Documentation de développement Unity du WarioWare

Version 0.2 – 23 novembre 2019

Retour au sommaire

“The most important property of a program is whether it accomplishes the intention of its user.”

- Charles Antony Richard Hoare

Le développement du WarioWare diffère de l’organisation habituelle des projets sur Unity du fait de sa forme : un jeu Macro constitué d’une multitude de micro-jeux. De cette situation exceptionnelle naît une organisation de travail exceptionnelle, nécessitant une perméabilité dans le développement des micro-jeux vis-à-vis du Jeu Macro.

L’équipe en charge du développement du Jeu Macro, les Mousquetaires, s’est donc chargée de réfléchir à la rigueur à suivre dans le développement des micro-jeux, aux fonctionnalités communes, et à l’échange d’informations entre le Jeu Macro et les micro-jeux.

L’objet de ce document est précisément l’explication de ce cadre de travail mis en place. Ce cadre est pensé pour être accessible à tout le monde, et prend en compte plusieurs méthodes de développement possibles. Il impose aussi une certaine rigueur à respecter de la part des développeurs-micro, sans laquelle l’intégration du travail de chacun sera plus laborieuse et chronophage.

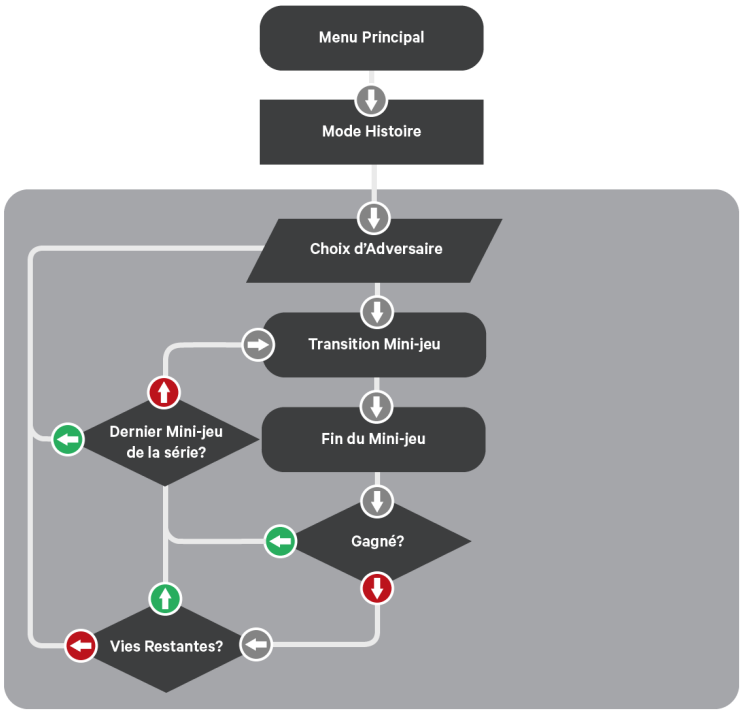
Le document se veut clair et fonctionnel. Il est important que chaque développeur-micro en prenne connaissance et fasse remonter toute incompréhension ou besoin de précision. Il est divisé en cinq parties comme il suit :

1. *Vue d’ensemble*, qui aborde de manière générale la structure du jeu Macro et la logique interne aux micro-jeux ;
2. *Détail des systèmes*, qui explique plus en profondeur les outils mis à disposition des développeurs-micro ;
3. *Contraintes de développement*, qui donne la rigueur à suivre dans le développement et l’organisation ;
4. *Exportation des micros-jeux*, qui détaille la procédure à suivre pour garantir une exportation efficace ;
5. *Annexes*, qui liste les termes complexes et les exemples donnés dans le document.

Ce document, c’est le Saint Graal des développeurs-micro. En travaillant sur les projets, il est nécessaire de s’assurer de la conformité des systèmes développés avec le contenu de ce document.

I. Vue d’ensemble

A. Boucle du programme



La boucle du programme consiste en une suite simple d'actions permettant de jouer tous les micros-jeux. Cette boucle fait en sorte de respecter le principe suivant :

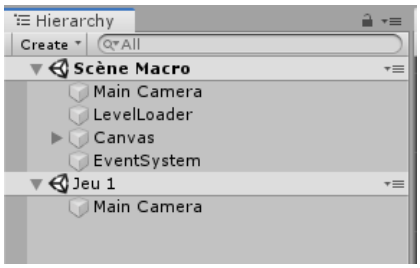
1. Fonctionnement jeu Macro.
2. Contrôle donné au micro-jeu.
3. Fonctionnement micro-jeu.
4. Contrôle donné au jeu Macro.
5. Répétition.

Ainsi, il est important pour le micro-jeu de respecter certaines règles pour être sûr que le jeu Macro puisses reprendre la main et passer au prochain micro-jeu. N'importe quel développeur-micro doit absolument respecter les deux instructions précisées ci-après.

---

1. Appel de `StartGame()`

Durant l'animation de transition permettant de passer d'un micro-jeu à l'autre, la scène du prochain micro-jeu est chargée en additif, c'est à dire que son contenu est ajouté à la scène actuelle du jeu Macro



Une fois le chargement de la scène fini, les fonctions `Awake` et `Start` de tout `MonoBehaviour` sont appelées. À partir de ce moment, c'est au développeur-micro d'appeler la fonction `static StartGame` du type `Macro` de la façon suivante, pour être sûr que le micro-jeu puisse commencer :

- `Macro.StartGame()`

Une fois cette fonction appelée, le gameplay du jeu pourra commencer à partir du moment où l'événement `OnGameStart` est appelé.

2. Appel de `EndGame()`

Enfin, le développeur-micro doit absolument appeler la fonction `static EndGame` du type `Macro` de la façon suivante, une fois que la transition pour charger le prochain micro-jeu peut être faite :

- `Macro.EndGame()`

Si cette fonction n'est pas appelée, la boucle du programme ne peut pas se répéter et le jeu Macro ne peut jamais passer au micro-jeu suivant.

B. Séquence de micro-jeu

La séquence de micro-jeu est la partie où le jeu Macro laisse la main au développeur-micro et le laisse opérer pour que son gameplay se déroule selon ses termes. Au niveau du micro-jeu, on retrouve le déroulement représenté ci-contre.

---

1. L'événement `Start()`

À la fin du chargement de la Scène, le jeu ne s'affiche pas directement. Dans un premier temps, il faut exécuter une première génération du micro-jeu. *(Ex : Créer un niveau de manière procédurale, préparer l'affichage du jeu, etc…)*

De plus, cette première génération est lancée avec la fonction **`Start`** de Unity. Ainsi toute logique de génération ou de préparation de la scène d'un micro-jeu doit être appelée depuis cet événement.

---

2. La Fonction `StartGame()`

La fonction **`StartGame`** doit être appelée par le développeur-micro à partir du moment où il souhaite que son micro-jeu puisses commencer. Cela permettra à l'évènement **`OnGameStart`** d'être appelé au prochain battement.

---

3. L'événement `OnGameStart()`

Il s'agit de l'événement concrétisant le lancement du micro-jeu mais pas de son timer. C'est au développeur de définir si à ce moment il veut lancer son gameplay immédiatement, jouer une animation d'intro ou autre.

---

4. Déroulement du Micro-Jeu

Un fois que l'événement **`OnGameStart`** est appelé, le micro-jeu est lancé et sa logique devient prioritaire. Plusieurs fonctions et propriétés sont à la disposition des développeurs-micro pour aider à la création des micros-jeux tout en laissant le plus de libertés possibles. Chacun est donc libre d'utiliser ces fonctions comme il le souhaite. Cependant, il faut tout de même respecter certaines lignes de conduites.

a. Affichage du verbe d'action :

Permet d'afficher un verbe d'action avec le système d'UI macro pendant un temps définit.

b. Lancement du timer

Permet de lancer le timer du micro-jeu qui sera affiché sur l'UI Macro. Il est possible d'exécuter du code à la fin du timer et de récupérer le temps restant.

c. Déroulement du Micro Jeu

Exécution de toute la logique donnée par le créateur-micro et accès à une variété de callbacks, de fonctions et de propriétés. *(Ex : Le callback OnBeat qui permet de lancer du code à chaque battement)*

d. Victoire/Défaite du Joueur

Fonctionnalité permettant de savoir si le joueur a perdu ou gagné.

---

5. La fonction `EndGame()`

La fonction **`EndGame`** doit être appelée à la fin du micro-jeu. C'est au développeur de l'appeler en cas de victoire ou défaite du joueur.

Si la condition de défaite du micro-jeu est basée sur la fin du timer, le développeur peut appeler sa logique depuis l'événement **`OnTimerEnd`** qui exécutera son contenu au moment où le timer atteint une valeur de **`0`**.

II. Détail des systèmes

A. Fonctionnalités Macro

La majorité des interactions Micro-Macro se font par le biais d'un seul script avec des propriétés et des méthodes **static**. Ainsi toutes communications avec le système du jeu Macro se fait selon la manière suivante :

- **Macro.BPM** → Accès à une propriété. *(Dans ce cas la valeur du BPM)*
- **Macro.Win()** → Appel d'une fonction. *(Dans ce cas l'appel de la fonction Win)*

Ci-dessous, une liste des différentes propriétés et méthodes que la classe **static Macro** offre aux développeurs micro.

---

1. Propriétés

BPM	Valeur du BPM actuelle.
TimeSinceLastBeat	Temps s'étant écoulé depuis le dernier battement.
TimeToNextBeat	Temps restant avant le prochain battement.
IsAzerty	Indique si le clavier est en AZERTY.

2. Fonctions publiques

StartGame	Permet le lancement de l'évènement OnGameStart.
Win	Annonce que le joueur a gagné le micro-jeu.
Lose	Annonce que le joueur a perdu le micro-jeu.
EndGame	Termine le micro-jeu actuel et lance l'animation de transition pour le prochain jeu.
StartTimer	Lance le timer du micro-jeu avec une certaine durée.
GetRemainingTime	Renvoie le temps restant avant la fin du timer.
Pause	Pause le jeu Macro.
GetData	Renvoie la donnée du micro-jeu actuel.
DisplayActionVerb	Affiche un verbe d'action pendant une certaine durée.
InsertMicroGame	Insert un micro-jeu dans la série actuel à la position indiqué.

3. Spécifications des fonctions

```
void StartGame()
```

Permet le lancement de l'événement `OnGameStart` au prochain battement. Il déclenche la fin de l'animation de transition et annonce le moment à partir duquel le gameplay du micro-jeu actuel peut commencer.

```
void Win()
```

Permet d'informer le jeu Macro que le joueur vient de gagner le micro-jeu actuel.

`void Lose()`

Permet d'informer le jeu Macro que le joueur vient de perdre le micro-jeu actuel.

`void EndGame()`

Permet de mettre fin au micro-jeu et de lancer le chargement du micro-jeu suivant. Cette fonction sera la dernière à être appelée dans la scène. Aussi, si une animation de fin est mise en place, **`EndGame`** doit être lancée à la fin de l'animation

---

`void StartTimer()`

Permet de lancer le timer du micro-jeu, ainsi que de l'afficher. Si le timer arrive à `0`, l'événement `OnTimerEnd` se lance.

Overloads :

`void StartTimer(float time, bool bpmAffect)`

Permet de lancer le timer du micro-jeu avec une valeur de temps spécifique en secondes. Si **`bpmAffect`** est vrai, le temps en seconde est affecté ou non par le **BPM**. Une durée affectée par le BPM passera plus vite si le **BPM** est plus grand.

`void StartTimer(int beatDuration)`

Permet de lancer le timer du micro-jeu avec une valeur de temps spécifique en nombre de battements *(Ex : Permet de lancer le timer pour une durée de 8 battements)*

`float GetRemainingTime()`

Permet d'obtenir le temps restant avant la fin du timer. Si le timer n'as pas encore été lancé, la valeur renvoyée sera de `1`. Si le timer est déjà terminé, la valeur renvoyée sera de `0`.

`void Pause()`

Permet de mettre tout le jeu en pause.

`MicroGameData GetData()`

Permet d'obtenir la donnée du micro-jeu actuel *(nombre de fois jouées, etc…)*

`void DisplayActionVerb()`

Permet d'afficher un verbe d'action selon le système d'UI du jeu Macro pendant une certaine durée.

Overloads :

`void DisplayActionVerb()`

Affiche le verbe d'action présent sur la `GameID` du micro-jeu actuel pendant `X` battements.

`void DisplayActionVerb(string verb)`

Affiche un verbe d'action spécifique pendant `X` battements.

`void DisplayActionVerb(string verb, int beatDuration)`

Affiche un verbe d'action spécifique pendant une durée spécifique en battements.

`void InsertMicroGame(GameID verb, int index)`

Permet d'insérer un micro-jeu spécifique à une position dans la série actuelle supérieur de celle du micro-jeu actuelle.

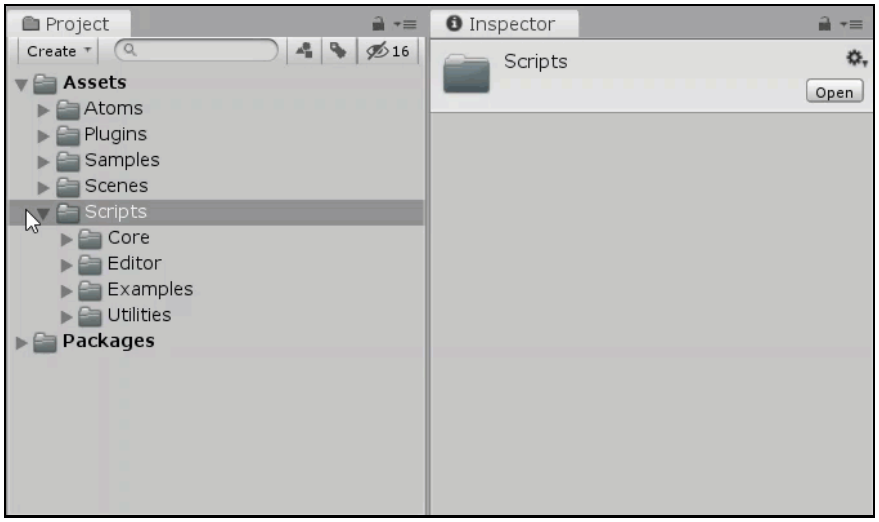
B. MicroMonoBehaviour

La **class** **MicroMonoBehaviour** a été créée pour offrir des fonctionnalités non-présentes dans la **class** **MonoBehaviour**. On compte comme ajout principal, l'addition de différents événements.

---

1. Création d'un script héritant de **MicroMonoBehaviour**

La méthode permettant de créer un script héritant de **MicroMonoBehaviour** est la même méthode utilisée pour créer un script héritant de **MonoBehaviour**. Il suffit d'effectuer un clic-droit dans la fenêtre ''Projet'' d'Unity, ceci va spécifier la destination du script créé, et de sélectionner Create puis Micro C# Script.




```
using System.Collections;
using System.Collections.Generic;

using UnityEngine;

namespace WarioWare.MyMicroGame
{
    public class MyMicroScript : MicroMonoBehaviour
    {
        // GameStart is called at the end of the transition to the current micro-game
        protected override void GameStart()
        {

        }

        // OnBeat is called per beat
        protected override void OnBeat()
        {

        }

        // OnActionVerbDisplayEnd is called after an action verb has been displayed
        protected override void OnActionVerbDisplayEnd()
        {

        }

        // OnTimerEnd is called when the timer for the current micro-game ends
        protected override void OnTimerEnd()
        {

        }

    }
}
```

2. Édition d'un MicroMonoBehaviour

Comme pour l'utilisation d'un script héritant de **MonoBehaviour**, un **MicroMonoBehaviour** possèdent différentes fonctions auxquelles du code peut être attaché. Cette logique sera appelée à certains moments selon la fonction. *(Ex : dans le cas de* **MonoBehaviour** *: Le code se trouvant dans la fonction* **Update** *est exécuté chaque frame)*

Pour un **MicroMonoBehaviour**, la fonction **OnBeat** est appelée à chaque battement.
Pour qu'une logique soit effectuée selon un certain événement, l'écriture ci-dessous est à respecter obligatoirement.

- `protected override void EventName() { }`

---

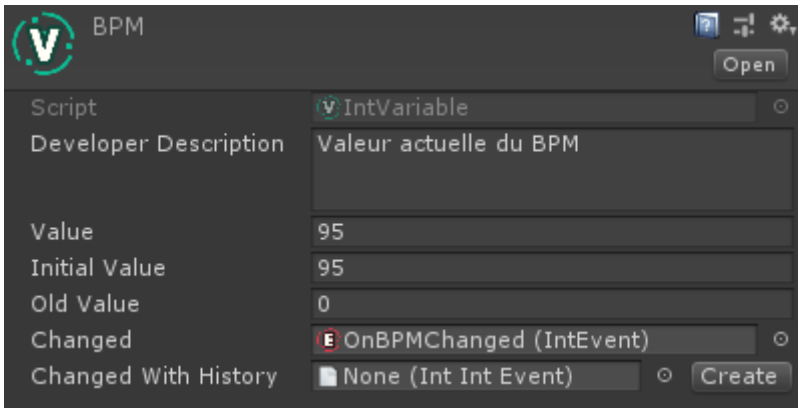
3. Événements

OnGameStart	Est appelé une fois l'animation de transition macro terminée. Appelle le lancement du micro- jeu actuel
OnBeat	Est appelé à chaque battement.
OnActionVerbDisplayEnd	Est appelé une fois que l'affichage d'un verbe d'action est terminé.
OnTimerEnd	Est appelé une fois que le timer lancé par le développeur-micro arrive à sa fin.

C. UnityAtoms

Le système de **UnityAtoms** est mis à disposition des développeurs. Il s'agit de fichiers de **ScriptableObject** contenant des **variables** utilisées par les scripts. Ce système permet d'utiliser des fonctionnalités Macro en ne passant que par l'inspecteur de Unity, et en évitant ainsi de s'embêter avec du code.

3. AtomVariable



a. Description

Les **AtomVariable** sont des **ScriptableObject** qui contiennent une unique valeur. On peut récupérer leur valeur pour l'utiliser de deux manières :

- **TypeVariable**, qui est la référence directe au **ScriptableObject**, que l'on peut déplacer directement.
- **TypeReference**, qui permet de choisir si l'on veut définir la valeur, mettre un **IntConstant** ou mettre un **IntVariable**.

"Type" est à remplacer par le type voulu.

Il est donc possible d'utiliser les **AtomVariable** avec les types de variables traditionnels, en voici quelques exemples :

- **String** → **StringVariable** ;
- **Float** → **FloatVariable** ;
- **Bool** → **BoolVariable**.

Dans l'exemple ci-dessus, le BPM est donc un **IntVariable**.

Vous ne devez jamais changer les valeurs d'un AtomVariable que l'on met à votre disposition. Une erreur se produira pour vous le rappeler !

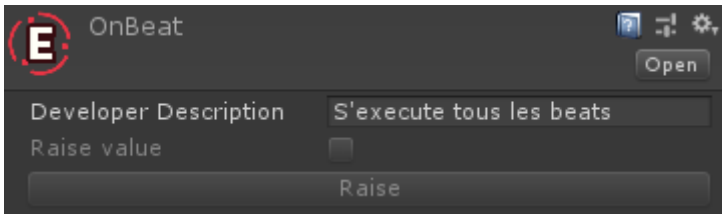
---

b. Propriétés

Les propriétés sont ce que l'on peut trouver dans un AtomVariable.

`DeveloperDescription`	Correspond aux notes laissées par les Mousquetaires pour savoir rapidement à quoi sert cet **`AtomVariable`**.
`Value`	Correspond à la valeur modifiée par le jeu Macro. **Ne la modifiez pas**.
`InitialValue`	Correspond à la valeur par défaut donnée à Value au lancement du jeu.
`OldValue`	Correspond à la valeur précédente de Value après changement. Elle sert principalement de valeur pratique.
`Changed`	**`AtomEvent`** executé à chaque changement de valeur, avec la Value passée en argument.
`Changed with history`	**`AtomEvent`** executé à chaque changement de valeur, avec la Value et l'Old Value passée en argument

4. AtomEvent



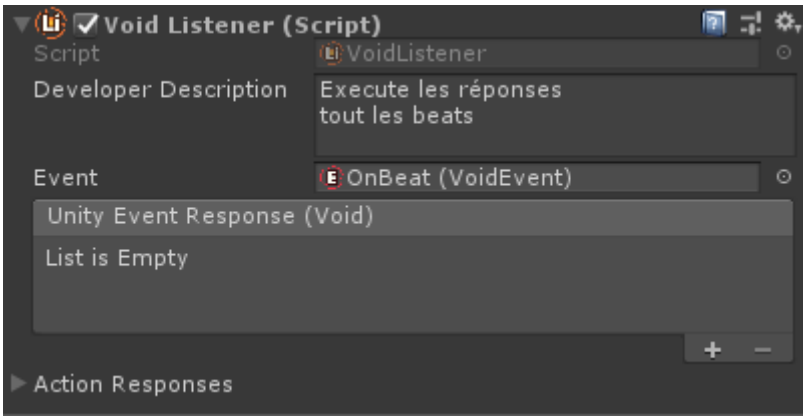
a. Description

Un **`AtomEvent`** est juste un **`ScriptableObject`** qui s'exécute grâce à notre système, tous les **`AtomListeners`** vont s'exécuter à chaque fois que **`OnBeat`** est exécuté.

b. Détails

Le bouton *Raise*, permet d'exécuter **l'**`AtomEvent` quand le jeu est lancé. Si le jeu n'est pas lancé, le bouton est grisé.

5. AtomListener



Un `AtomListener` s'enregistre automatiquement à un `AtomEvent`, si celui-ci est exécuté, alors il exécute le `UnityEvent` en dessous. Il permet de simplifier les étapes par rapport à l'utilisation directe d'un `AtomEvent`

---

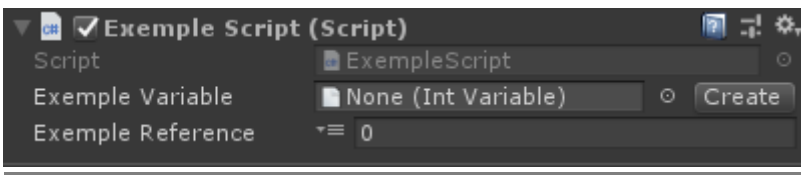
6. Application

a. Utilisation des `AtomVariable`

```
using UnityAtoms;
using UnityEngine;

namespace Game.NameOfMicroGame
{
    0 references
    public class ExempleScript : MonoBehaviour
    {
        [SerializeField] IntVariable exempleVariable = default;
        [SerializeField] IntReference exempleReference = default;

        0 references
        private void Start()
        {
            Debug.Log(exempleVariable.Value);
            Debug.Log(exempleReference.Value);
        }
    }
}
```



Dans cette situation :

Dans les deux cas, on peut récupérer la valeur avec **exempleVariable**.**Value** comme l'exemple ci-dessus.

Ajouter "using UnityAtoms" est nécessaire pour pouvoir écrire **`IntVariable`** et **`IntReference`**.

On ajoute ".**Value**" après le nom de la variable pour récupérer la valeur du **`ScriptableObject`** et l'utiliser selon vos besoins.

---

b. Utilisation des AtomEvent

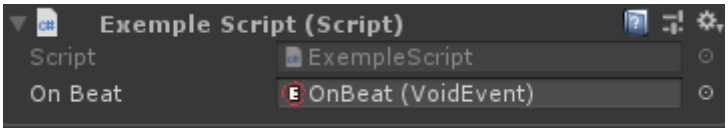
```
using UnityAtoms;
using UnityEngine;

namespace Game.NameOfMicroGame
{
    0 références
    public class ExempleScript : MonoBehaviour
    {
        [SerializeField] VoidEvent onBeat = default;

        0 références
        private void Awake() ⇒ onBeat.Register(OnBeat);

        0 références
        private void OnDestroy() ⇒ onBeat.Unregister(OnBeat);

        2 références
        private void OnBeat() ⇒ Debug.Log("Beat !! ");
    }
}
```



Les => sont des simplifications sur une ligne des {} classiques.

On a ici un exemple d'utilisation d'un AtomEvent :

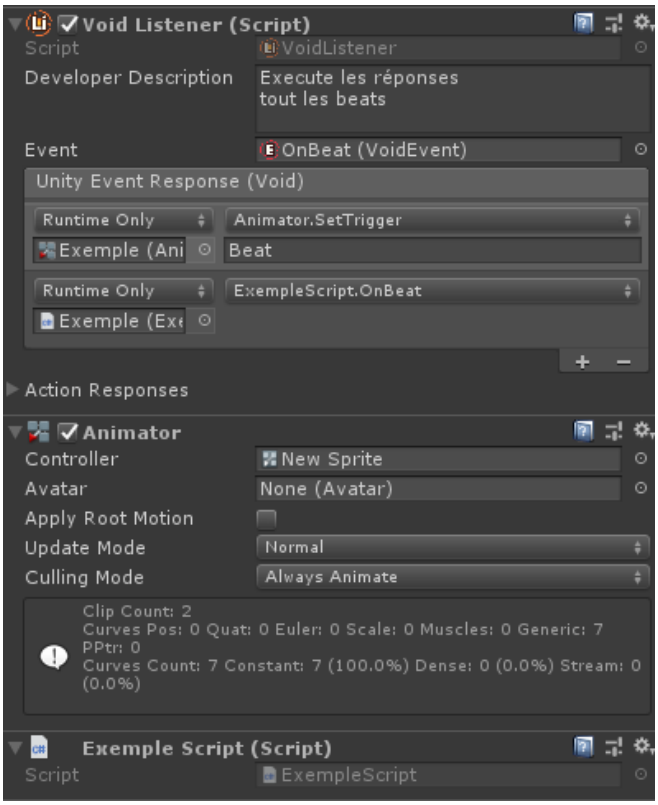
Awake() - On enregistre la méthode OnBeat() dans l'AtomEvent pour la lancer à chaque exécution de l'AtomEvent en question.

Destroy() - On décharge la méthode OnBeat() de l'AtomEvent pour éviter des bugs après la destruction de l'objet. C'est-à-dire on enlève la méthode OnBeat() de l'AtomEvent.

OnBeat() - Grâce à Awake() et à Destroy(), la méthode s'exécute au même moment que l'AtomEvent est exécuté.

c. Utilisation des AtomListener

```
0 references
public class ExempleScript : MonoBehaviour
{
    0 references
    public void OnBeat()
    {
        Debug.Log("Beat !! ");
    }
}
```



Pour rendre sa méthode visible au UnityEvent, il faut la rendre publique.

Le **UnityEvent** exécute le trigger de **l'Animator** et la méthode **OnBeat** du script d'exemple avec beaucoup moins d'étapes que l'enregistrement manuel de **l'AtomEvent** *(voir ci-dessus)* Les deux sont malgré tout laissés à disposition pour que chacun décide de sa manière de faire.

L'avantage de **l'AtomListener** est surtout d'être très accessible et rapide pour l'itération grâce à l'UnityEvent. Les développeurs-micros ont la possibilité d'utiliser les **AtomEvent** et faire ainsi les choses directement dans un script. Ou bien il est possible de passer par un **AtomListener** pour tout faire dans l'inspecteur de Unity.

III. Contraintes de développement

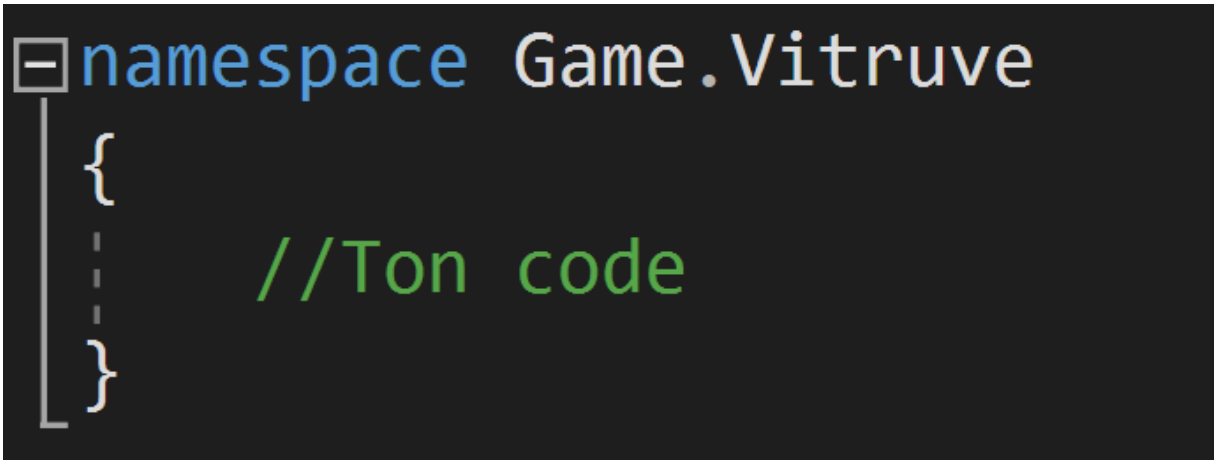
D.Obligation du **namespace**

1. Consigne

Il est obligatoire d'utiliser un **namespace** dans tous les scripts utilisés par le développeur-micro. Ce **namespace** est constitué comme il suit :

`Game.NomDuMicroJeu`

Exemple pour le micro-jeu intitulé « Vitruve » :



2. Nomenclature imposée

Ici la nomenclature à respecter pour le **namespace** se divise en deux parties :

`Game.NomDuMicrojeu`

La partie « **Game** » permet de rattacher l'intégralité des scripts à ce qui a été préparé par les Mousquetaires. Elle est commune à tout le monde.

La partie « **NomDuMicrojeu** » est à changer en fonction du micro-jeu développé et permet d'isoler les scripts de chaque développeur au moment de l'intégration au jeu Macro, évitant la génération de conflits. Elle est unique à chaque jeu.

E. Bannissement des **Tags**

1. Consigne

En aucun cas le micro-jeu développé doit utiliser le système de **Tags** de Unity.

2. Explications

L'utilisation de **Tags** peut faciliter des opérations notamment dans des scènes avec une multitude de GameObjects, mais il ne s'agit pas d'un système indispensable. En réalité il est facilement contournable.

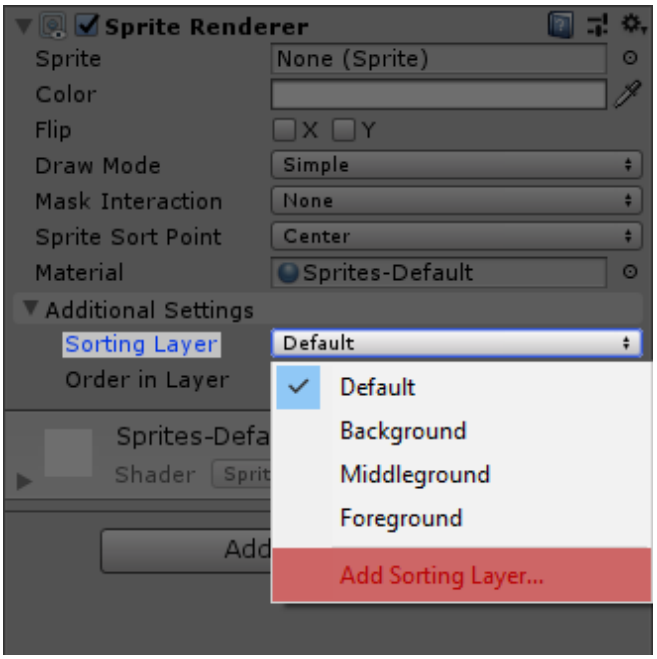
Le problème est surtout qu'il est générateur de conflits : si jamais les projets importés, à savoir les micro-jeux, utilisent des **Tags** qui ont été créés de leur côté, ces **Tags** ne seront pas trouvés, ni indexés de la même manière que dans le projet d'export.

Ce problème n'est pas présent pour les sept **Tags** d'origine de Unity, mais par souci de rigueur, il est demandé aux développeurs ne pas du tout utiliser le système de Tags.

F. Confinement des **Sorting Layers**

7. Consigne

Il est interdit d'ajouter de nouveaux **Sorting Layers** à son projet. En plus du **Sorting Layer** Default, trois ont été ajoutés au cas où les développeurs-micro souhaitent en utiliser en plus. Aucune règle n'est imposée quant à leur utilisation, il s'agit juste de ne pas en rajouter.



8. Explications

Le problème ici est similaire à celui des Tags. En ajoutant de nouveaux **Sorting Layers** sur un micro-jeu et pas sur l'ensemble de tous les projets, des conflits peuvent se créer.

G. `Macro.StartGame()` et `Macro.EndGame()`

9. Consigne

Le micro-jeu doit appeler les fonctions `Macro.GameStart()` et `Macro.EndGame()` une unique fois chacune.

10. Explications

Comme expliqué dans I. A. Boucle du programme, le jeu Macro a besoin de détecter à quel moment le micro-jeu commence – **GameStart** – et à quel moment il se finit – **EndGame** –, pour pouvoir prendre en charge la Scène comme il faut.

Sans ces fonctions, le jeu Macro ne pourra pas passer au micro-jeu suivant.

H. **Start()** et **OnGameStart()**

Comme expliqué dans **I. C. Séquence du jeu Macro**, Le micro-jeu différencie le **Start** d'un outil créé pour le projet, appelé **OnGameStart**, imposant la logique suivante :

- Le **Start** est utilisé pour tout ce qui est calculs initiaux, générations procédurales, mise en place de la scène ;
- Le **OnGameStart** est utilisé pour déclencher la logique de la scène.

Afin de régler des problèmes liés au chargement des différentes scènes, aux phases de transitions, et à la gestion des Beats, il a été nécessaire de créer une nouvelle méthode qui divise en deux parties ce que l'on met traditionnellement dans un **Start**.

I. Gestion de l'audio

3. Consigne

Le traitement de l'Audio par le jeu Macro différencie la musique et les sons. Le jeu Macro utilise un **AudioMixer** pour gérer cela.

- Pour ajouter une musique, il faut placer MicroBGM dans le champ *Output* de son **AudioSource** ;
- Pour ajouter un son, il faut ajouter SFX dans le champ *Output* de son **AudioSource**.

[GIF]

4. Explications

Il y a deux raisons qui expliquent cette utilisation de **l'AudioMixer** intégré à Unity :

- Il permet d'appliquer une modification de volume dans les paramètres à tous les sons et toutes les musiques du projet, directement ;
- Le MicroBGM permet de gérer plus facilement la transition entre les musiques du micro-jeux et celles du jeu Macro.

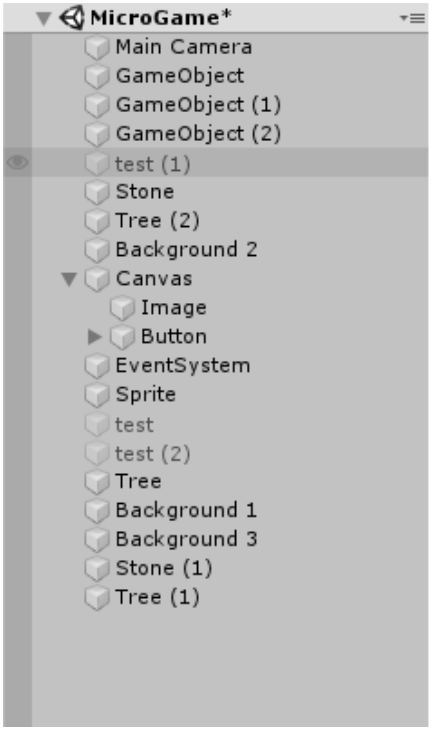
En conséquence, si des sons ou des musiques sont joués sans passer par ces paramètres, les audios du micro-jeu ne seront pas pris en charge par les paramètres du jeu Macro, entraînant des superpositions et des décalages sonores.

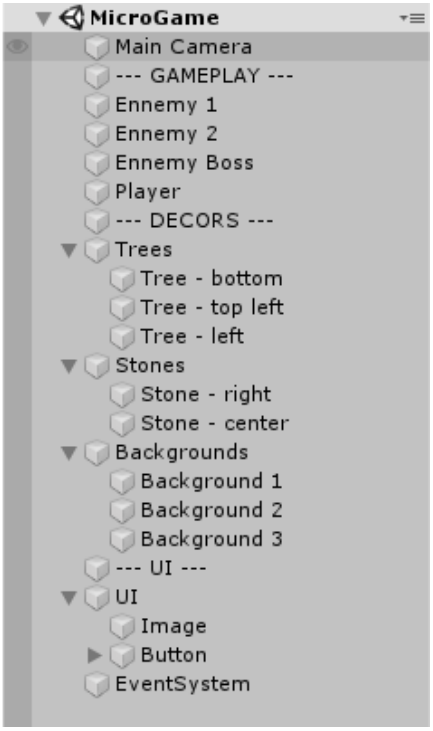
J. Organisation d'une scène de micro-jeu

11. Consigne

La Hiérarchie de la Scène doit être organisée tel qu'il suit :

- Les GameObjects doivent être regroupés par cohérence *(Environnement, Gameplay, Managers …) ;*
- Les GameObjects doivent être renommés en fonction de leur utilité ;
- Les GameObjects inutilisés doivent être supprimés de la scène ;
- La Scène doit avoir le nom du micro-jeu.





12. Explication

En cas de bug au moment de l'importation du micro-jeu, une organisation claire de la scène permet aux Mousquetaires d'intervenir directement sans avoir à passer par le développeur. En étant rigoureux sur les noms de GameObjects, sur l'arborescence de la Hiérarchie et sur les regroupements cohérents des éléments, on accélère la prise en charge des problèmes et leur résolution.

K. Les 10 commandements

La section peut donc se résumer en 10 commandements :

1. Un `namespace`, tu utiliseras.
2. Les **`Tags`**, tu banniras.
3. Aux **`Sorting Layers`** existant, tu te limiteras.
4. À `Macro.StartGame()` et `Macro.EndGame()` tu penseras.
5. Dans `Start()` ta scène tu prépareras.
6. Dans `OnGameStart()` ton jeu commencera.
7. Pour `OnEnable()` et `OnDisable()` utiliser, les Mousquetaires tu appelleras.
8. L'AudioMixer, tu respecteras.
9. D'une seule et unique Scène ton micro-jeu constitué sera.
10. La hiérarchie de ta Scène, clairement tu organiseras.

IV. Exportation du micro-jeu

[à venir]

A. GameID – la fiche d’identité du micro-jeu

[à venir]

B. Agencement des dossiers

[à venir]

C. Procédure d’exportation

[à venir]

V. Annexes

A. Lexique

Le lexique vise à clarifier l'utilisation de certains termes utilisés au sein de ce document. De plus le lexique sert aussi de rappels sur certaines fonctionnalités pour la programmation orientée objet en C#.

1. ScriptableObject

Les **ScriptableObjects** permettent de stocker des données indépendamment d'une classe. Ils n'ont pas besoin d'être instanciés mais peuvent être drag'n'droppés dans n'importe quel script. Les données sont conservées d'une scène à l'autre et aussi lorsqu'on les change en runtime, c'est-à-dire quand la scène est en **PlayMode**.

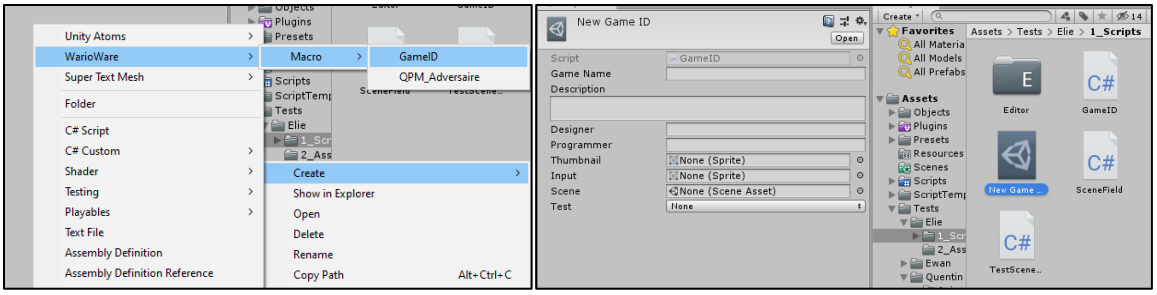
Ci-dessous, un **ScriptableObject** stockant plusieurs données comme un string **gameName** ou un **Sprite** Input.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  namespace Game
6  {
7      [CreateAssetMenu(menuName = "WarioWare/Macro/GameID")]
       2 references
8      public class GameID : ScriptableObject
9      {
10         public string gameName;
11         [TextArea(2,10)]public string description;
12         public string designer;
13         public string programmer;
14         public Sprite thumbnail;
15         public Sprite input;
16         public SceneField scene;
17         public test test;
18     }
```

Grâce à la ligne 7 :

```
[CreateAssetMenu(menuName = "WarioWare/Macro/GameID")]
```

Il est possible de créer n'importe quel **ScriptableObject GameID** dans nos dossiers :



Ainsi, le **ScriptableObject** est créé, et constitue une instance où ses variables sont modifiables et accessible par le script ou par l'inspecteur.

2. Keyword `static`

```
6   public class MyStaticScript
7   {
8       public static int SomeProperty
9       {
10          get { return 60; }
11      }
12
13      public static void SomeMethod()
14      {
15          // Do something
16      }
17  }
```

```
6   public class MyMonoScript : MonoBehaviour
7   {
8       void Start()
9       {
10          var someValue = MyStaticScript.SomeProperty;
11          MyStaticScript.SomeMethod();
12      }
13  }
```

Le keyword C# **`static`** indique que le champ, la propriété ou la méthode est relié au type dans lequel il est écrit, et non à une instance de ce type.

Dans l'exemple ci-dessous, le type **`MyStaticScript`** possède une propriété **`SomeProperty`** et une fonction **`SomeMethod`** qui sont toutes deux indiquées comme **`static`**. Ce qui fait que dans le type **`MyMonoScript`**, il n'y a pas besoin de références pour obtenir **`SomeProperty`** ou appeler **`SomeMethod`**. Il suffit d'écrire le type et d'accéder à la propriété ou la méthode.

3. Namespaces

Quelle est la différence entre ces deux images ?

```
0 references
public class Player : MonoBehaviour
{
    // Start is called before the first frame update
    0 references
    void Start()
    {
        Debug.Log("I'm a Player !");
    }

    // Update is called once per frame
    0 references
    void Update()
    {

    }
}
```

```
namespace Game.Vitruve
{
    0 references
    public class Player : MonoBehaviour
    {
        // Start is called before the first frame update
        0 references
        void Start()
        {
            Debug.Log("I'm a Player !");
        }

        // Update is called once per frame
        0 references
        void Update()
        {

        }
    }
}
```

À gauche il n'y a pas de **namespace**, à droite il y en a un.

Le **namespace** est une ligne de code qui isole les scripts créés en son sein pour éviter qu'ils n'entrent en conflit les uns avec les autres.

Il est particulièrement utile dans l'utilisation de scripts avec des noms récurrents comme **GameManager**, **Player**, **Controller** etc.

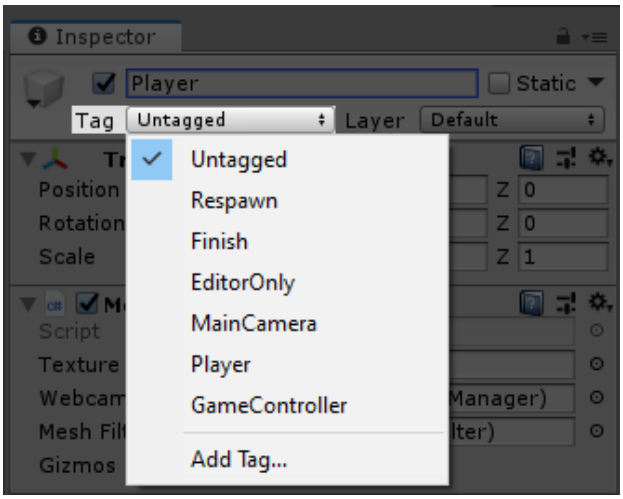
Pour déclarer un **namespace**, rien de plus simple : il suffit de mettre le mot-clef « **namespace** » suivi de n'importe quel mot, et d'ouvrir les accolades. Tout ce qui se trouve dans les accolades sera rattaché au **namespace**.

Dans l'exemple plus haut, l'ordinateur comprendra le script non plus comme simplement « Player » mais comme **Game.Vitruve.Player**, permettant d'éviter un conflit avec un autre micro-jeu qui aurait son script Player.

Attention ! Ne mets pas d'espace ou de ponctuation autre que le point après le « **Game** ». Aussi, les « **using** … » en haut du code se mettent en dehors du **namespace**.

Plus d'informations sur la documentation officielle : **ici**.

4. Les Tags de Unity



Les **Tags** correspondent à l'utilisation de la zone éclairée dans le screenshot ci-dessus. Par défaut à leur création, les GameObjects ont comme valeur de Tag « **Untagged** ». Sur tous les projets, Unity définit de base six valeurs de Tag en plus de « **Untagged** », ainsi que la possibilité de rajouter des tags.

Dans les scripts, certaines fonctions permettant de sélectionner les objets de la scène en fonction de leur Tag, et ainsi faciliter le travail de référencement ou de recherche de GameObjects précis.

Plus d'informations sur la documentation officielle : **ici**.

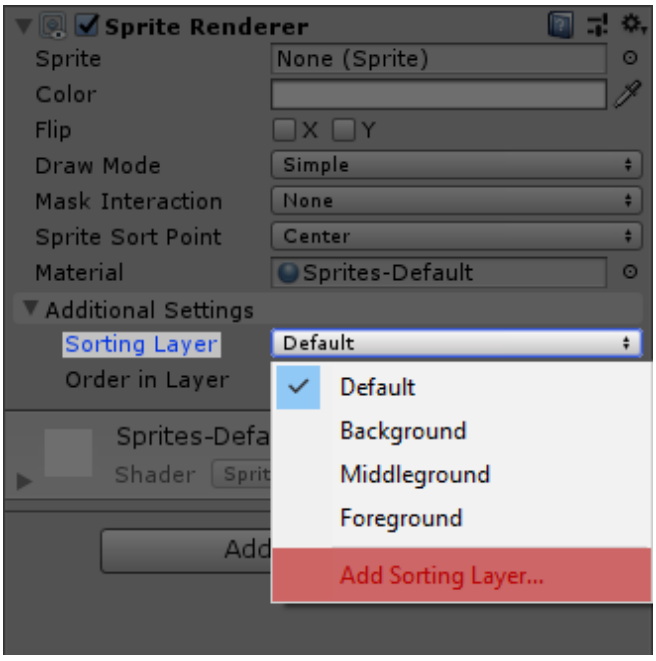
5. L'AudioMixer

L'AudioMixer est un asset semblable à un **ScriptableObject**. Il a déjà été paramétré par l'équipe des Mousquetaires. Il sert d'intermédiaire entre la source du son joué et ce que le joueur entend. On peut comparer l'AudioMixer à une table de mixage pour les sound designers en herbe.

6. Sorting Layer

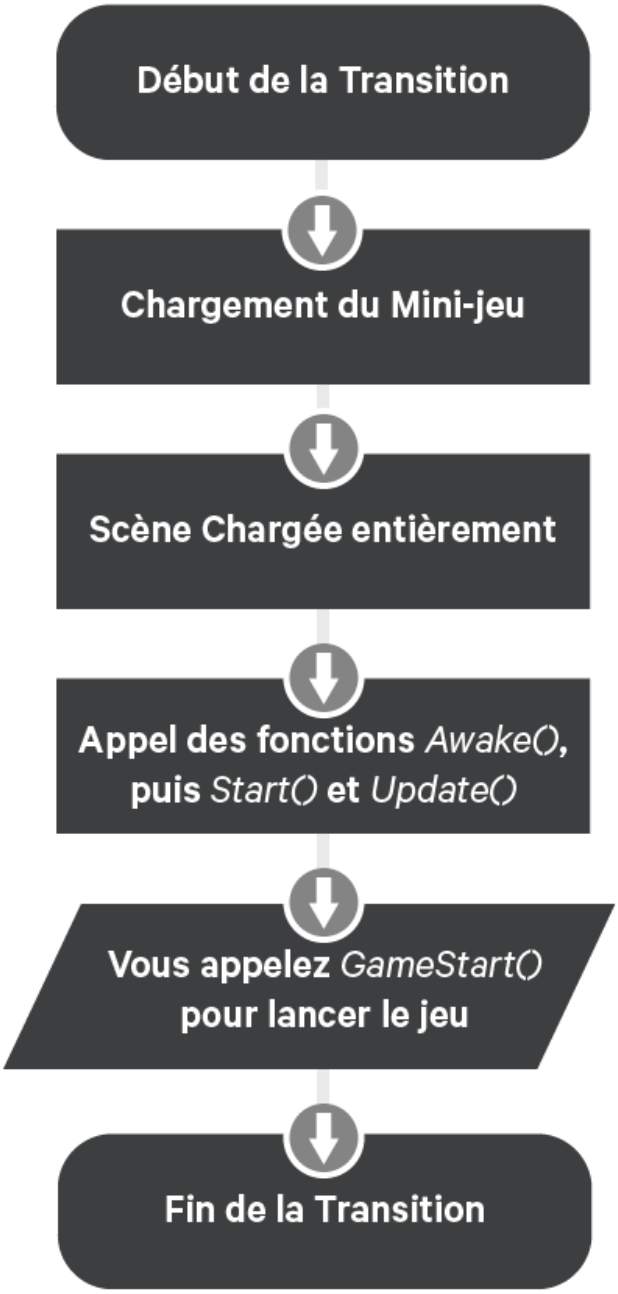
Les **`Sorting Layer`** servent de calques mais uniquement pour les **`Sprites`**.

Ainsi, un **`Sprite`** qui aura un **`Sorting Layer`** différent d'un autre **`Sprite`** sera rendu par la caméra en dessous ou au-dessus de l'autre **`Sprite`**. Les **`Sorting Layer`** sont accessibles dans l'inspecteur du composant **`Sprite Renderer`**.

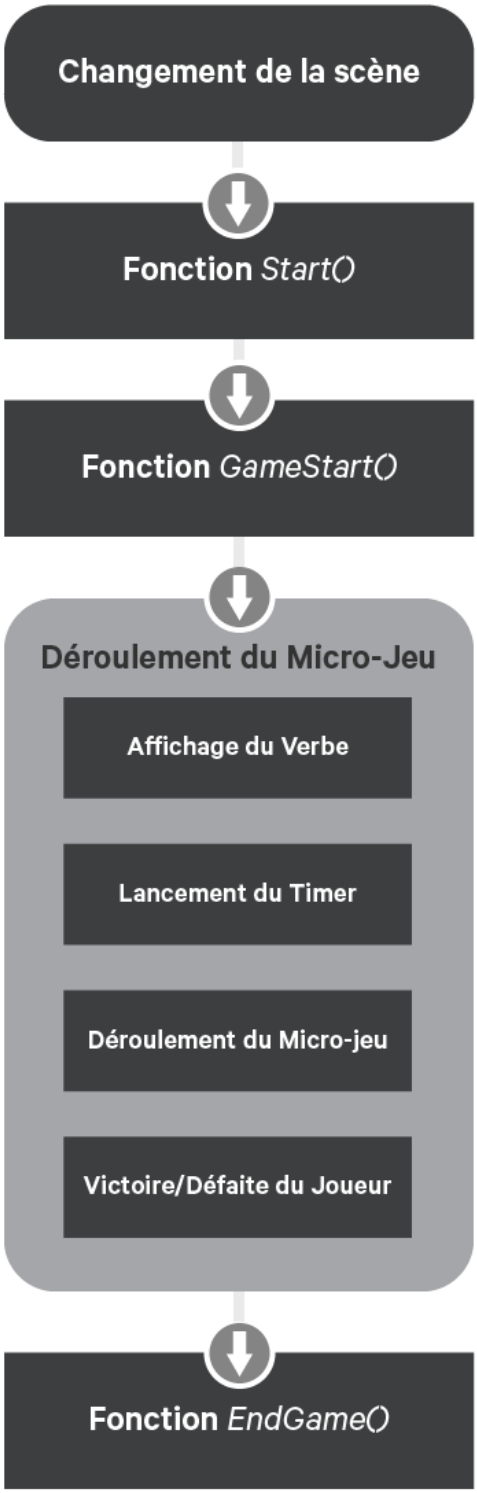


B. Flowcharts

1. Lancement d'un micro-jeu



2. Fonctions d’un micro-jeu



3. Séquence d’un micro-jeu