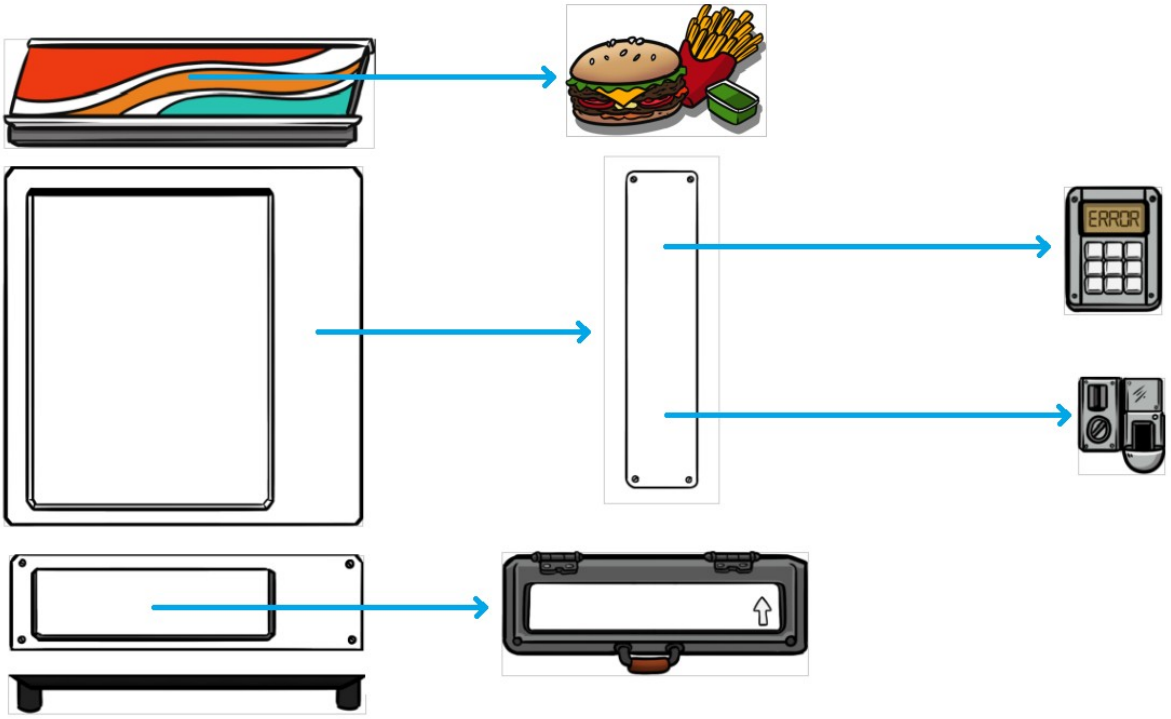
Une fonctionnalité importante du jeu *Smashine* a nécessité beaucoup de réflexion sur son implémentation : la génération de machine aléatoire.

B - Objectif et problématiques

Dans le jeu *Smachine*, les machines doivent être générées aléatoirement, c'est à dire que chaque élément qui compose une machine doit être pris aléatoirement. En l'occurrence il s'agit de **sprite** étant donné que la machine est en 2d.
Une machine est composée d'un toit, d'un corps, d'un distributeur et de pieds. Chacun de ces composants peut avoir en superposition, d'autres composants. Par exemple, le corps tient un panel (panneau) et ce panel tient un monnayeur et un digicode.

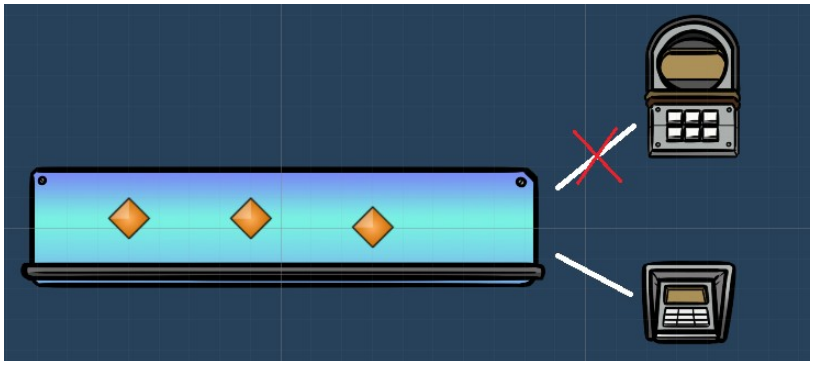
Voici une machine complètement construite :



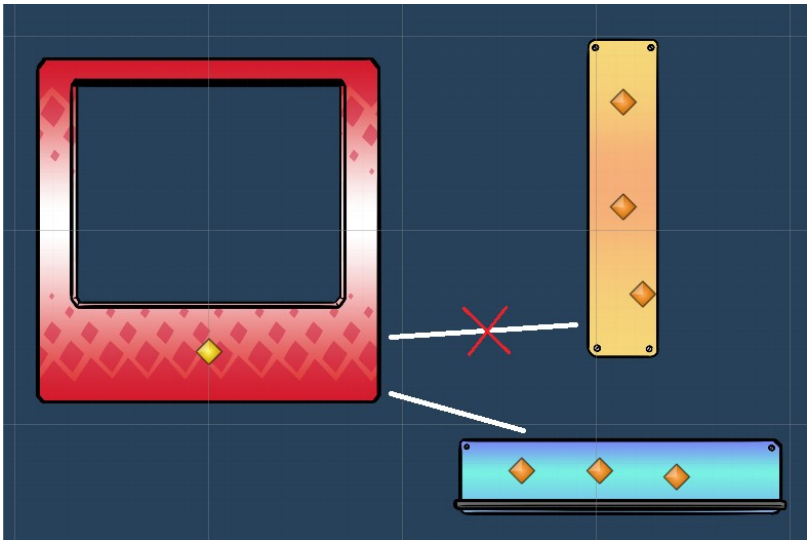


Lorsqu'on arrive à la création du corps, un sprite de corps est choisi aléatoirement. Puis un sprite de panel est choisi de la même manière pour se poser par dessus le corps et ainsi de suite jusqu'à la construction complète de la machine.

La construction d'une machine doit respecter certaines contraintes : certains sprites sont de taille ou d'architectures différentes, rendant certains sprites incompatible entre eux.



Contrainte de taille des sprites : certains éléments peuvent déborder du sprite sur lequel ils sont posés. Sur le schéma ci contre, le digicode en haut à droite n'est pas compatible avec le panel horizontal.



Contrainte d'architecture des sprites : certains sprites ont été créé avec une architecture de machine différente. Le sprite de corps en rouge a été créé pour ne supporter que des panels horizontaux comme le panel bleu, et non verticaux, tel que le panel jaune.

Pour donner la sensation au joueur que chaque machine est unique, on crée des machines ayant une architecture globale qui est la même, c'est-à-dire, un toit, un corps, un distributeur et des pieds mais ayant des composants pouvant présenter plusieurs architectures différentes.

Cependant, avec les contraintes présentées plus haut, il est impossible de laisser la génération complètement aléatoire, car il faut éviter la combinaison de certains sprites entre eux pour éviter la création de machines incohérentes.

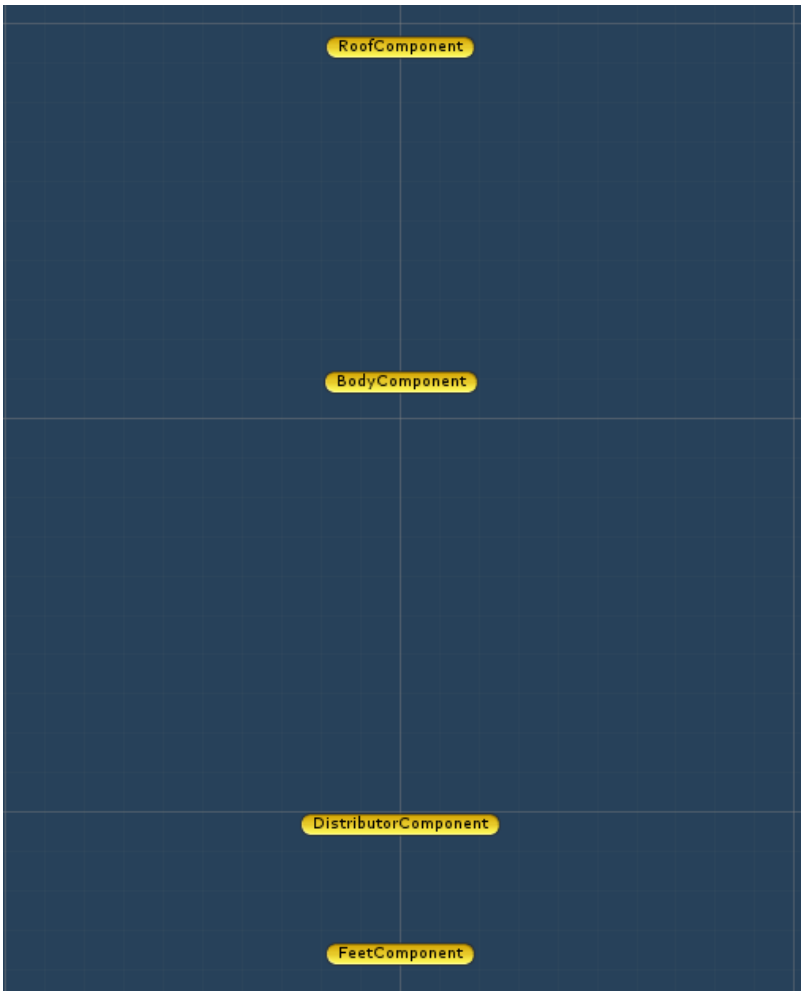
C - Première implémentation

1 - Architecture et fonctionnement

Dans la première implémentation fonctionnelle de la construction de machine, nous avons élaboré un système de squelettes imbriqués. Ces derniers ne contiennent que des positions de partie de machine.

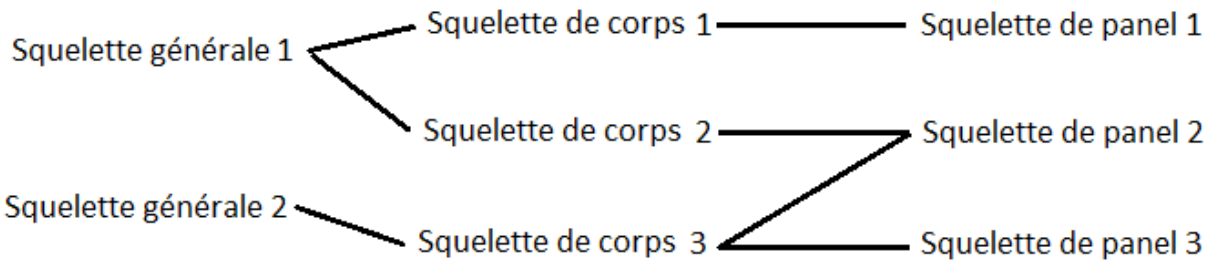
Dans notre cas nous avons besoin du squelette général contenant que les positions des premiers composants de la machine (le toit, le corps, le distributeur et les pieds). Ensuite nous avons le squelette du corps contenant la position du panel, puis le squelette du distributeur avec la position du tiroir et enfin le squelette du panel avec les positions du monnayeur et du digicode.

Voici à quoi ressemble le squelette générale :

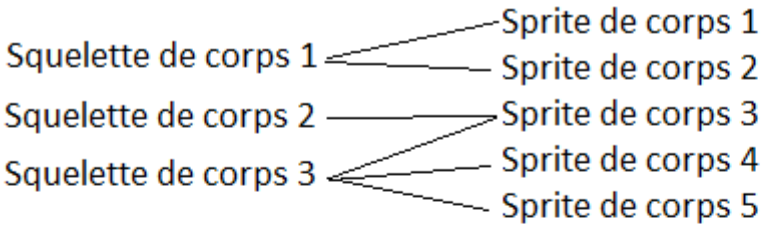


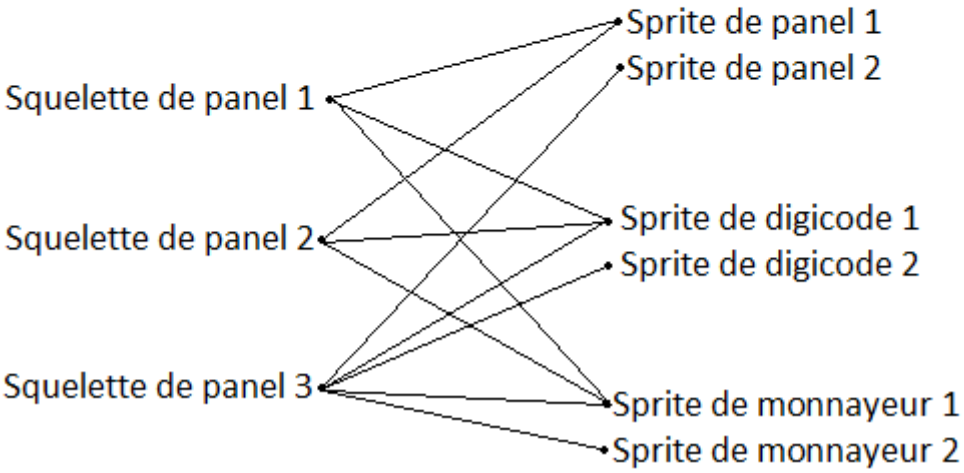
Pour utiliser ces squelettes, nous avons créé un script prenant en paramètre tous les squelettes possibles. C'est-à-dire, pour un squelette général, il faut référer tous les squelettes de corps et de distributeur qui lui sont compatibles. Et pour tous les squelettes de corps, il faut référer tous les squelettes de panel compatibles. Enfin, pour respecter les contraintes de taille et d'architecture de sprites, pour chaque squelette, on réfère uniquement les sprites qui leurs sont compatibles.
Par exemple, pour un squelette général donné, nous pouvons avoir deux squelettes et sprites de corps qui lui sont compatible. Puis pour chacun de ces squelettes de corps, on renseigne tous les squelettes et sprites de panel qui leurs sont compatibles et ainsi de suite pour tous les éléments de la machine.

Voici un schéma illustrant l'organisation des compatibilités entre squelettes :



Et un autre schéma pour la compatibilité entre les squelettes et les sprites :





Pour résumer, dans la construction d'une machine, un squelette général est pris au hasard. De là, on ne dispose que des squelettes et des sprites qui lui sont compatibles, puis un squelette de corps compatible est choisi au hasard. Ainsi, on dispose des squelettes et des sprites compatibles avec le squelette de corps. Et ainsi de suite pour tous les éléments constituant la machine. Par exemple, si on a aléatoirement reçu le squelette général 1, on a uniquement accès aux squelettes de corps 1 et 2. Suite au choix aléatoire, si on a le squelette de corps 1, nous n'aurons accès qu'aux squelettes de panel 1 et aux sprites de corps 1 et 2. Et avec le panel 1, nous pouvons avoir les sprites de panel 1, de digicode 1 et de monnayeur 1.
De cette manière, on peut respecter les contraintes de taille et d'architecture de sprites en définissant explicitement les sprites et les squelettes qui sont compatibles entre eux.

2 - Évaluation de l’implémentation

Cette implémentation était fonctionnelle et répondait aux besoins, ainsi qu’aux critères de construction de machine aléatoire, tout en respectant les contraintes.

Nous nous sommes rendu compte que ce système n’était pas pratique pour l’équipe : il fallait référer tous les sprites compatibles pour chaque squelette et le système était lourd à utiliser car il fallait créer tous les squelettes possibles dans des objets à part. Il pouvait également y avoir des cas de redondance dans le référencement des squelettes et des sprites compatibles.

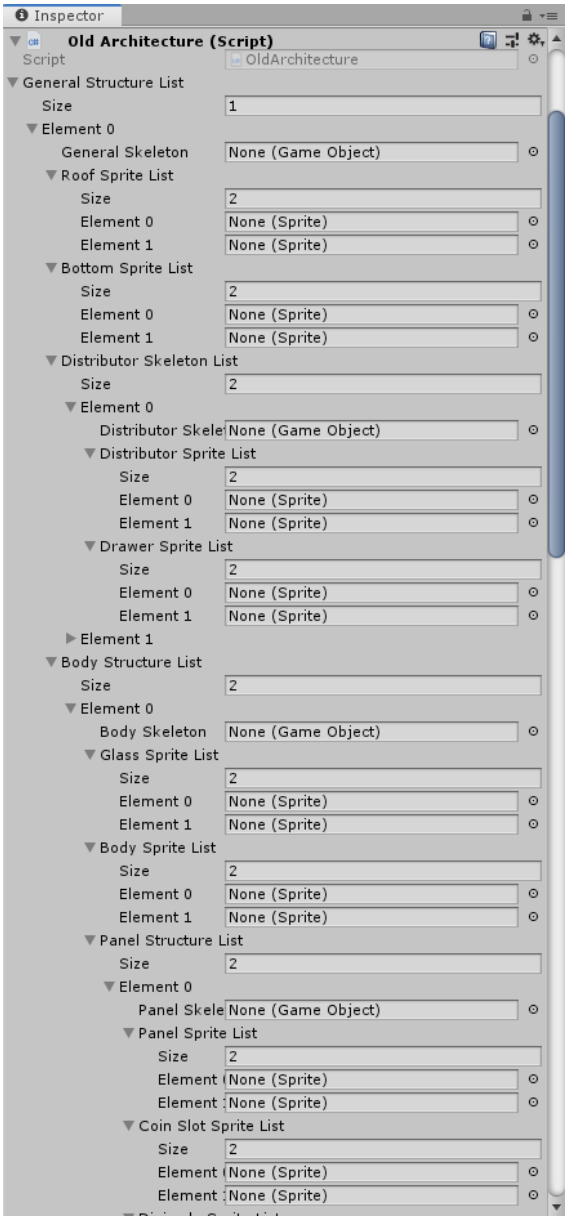
Imaginons que nous avons deux squelettes de corps pouvant avoir un panel vertical. Dans le premier squelette, le panel vertical est à droite et dans le second, à gauche. Avec l’implémentation actuelle, on ne pouvait pas gagner de temps en réutilisant les mêmes paramètres du panel vertical. Il fallait référencer tous les squelettes et sprites compatibles à nouveau.

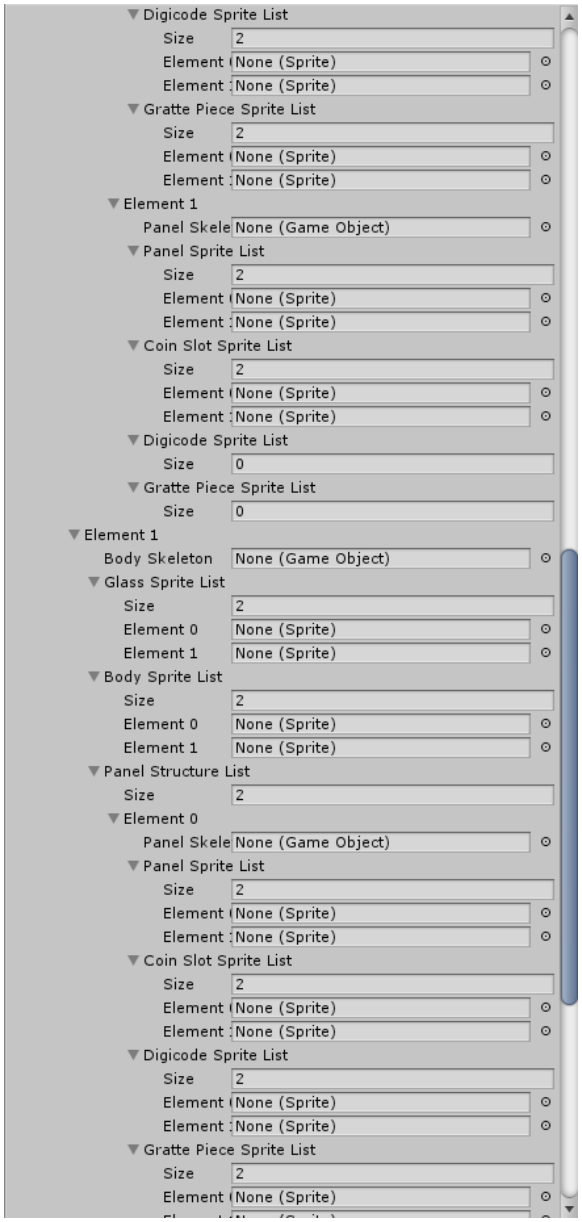
Avec ce système, plus la machine est complexe, plus elle présente d’éléments et possède de grande chance d’avoir des redondances. De plus, si le Game Designer se trompe dans les référencements, certaines machines peuvent présenter des combinaisons de sprites qui sont incompatibles.

Ensuite, Unity stocke les informations telles que le paramétrage d’un script dans des fichiers .meta. Lorsque l’on référence les squelettes et le sprites compatibles entre eux dans le script, toutes ces informations de référencement sont stockés dans un fichier .meta. Lorsque l’objet responsable de la gestion des squelettes et des sprites est créé, il génère un fichier .meta qui lui est propre et qui stockera les paramètres du script. Si ce fichier contenant tous les référencements venait à être perdu ou supprimé, il faut recommencer ce travail de référencement depuis le début. On ne pouvait pas se permettre de travailler avec cet outil qui demande beaucoup de temps à mettre en place et d’attention pour éviter les erreurs.

Enfin, le Game Designer ne disposait pas de prévisualisation de la machine lorsqu’il voulait créer une nouvelle architecture. Il devait pour cela, créer de toutes pièces une machine exemple et sauvegarder la position de toutes les parties de la machine pour en faire des squelettes.

Voici des captures d'écran montrant l'interface que le Game Designer devait utiliser :





Il est très facile de se perdre dans ce genre d'interface. Tous ces paramètres qui forment cette liste monumentale ne permettait de générer qu'une poignée de machine différentes. Pour générer plus de machines aléatoires, il fallait renseigner plus d'éléments, ce qui augmente davantage la taille de cette liste, ainsi que les chances de faire des mauvais référencements.

3 - Conclusion

Toutes les raisons énoncées précédemment nous ont poussé à repenser l'architecture de la construction de machine pour éviter d'investir trop de temps dans la construction de squelette et dans leurs référencements. Nous avons voulu rajouter plus de sécurité en ce qui concerne les référencements en rendant la configuration des machines plus unitaire. Plutôt que d'avoir tout référencé dans un seul et même endroit comme présenté dans la capture d'écran plus haut, il est préférable de segmenter tous ces référencements pour éviter de tout perdre d'un coup et pouvoir apporter des modifications plus facilement en évitant de chercher dans un script interminable pour changer ou ajouter des éléments.

D - Seconde implémentation

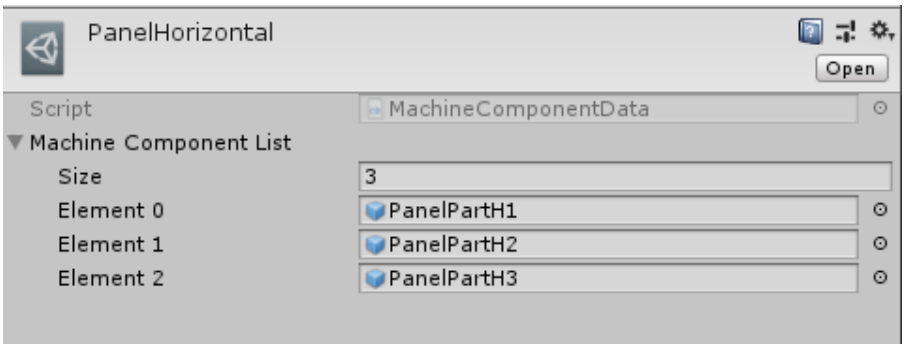
1 - Architecture

Pour régler ces problèmes de difficulté d'utilisation, nous nous sommes tourné vers un système utilisant les **prefabs** d'Unity et l'instanciation en cascade.

Premièrement, pour le problème de référencement complexe dans la première implémentation, nous avons utilisé des **ScriptableObjects** de Unity. Dans notre cas, nous les avons utilisés pour regrouper tous les sprites compatibles entre eux. Par exemple, nous avons regroupé tous les sprites de panel verticaux ensemble, tous les sprites de monnayeurs compatible avec les panels horizontaux, etc. Ainsi, il était plus facile de faire respecter les contraintes de construction de la machine à travers des conteneurs de données indépendants les uns des autres. Les sprites sont classés dans des conteneurs de données selon leurs compatibilités.

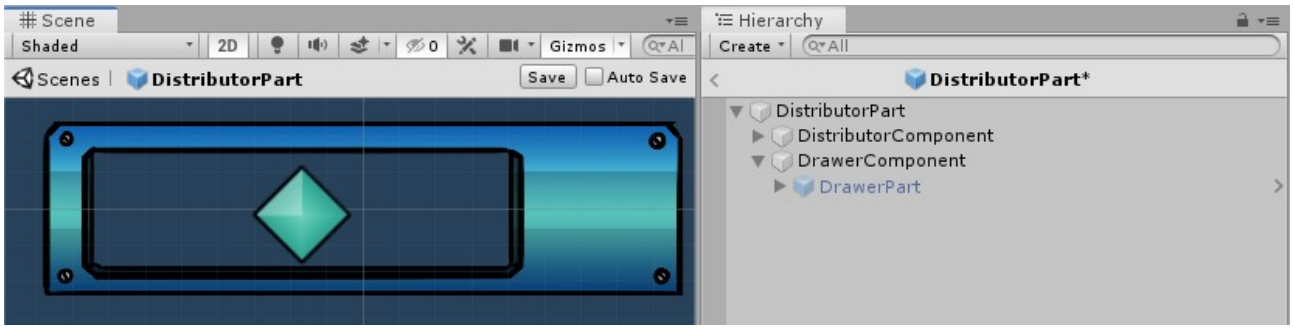
Ci dessous, les capture d'écran concernant les objets cités ci dessus :

- À gauche, la liste des conteneurs regroupant les sprites compatibles entre eux ;
- À droite, les sprites de panels horizontaux regroupés dans un *ScriptableObject.*



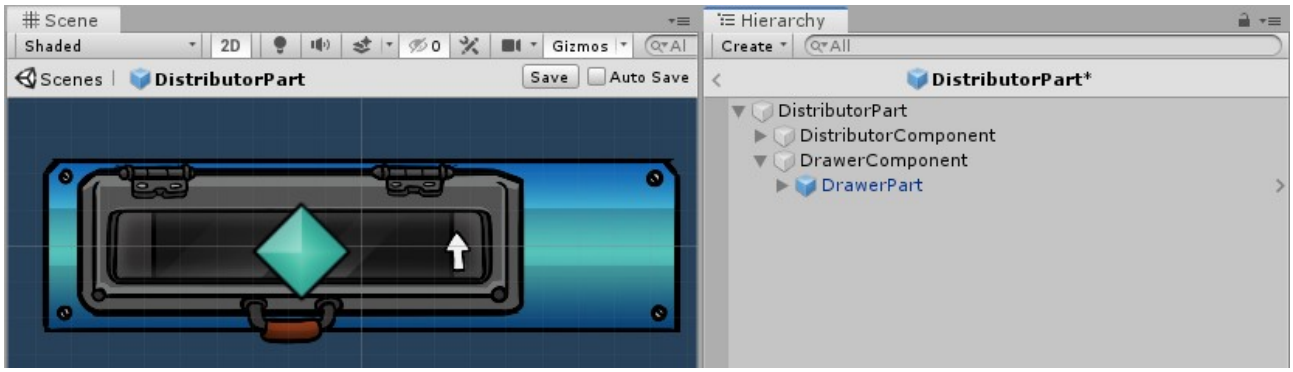
Deuxièmement, la notion de squelette est toujours présente mais est masqué puisque les positions des éléments d'une machine sont stockés directement dans les prefabs. Au lieu de créer des objets pour stocker un squelette, on se sert des prefabs pour pré-positionner les éléments d'une machine : le système de squelette est plus compréhensible et visuel pour l'ensemble de l'équipe.

Voici une capture d’écran montrant le paramétrage de certaines parties de machines :

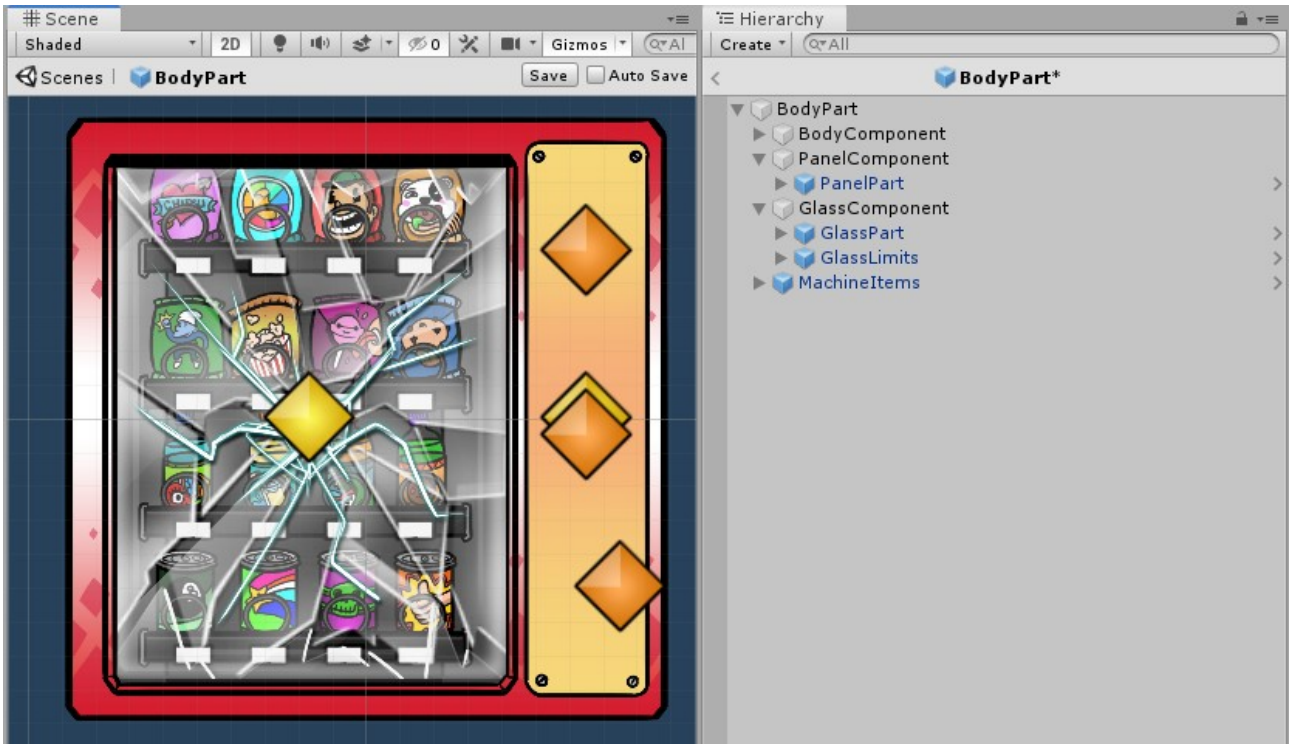


Les losanges sont des repères visuels pour savoir où le sous composant sera instancié lors de la génération aléatoire. Le Game Designer peut ainsi déplacer ces points d’instanciation pour bien positionner les sous composants de la machine. Il a la possibilité de prévisualiser la position du composant pour bien placer son repère.

Voici une capture d’écran avec la prévisualisation :



Voici une autre capture d'écran montrant le paramétrage d'un élément plus complet de la machine, le corps :



Le Game Designer peut déplacer chaque élément à sa convenance en déplaçant les objets ayant le mot « Component » dans la fenêtre « Hierarchy ».

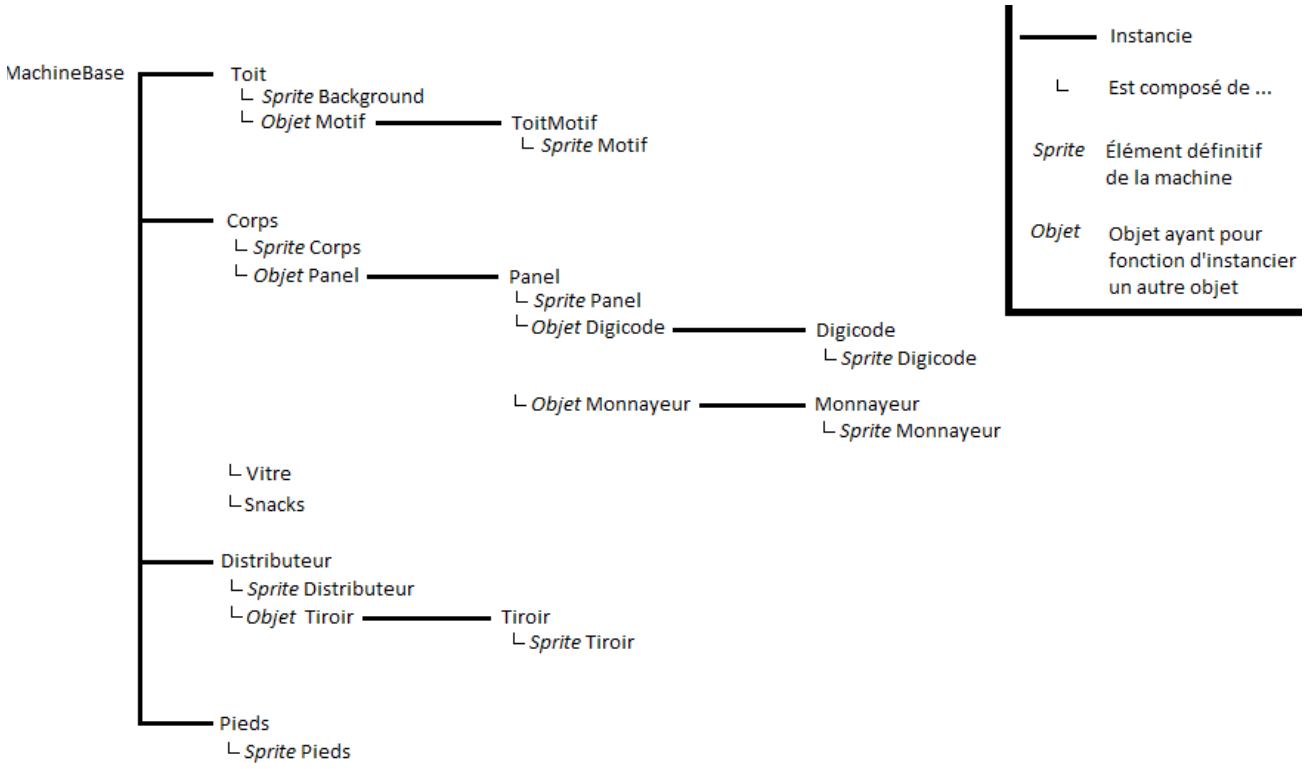
2 - Fonctionnement

Pour construire une machine avec cette nouvelle implémentation, nous instancions un prefab de base que nous appellerons MachineBase. Celui-ci contient des objets représentant chaque élément de la machine de base (le toit, le corps, le distributeur et les pieds).
Un script d'instanciation en cascade est attaché à chacun de ces objets et prend en paramètre un ScriptableObject dont le but est d'injecter des données : de manière plus précise, ce script a pour fonction d'instancier un des objets contenus dans les données de façon aléatoire lorsqu'il est activé.

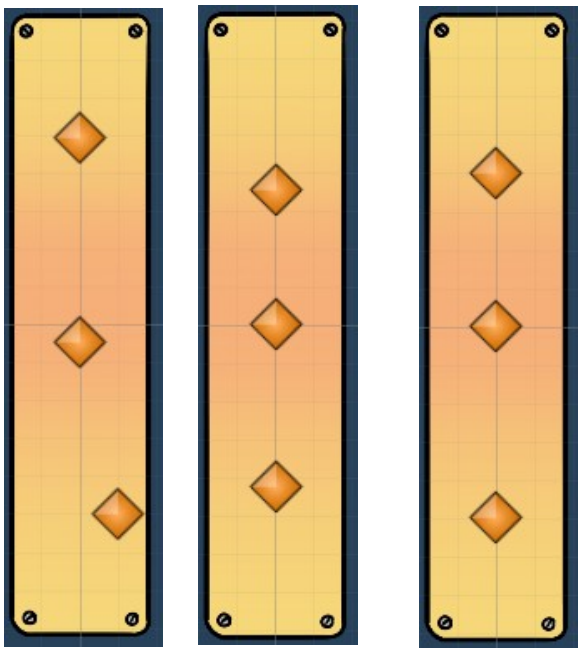
Au chargement d'un niveau, la MachineBase est instanciée. Les objets représentant le toit, le corps, le distributeur et les pieds vont instancier à leur tour un objet choisi au hasard dans leurs ScriptableObjet donné en paramètre. Ce dernier peut être un sprite (un élément définitif de la machine) ou un objet portant également un script d'instanciation en cascade, qui sera déclenché à son tour, et ainsi de suite jusqu'à la construction complète de la machine.

Voici un schéma pour mieux visualiser le processus de création de machine avec le système d'instanciation en cascade.

Le mot clé *Objet* indique que l'objet porte le script d'instanciation en cascade.



Pour la sélection des sprites et le respect des contraintes de taille et d'architecture de ces derniers, il suffit référer le ScriptableObject regroupant les sprites compatibles. Prenons l'exemple des panels. Nous avons créé trois prefabs de panel avec des configurations de monnayeur et de digicode différents pour rajouter de la diversité à nos machines.

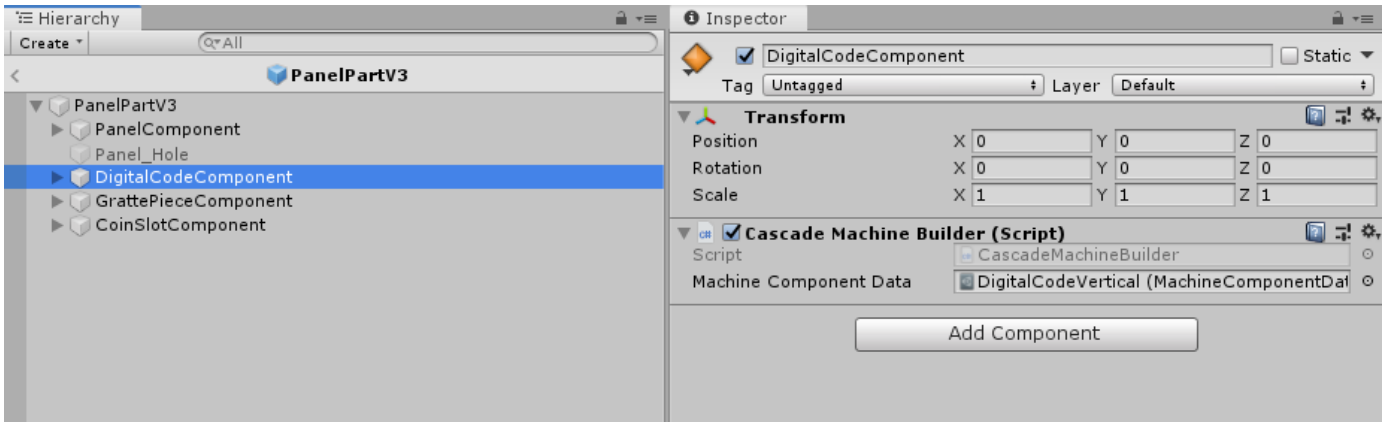


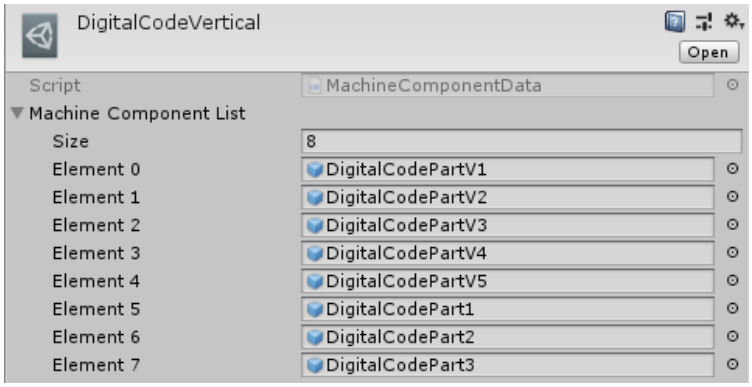
Les losanges orange sont des objets placés aux bons endroits sur le panel. Nous en voyons trois dans les captures d'écran mais nous prendrons uniquement que deux objets dans les exemples illustrés jusqu'à présent et futur pour simplifier les explications.

Chaque objet possède le script d'instanciation en cascade. Il suffit alors de donner les bons ScriptableObject en paramètre : c'est-à-dire le celui regroupant les monnayeurs ainsi que celui regroupant les digicodes compatibles avec les panels verticaux.

Ainsi, lorsqu'un panel vertical est instancié, il ne disposera que des sprites de monnayeurs et digicodes compatibles car il n'aura en référence que le ScriptableObject regroupant les bons sprites. Aussi, les cas de redondances ne peuvent pas exister avec ce système car si on a un autre panel vertical, nous lui donnons les mêmes ScriptableObject en paramètre.

En sélectionnant l'objet digicode (surligné en bleu sur la capture d'écran suivante), nous voyons qu'il possède le script *CascadeMachineBuilder* qui est chargé de l'instanciation en cascade. Celui-ci prend en paramètre un *DigitalCodeVertical* (un ScriptableObject regroupant tous les sprites compatibles avec les panels verticaux).





Ci-contre, le ScriptableObject *DigitalCodeVertical* regroupant la liste des sprites de digicode compatibles avec les panels verticaux.

Les prefabs de digicode de ce type étant regroupés dans cette liste, si un objet ayant un script d'instanciation en cascade dispose de ce ScriptableObject en paramètre, alors un des éléments de la liste sera pris au hasard et sera instancié.

Tous les composants d'une machine sont gérés de cette manière. Les éléments sont pré-positionnés et on leur donne les bonnes données pour éviter une génération de machine incohérente.

Si l'on prend un cas pratique, nous commençons avec MachineBase qui instancie le toit, le corps, le distributeur et les pieds. Le toit va choisir aléatoirement un élément compatible avec la structure de MachineBase en lui spécifiant les données contenues dans le bon ScriptableObject. Le corps va instancier la vitre, les snacks de la machine, le panel et choisir aléatoirement une structure de corps : Dans le cas où le corps de la machine ne peut avoir que des panels horizontaux, le panel instancié choisira au hasard une structure de panel horizontale et instanciera les monnayeurs et digicodes. Par cascade, lorsque le monnayeur est créé, celui-ci va choisir un sprite de monnayeur compatible avec le panel horizontal.

3 - Conclusion

Ce système permet la création de machines aussi complexe que l'on veut, tant qu'on regroupe correctement les éléments dans des ScriptableObject et qu'on attache le script d'instanciation en cascade aux bons composants de la machine. Ce système est suffisamment flexible pour gérer des cas particuliers tel que les machines boss, pouvant présenter deux tiroirs ou aucun monnayeur.

4 - Production de fonctionnalités optimisées

A - Contexte

Bien que ces dernières années, les performances des smartphones (et tablettes) ont augmenté de manière significative, la majorité du parc mobile reste malgré tout derrière les consoles et les PC, par leurs ressources limités, tant en mémoire qu'en puissance de calcul.

Les technologies pour développer les jeux ont évolué pour atteindre un plus haut niveau de réalisme ou d'immersion. Cependant la plupart des appareils les plus modestes ne sont pas capable de supporter les dernières avancées technologiques. En développement mobile, il est obligatoire de penser aux limites des performances des appareils et de concevoir des fonctionnalités n'excédant pas ces limites. Malgré a faible taille des jeux hyper casual, il n'est pas impossible qu'ils peuvent consommer beaucoup de **mémoire vive** et de **temps processeur**, d'où l'importance en programmation, il est important de produire du code bien architecturé et optimisé. Cependant, il n'est pas nécessaire d'optimiser prématurément un projet si ce dernier ne présente aucun problème de performance lorsqu'il est installé sur un smartphone peu puissant. Il est alors plus judicieux d'investir des ressources dans l'amélioration graphique ou gameplay d'un jeu et faire l'optimisation à une étape plus évoluée du développement (ce qui n'empêche pas de vérifier les performances du jeu avec le profiler de Unity).

Pour vérifier si nos jeux respectent les limites des ressources, nous avons à notre disposition un smartphone haut de gamme et un autre bas de gamme (même chose pour les tablettes), l'objectif étant d'avoir trente images par secondes au minimum sur les deux appareils.

Nous avons créé des applications exécutables de nos jeux à des moments clé du développement. En général, après l'implémentation d'une brique de gameplay majeure, après la réimportation d'un package provenant de précédant projets (tel que les *UI* et certains scripts utilitaires) ou après l'intégration d'une quantité importante d'éléments graphiques. Il fallait tester les performances de nos jeux assez souvent pour pouvoir déterminer plus facilement quelle est, ou quelles sont, les fonctionnalités responsables d'une baisse de performance.

Dans notre cadre, nous avons produit des éléments optimisés dans la limite de notre expertise sur Unity : l'équipe de développement n'a pas défini un **pipeline de production** solide (compte tenu du fait que l'entreprise s'est lancée dans l'hyper casual) et cela nous a amené à rechercher des moyens d'optimiser nos jeux. Nous avons axé nos recherches sur des systèmes couramment utilisés pour optimiser les performances des jeux sur plateforme mobile. Dans cette section, nous allons voir quelques moyens mis en place pour économiser des ressources sur nos jeux.

B - Système de Pooling

L'un des systèmes mis en place, ayant été fortement bénéfique, est le système de **pooling**. Avant de présenter ce système, nous devons expliquer en quoi il a un intérêt.

1 - Fonctionnement du Garbage Collector

Unity utilise un **Garbage Collector** (GC), de style « stop the world », pour gérer automatiquement sa mémoire pour son allocation ou sa libération. La récupération de mémoire par le GC nécessite un temps processeur considérable et le programmeur doit donc s'adapter au fonctionnement du GC pour le solliciter le moins souvent possible ou l'utiliser le plus intelligemment possible. Pour déterminer quels blocs de mémoire ne sont plus utilisés, le gestionnaire de mémoire effectue une recherche dans toutes les variables de référence actuellement actives et marque les blocs auxquels ils se réfèrent comme «actifs». À la fin de la recherche, tout espace entre les blocs actifs est considéré comme vide par le gestionnaire de mémoire et peut être utilisé pour des allocations ultérieures.

Pour éviter les appels trop fréquents du GC, ce qui peut demander beaucoup de ressources, nous avons implémenté un système de *pooling*.
Le principe du *pooling* est d'instancier un ou plusieurs objet fréquemment utilisés dans le jeu au démarrage de l'application au lieu de les instancier pendant que le jeu est en cours d'exécution. Cela impactera le temps de chargement au lancement de l'application mais il permet d'avoir un gain de performance lorsque le jeu est en cours d'exécution.

Instancier ou détruire des objets à l'exécution peut potentiellement amener des baisses de performances visibles par le joueur tel que des arrêts sur image (appelé *freeze)* ou un ralentissement du jeu (baisse du nombre d'image par seconde) car les méthode d'instanciation et de destruction ont des répercussions lourdes sur la mémoire.

2 - Intérêt du système de pooling

Avec le système de pooling, lorsqu'on a besoin d'un des objets en question, il n'y a plus besoin de l'instancier puisqu'il a déjà été créé au démarrage du jeu, Il suffit donc de l'activer dans la scène et l'utiliser. Lorsque l'on en a plus besoin, on le désactive pour le remettre dans le **pool** pour une réutilisation ultérieure au lieu de le détruire.

Premièrement, l'activation d'un objet ne fait qu'activer les différents composants déjà existant attachés à ce dernier. À l'inverse, une instanciation classique demande de créer tous ces nouveaux composants et d'allouer de la mémoire pour stocker les informations de ces derniers, ce qui demande plus de ressources.

Deuxièmement, le processus de récupération de mémoire est souvent déclenché lorsque l'on détruit un objet dans Unity. Un objet est un ensemble de variables et lorsque l'objet est détruit, le GC libère la mémoire utilisée par les composants de l'objet.

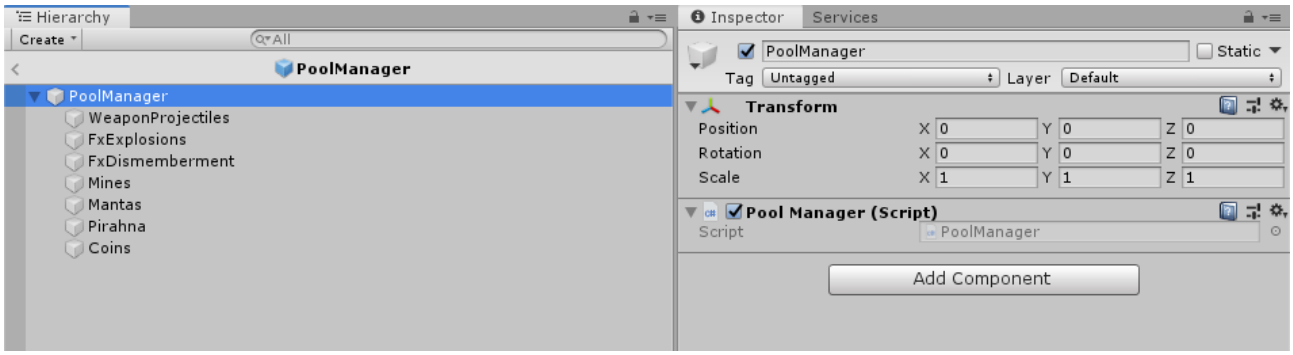
Or avec le système de pooling, de la mémoire est allouée lorsqu'on instancie les objets au démarrage du jeu et ne sont jamais détruit : ils sont simplement désactivés puis réutilisés si besoin. Par conséquent le GC n'est pas appelé pour ces objets, ce qui économise des ressources de façon considérable si on en manipule un grand nombre (Ex : les projectiles tirés par le joueur).

3 - Implémentation

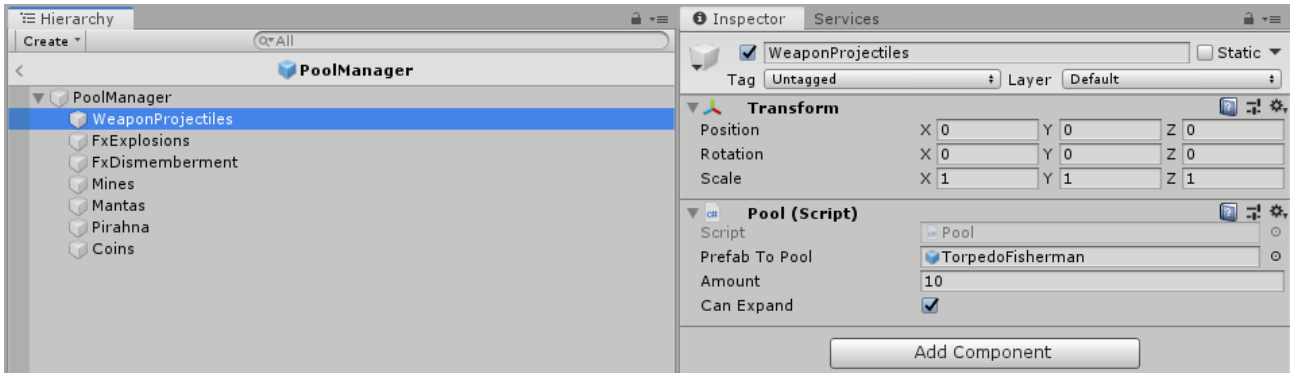
Le pooling est utilisé dans les projets suivants :

- Aquabyss, pour les ennemis, les effets graphiques (vfx) et les projectiles du sous-marin ;
- SlashingMachine, pour les ennemis et les vfx.

Voici comment le système a été implémenté.



Nous avons un objet PoolManager responsable de la gestion des objets dit *poolable*, c'est-à-dire les objets fréquemment utilisés et gérés dans le système de *pooling*. Le PoolManager contient des objets portant le script *Pool*.

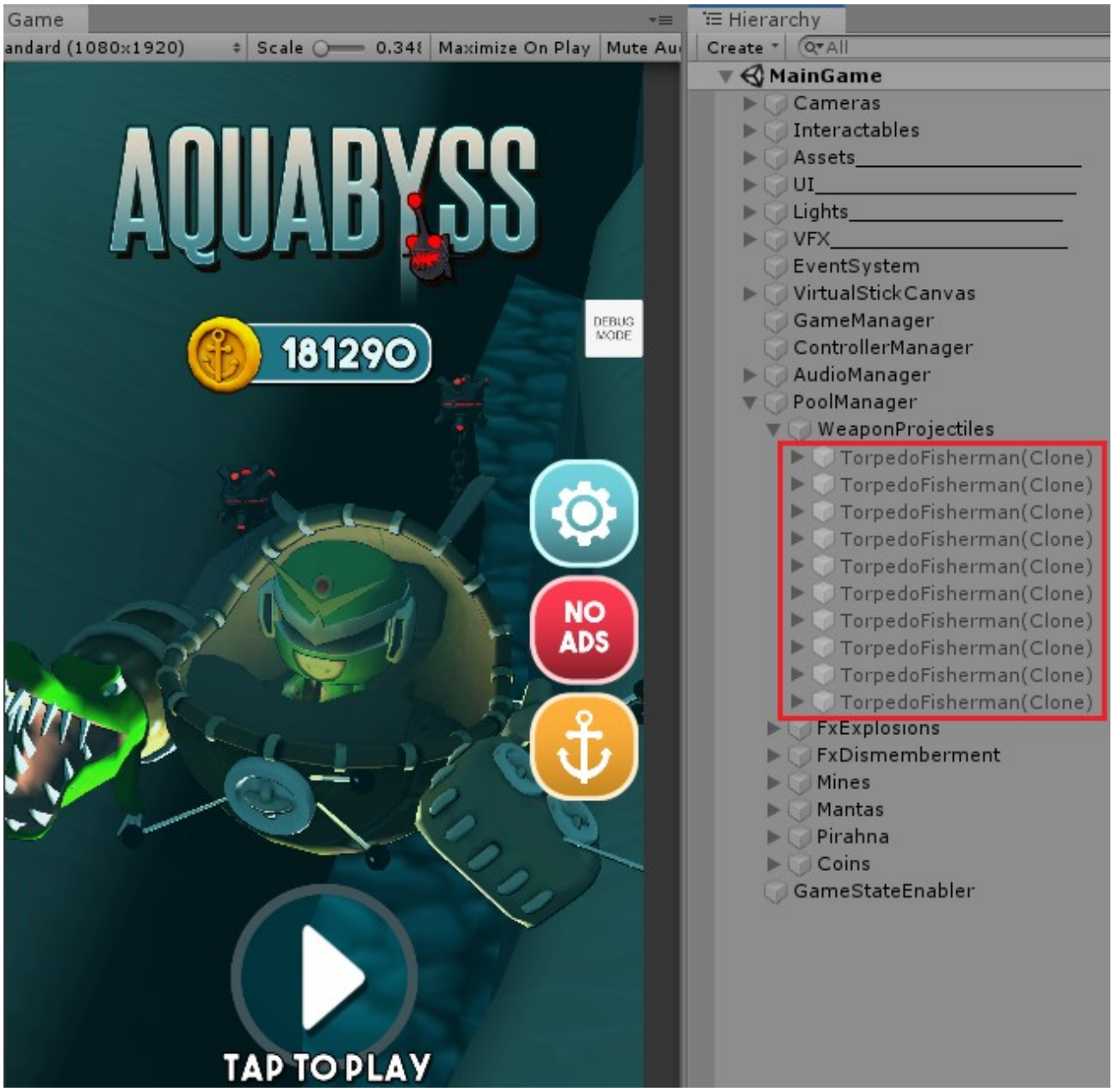


Les objets contiendront tous les objets *poolable* qui leurs correspond. Dans la capture d'écran du dessus, surligné en bleu, l'objet WeaponProjectiles tient le script Pool. Il prend en paramètre le prefab de l'objet à instancier au lancement de l'application ainsi que le nombre d'instances qui sont renseignés dans les variables nommées respectivement *PrefabToPool* et *Amount*. Le PoolManager est accessible par tous les objets extérieurs et ces derniers peuvent demander au PoolManager d'exécuter la méthode d'activation d'un objet

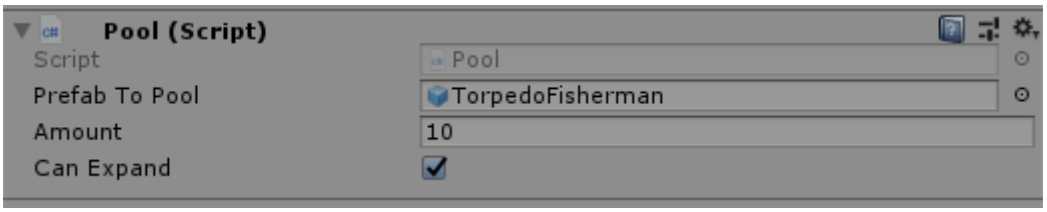
poolable à une position donnée en paramètre. Dans *Aquabyss*, l'objet responsable de la gestion des ennemis peut demander au PoolManager d'activer des Mines, des Mantas ou des Piranhas à une position donnée. De la même manière, l'objet responsable de faire apparaître des vfx peut demander au PoolManager d'activer un effet d'explosion une position donnée etc.

Une fois que ces objets remplissent leurs fonctions, il faut les désactiver, les retourner au *pool* pour qu'ils puissent être utilisés à nouveau. Des conditions de retour au *Pool*/désactivation sont définies pour chacun de ces objets. Les objets *poolable* implémentent une **interface** *IPoolable* les forçant à définir une méthode de retour au *pool* au sein de leurs classes. Ces méthodes sont ensuite appelées lorsque certaines conditions sont remplies, conditions propre à chaque objet.

Par exemple, un vfx d'explosion implémente l'interface *IPoolable* et retournera au *pool* lorsqu'il aura fini son animation. De la même manière, un ennemi retournera au *pool* lorsqu'il aura été détruit par un projectile du joueur, un projectile retourne au *pool* lorsqu'il a détruit un ennemi ou quand il est hors caméra, etc.



Sur la capture d'écran ci dessus, dix torpilles sont instanciées et désactivées au lancement du jeu. Nous sommes sur le menu principal et les torpilles existent déjà mais sont désactivés (objets grisés dans la fenêtre « Hierarchy »), car nous en avons pas encore d'utilisation. Ces derniers seront activés et lancés lorsque le joueur tirera une torpille : cette dernière sera téléportée à l'endroit du tir avec la bonne rotation en fonction de la position du joueur. Si la torpille touche un ennemi, ou a dépassé sa durée de vie, la torpille se désactive et est à nouveau disponible pour un prochain tir.



Le booléen CanExpand du script Pool permet au système de créer des nouvelles instances de l'objet *poolable* lorsque toutes les instances de cet objet sont déjà actives sur la scène. Prenons un exemple avec les torpilles. Si le joueur a déjà tiré dix torpilles mais qu'aucune d'entre elles n'est disponible, le joueur est à court de munitions. Si le joueur tire une autre torpille, en ayant le booléen CanExpand à « vrai », le système permettra de créer une nouvelle instance de l'objet et ainsi agrandir son *pool*. Dans le cas inverse, aucun objet n'est instancié.

On définit donc en amont, le nombre d'instances qu'on estime avoir besoin et si cette estimation est sous-estimée, on laisse la possibilité d'étendre le *pool* de l'objet demandé. Il n'est pas nécessaire d'instancier des centaines d'objets dont on utilisera qu'une fraction, car cela occupe de la mémoire inutilement. Certes, de cette manière, il ne sera plus nécessaire d'étendre à l'exécution les *pools* d'objets mais cela se fera au détriment du temps de chargement du jeu.

Le coût processeur lié à l'allocation de mémoire lors de l'instanciation des objets *poolable* est amoindri, car au lieu d'instancier un objet à chaque fois où il est nécessaire, on demande simplement au PoolManager d'en activer un. Le nombre total d'instances est limité au nombre total d'objets possiblement présent sur la scène.

Le nombre total d'instanciations à faire dans le système de pooling dépend du paramétrage du jeu, comme la durée de vie d'une torpille et la vitesse de tir du joueur. Par exemple, le joueur tire une torpille toutes les secondes, et chaque torpille a une durée de vie de trois secondes. Le jeu permet donc d'avoir une situation où le nombre maximal de torpilles présentes sur la scène en même temps est de trois.

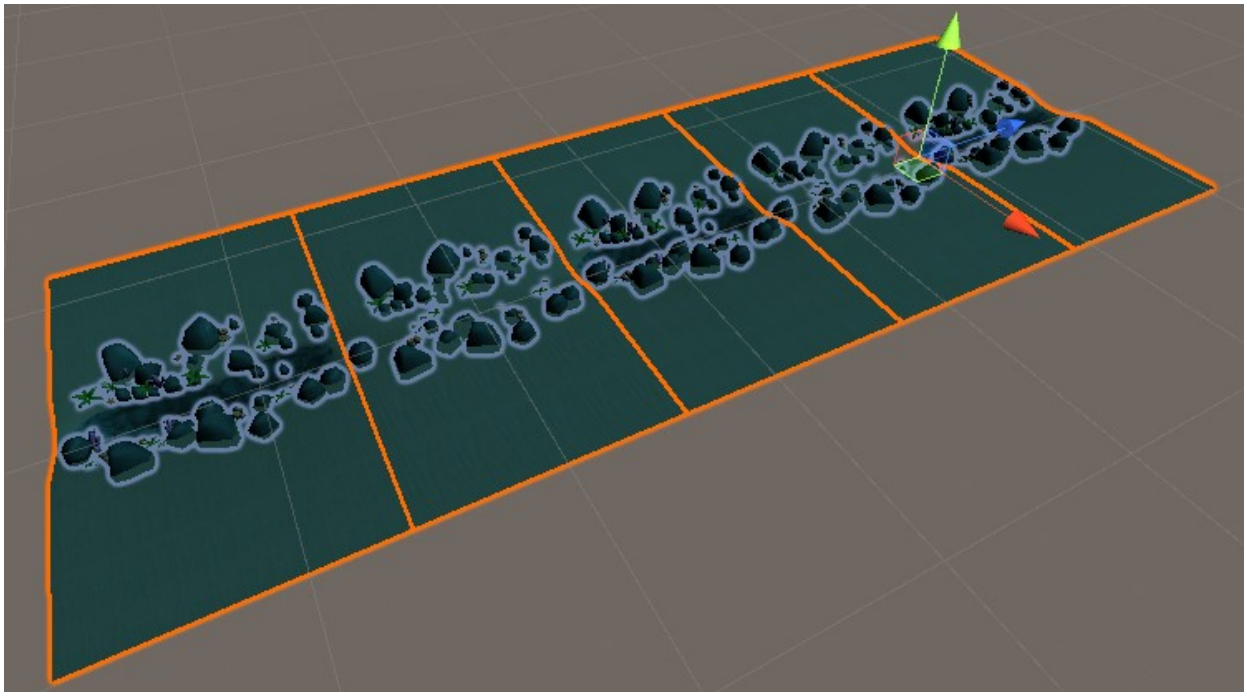
4 - Conclusion

Pour conclure sur le système de pooling, les objets les plus fréquemment utilisés sont gérés par le système de *pooling*. Ils sont activés à divers moment du jeu et se désactivent quand certaines conditions sont remplies. Avec ce système, on évite la sollicitation du GC (destruction des objets) et on évite l'allocation de mémoire superflue lié aux instanciations répétées, économisant ainsi beaucoup de ressources. Cependant, les instanciations sont faites au démarrage du jeu, ce qui augmente le temps de chargement au lancement de l'application. Mais comparé au gain de performances, le temps de chargement supplémentaire au démarrage du jeu est acceptable étant donné que le but est de ne pas détériorer l'expérience du joueur pendant les phases de gameplay.

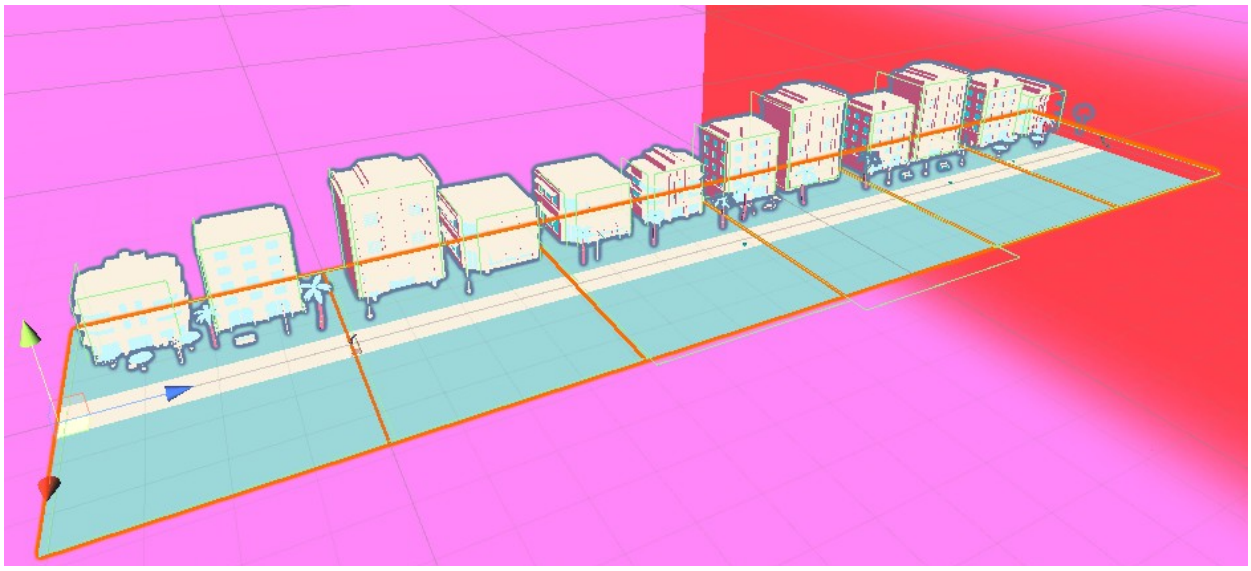
D - Réutilisation de décors

Les jeux Aquabyss et Slashine Machine sont des jeux de type **runner** : le joueur se déplace en permanence en rencontrant des blocs de niveau qui bouclent. Dans ces deux jeux, nous laissons le joueur à la même position et nous faisons défiler le décor pour donner cette illusion de progression dans le niveau. Pour le défilement de niveau à l'infini, nous avons conçu des briques de niveau pouvant se répéter. Au lieu d'instancier des nouveaux blocs de niveau au fur et à mesure que le joueur avance, on remet simplement les blocs qui passent hors camera au début du niveau. Début du niveau qui est caché par un brouillard ou tout simplement hors caméra pour éviter que le joueur puisse voir un morceau de décor apparaître soudainement à l'horizon. Cette technique permet d'économiser des ressources de la même manière que le système de pooling. On évite d'instancier les morceaux de niveaux à venir et on évite de détruire les blocs de niveaux qui passe hors camera derrière le joueur.

Ci-dessous le niveau pour *Aquabyss* :



Ci-dessous, le niveau pour SlashingMachine (avec le fondu visible à droite) :

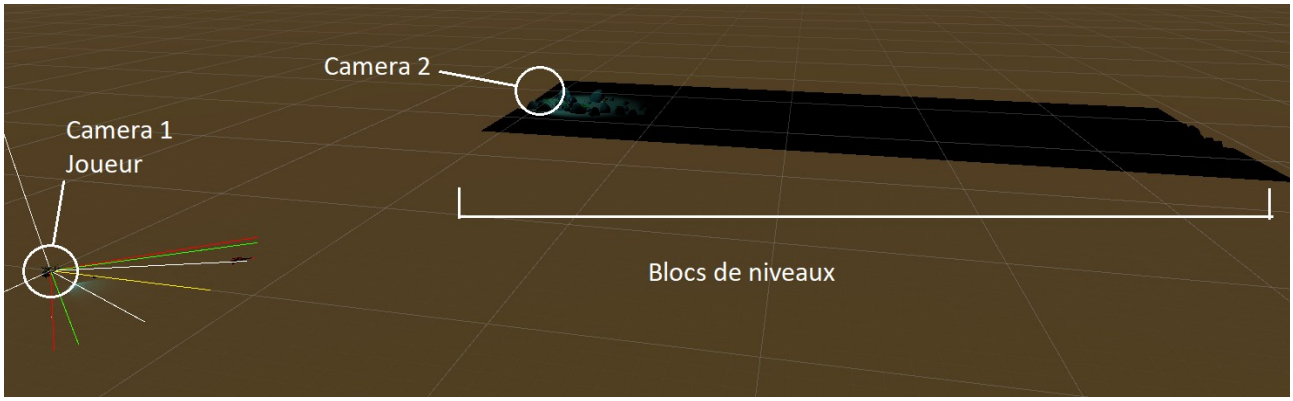


Mais cette solution n'était pas suffisante pour le jeu Aquabyss. Nous avions une baisse significative d'images par seconde. Contrairement à Slashing Machine, le sous-marin avait une source de lumière attaché pour éclairer le fond marin et le moteur d'Unity devait gérer le rendu des textures du niveau en temps réel.

Pour économiser des ressources, le Lead Programmer a rendu les blocs du niveau **statique**. Un objet statique permet de gagner en performances en indiquant au moteur de faire des pré-calculs concernant les rendus de textures avec les lumières, mais ne pourra plus se déplacer dans l'espace. Cette méthode est appelée **static batching**.

Nous n'avions plus le défilement de niveaux avec cette méthode, alors pour redonner cette illusion de mouvement, une nouvelle camera a été rajoutée. Celle-ci se déplace en passant par dessus tous les blocs de niveaux. Une fois arrivée au bout, la caméra est téléportée au début du niveau et continue d'avancer. Le rendu de cette camera est superposée à la camera principale, ce qui permet d'avoir le rendu final avec l'illusion qu'un sous-marin avance dans le niveau.

Dans la capture d'écran suivante, nous avons la première camera ainsi que le joueur à la même position, et nous avons la seconde camera qui défile sur tous les blocs de niveaux.



Le rendu des cameras est ensuite superposé et nous obtenons le rendu voulu d'un sous-marin avançant dans un niveau infini.

Conclusion

Ce stage m'a permis d'acquérir de l'expérience dans le développement de jeux dans une petite équipe et sur un environnement mobile. De plus, j'ai apprécié l'encadrement que j'ai reçu, ce qui m'a permis de progresser tant en programmation que dans l'utilisation du moteur de jeu Unity.

Ce stage m'a permis de mettre en application les connaissances que j'ai acquises lors de mon cursus et m'a également permis d'en acquérir de nouvelles telles que les principes SOLID, qui ont changé ma façon de concevoir et d'architecturer le code.

Avec le temps, on m'a donné de plus en plus d'autonomie au fur et à mesure que je progressais et que je m'adaptais aux conditions de travail en équipe. J'ai dû apprendre à me conformer aux normes de l'équipe en tant que développeur gameplay et aux conventions de codage donné par le Lead programmer.

Côtoyer au quotidien des personnes issues du milieu du jeux vidéo m'a permis d'approfondir mes connaissances du média en général, tant en développement de jeux que dans l'industrie et l'actualité du jeux vidéo.

Annexes

Principes SOLID

Le S correspond à "Single Responsibility Principle" (SRP) qui se traduit par une seule responsabilité. Ce principe implique qu'une classe ne doit avoir qu'une seule responsabilité, qu'un seul type de tâche à effectuer. Si deux tâches sont considérées comme différentes, on les sépare en classes différentes. Par exemple une classe responsable du déplacement du joueur et une classe responsable de l'état du joueur (points de vie, points d'énergie, etc.).

Le O correspond à "Open / Close Principle" (OCP) qui se traduit par Ouvrir / Fermer. La raison de ce principe est très simple, une classe doit être ouverte à une extension/mise à jour, mais doit-être fermée pour toutes modifications. La classe doit être ouverte aux extensions/ mise à jour uniquement si cela ne rajoute pas de nouvelles fonctionnalités au sein de la classe. La raison principale de ce principe est d'éviter les fortes dépendances entre les différentes classes. Il peut y avoir des cas où la modification est indispensable comme la résolution d'un bug. Mais ces modifications ne doivent pas introduire de nouvelles fonctionnalités.

Le L correspond à "Liskov Substitution Principle" (LSP) qui se traduit par substitution de Liskov. Si un morceau de code utilise un objet Parent, alors le code ne présentera aucun problème si on replace un Parent par un Enfant, où l'objet Enfant hérite de l'objet Parent. L'application de ce principe permet de renforcer le Single Responsibility Principle, de réduire les dépendances inutiles et les duplications. À noter que décomposer les responsabilités a tendance à augmenter le nombre de classes/interfaces isolées et peut être complexe à mettre en place, mais ce principe reste bénéfique au projet par une structure plus propre et compréhensible.
L'exemple concret le plus souvent utilisé pour évoquer ce principe est le rectangle avec le carré. Si la classe Carré hérite de la classe Rectangle (car un carré est un rectangle particulier), nous aurons un problème si on doit renvoyer l'aire d'un tel objet car l'aire d'un rectangle ne se calcule pas de la même façon qu'un carré. Un solution est alors de faire hériter Carré et Rectangle d'une classe abstraite Shape.

Le I correspond à "Interface segregation Principle" (ISP) qui se traduit par ségrégation des interfaces. L'ISP consiste à trouver les vraies dépendances et à éliminer celles non désirées, permettant ainsi de découpler grandement le code, de renforcer la bonne encapsulation et la sécurité du code. Ce principe impose d'identifier les ressources nécessaires à un morceau de code pour qu'il puisse réaliser les tâches qui lui sont confiées, afin de fournir uniquement les ressources pour remplir ces tâches, et rien de plus. Cela facilite beaucoup de choses comme les tests unitaires, une meilleure décomposition du code et donc, une meilleure compréhension générale.

Le D correspond à "Dependency Inversion Principle" (DIP) qui se traduit par inversion des dépendances. Aucun module de haut-niveau ne doit dépendre de modules de plus bas niveau, mais les deux devraient au contraire dépendre d'abstractions. Les abstractions ne devraient pas dépendre des détails, ce sont les détails qui doivent dépendre des abstractions. Le DIP permet de réduire les dépendances entre les classes et les implémentations concrètes. Il permet aussi de faciliter les tests unitaires et surtout, de pouvoir se resservir facilement des classes crées.
Concrètement, si vous avez un projet avec du code métier (Business Layer - BL) et un projet qui s'occupe de la persistance de vos données (Data Access Layer - DAL), généralement votre projet BL référence votre projet DAL. Avec ce principe, c'est l'inverse qui va se produire, on ne fait plus référence à DAL dans le projet BL. On crée une abstraction et c'est à partir de là qu'intervient ce que l'on appelle une injection de dépendance. Ceci à des avantages à plus d'un titre : gérer les objets plus efficacement, remplacer le DAL par une autre (changement d'une base de données), etc.

Glossaire

Asset : Un asset, dans Unity, désigne tout ce qui compose un projet, que ce soit des fichiers de code, d'images, de modèles 3D, d'animations, etc.

Combo : Dans les jeux vidéo, un combo est un enchaînement d'actions exécutées les unes après les autres. Ce terme est utilisé pour définir une combinaison d'actions ou pour comptabiliser le nombre de fois qu'une action est réalisée.

Developement Hell : « L'enfer du développement », en français est un jargon de l'industrie des médias, pour un film, un jeu vidéo ou un album qui reste en développement pendant une période particulièrement longue. Les projets en développement ne sont pas officiellement annulés, mais les progrès sont lents, ou souvent modifiés.

Endless : Les jeux « endless » n'ont pas de notions de complétion de niveau. Le joueur évolue au sein du même niveau et tente d'atteindre un certain objectif qui peut varier selon le type de jeu. Par exemple, tuer le plus d'ennemi, marquer le plus de points ou récolter le plus d'objets.

Flottant (nombre) : Nombre à virgule étant une approximation d'un nombre réel, car la partie décimale ne tient que sur un nombre fini de bits.

Garbage Collector : Un ramasse-miettes, ou récupérateur de mémoire (en anglais garbage collector, abrégé en GC), est un sous-système informatique de gestion automatique de la mémoire. Il est responsable du recyclage de la mémoire préalablement allouée puis inutilisée. La récupération de mémoire par le Garbage Collector est automatique et invisible pour le programmeur, mais le processus de collecte nécessite en réalité un temps CPU considérable en arrière-plan. Lorsque le GC est utilisée correctement, la gestion automatique de la mémoire est généralement égale ou supérieure à l'allocation manuelle en termes de performances globales. Cependant, il est important que le programmeur évite les erreurs qui déclenchent le collecteur plus souvent que nécessaire et introduisent des pauses dans l'exécution du programme en générale. Un GC de style « stop the world » va bloquer entièrement le programme tant que la mémoire n'a pas été libérée (pouvant se traduire par des freezes).

Input : En informatique, le processeur reçoit des informations provenant des périphériques qui lui sont associés. Par exemple, les informations du déplacement ou d'un clic de souris. Dans le cadre des smartphones, il s'agit des boutons de l'appareil et des mouvements d'un ou plusieurs doigts sur l'écran tactile.

Interface (programmation) : Une interface est un modèle de classe. Elle contient des méthodes et des propriétés, avec la particularité qu'elle ne fournit pas l'implémentation des méthodes. Cela veut dire qu'on décrit juste les méthodes sans décrire ce qu'elles font.
Une interface ne sert à rien toute seule : chaque interface est vouée à être implémentée par une classe (ou une structure) qui devra implémenter ses méthodes et posséder ses propriétés. Le but d'une interface est de définir les méthodes et propriétés proposées par toute classe (ou structure) qui l'implémente, sans que le développeur ait besoin de savoir comment celles-ci sont codées.

Joystick : Un joystick se compose d'une base et d'un manche qui peut être déplacé dans n'importe quelle direction. C'est un dispositif couramment utilisé dans les jeux vidéo pour contrôler un personnage.

Joystick virtuel : Simulation du comportement d'un joystick réel sur une interface telle qu'un écran tactile. Celui-ci peut se trouver à différente position de l'écran selon la façon d'interagir avec le dispositif.

Mémoire vive (RAM) : La mémoire vive est un type de mémoire équipant tout ordinateur et qui permet de stocker des informations provisoires.
Les opérations de lecture et d'écriture en mémoire vive s'effectuent bien plus rapidement que sur les autres types de stockage d'un ordinateur, tels que les disques durs, les clés USB et CD.

Pipeline de production : Un pipeline de production est une série de produits en cours de développement, de préparation ou de production, développés une entreprise et à différentes étapes de leur cycle de vie.

Prefab : Un Prefab (préfabriqué) est un modèle / patron / schéma de GameObject. Ce n'est pas une instance de ce GameObject mais sa représentation. Le système de Prefab de Unity permet de créer, configurer et stocker un GameObject complet avec tous ses composants, ses valeurs de propriété et ses GameObjects enfants comme un asset réutilisable. Le Prefab agit comme un modèle à partir duquel vous pouvez créer de nouvelles instances de ce dernier dans la scène.

Scène : Dans Unity, les scènes contiennent les environnements et les menus de votre jeu. Dans chaque scène, vous placez vos environnements, obstacles et décorations, essentiellement en concevant et en construisant votre jeu par briques.

ScriptableObjects : Un *ScriptableObject* est un conteneur qui est utilisé pour sauvegarder de grandes quantités de données, indépendamment des instances de classe. L'un des principaux cas d'utilisation de ScriptableObjects consiste à réduire l'utilisation de la mémoire d'un projet en évitant les copies de valeurs. Ceci est utile si votre projet a un prefab qui stocke les données immuables dans les scripts attachés à des objets.

Singleton : En génie logiciel, le singleton est un patron de conception (design pattern) dont l'objectif est de restreindre l'instanciation d'une classe à un seul objet. Il est utilisé lorsqu'on a besoin exactement d'un objet pour coordonner des opérations dans un système. Par exemple, si la classe GameManager implémente un singleton et que cette classe possède l'information de l'état du jeu, tout objet externe peut accéder à cette information.

Sprite : Un sprite, est dans le jeu vidéo un élément graphique en deux dimensions. L'animation est possible en faisant défiler d'autres sprites les uns après les autres.

Static batching : Il s'agit d'une technique permettant de combiner des objets statiques (qui ne se déplacent pas) et de calculer les rendus plus rapidement en réduisant les appels de rendu pour les géométries.

Statique (objet Unity) : Un objet statique est un objet considéré de manière spéciale par le moteur : il ne pourra pas bouger, et de nombreuses techniques d'optimisation se baseront sur cet état pour fonctionner. Par exemple, le rendu est optimisé car tous les calculs liés à la lumière seront pré-calculés.

Synchrone : Dans un programme, une exécution synchrone signifie que chaque instruction est exécutée l'une après l'autre, contrairement à l'exécution en parallèle dans le cas inverse.
Vis à vis des événements, l'exécution se fera de manière synchrone sur toutes les méthodes écoutant cet événement, avant de reprendre les instructions suivant son déclenchement.

Temps processeur : Temps durant lequel le processeur est sollicité pour réaliser une tâche.

Vecteur : Dans un espace en trois dimensions, un vecteur est un ensemble de trois valeurs pouvant décrire la position d'un point dans l'espace ou une direction.

Vecteur directeur : Un vecteur directeur est un vecteur contenant trois valeurs permettant de décrire une direction donnée dans un espace en trois dimensions.

Sources

Définition, contexte et historique du jeu hyper casual
Définition des principes SOLID
Présentation JLA
Définition Garbage Collector
Définition Prefab
Définition ScriptableObject
Static Batching