Trucs et astuces pour FYFY

Contenu

1 Accès au code source de FYFY

https://github.com/Mocahteam/FYFY

2 Boucle d'exécution de FYFY et étape de simulation

Chaque système FYFY peut être inscrit dans l'une des trois étapes de simulation suivantes : **FixedUpdate**, **Update** ou **LateUpdate**. La Figure 1 est une vue simplifiée du diagramme de flux d'Unity (https://docs.unity3d.com/uploads/Main/monobehaviour_flowchart.svg), elle présente les étapes de simulation dans lesquelles FYFY met à jour ces données. Ces mises à jour sont effectuées avant l'exécution des systèmes inscrits dans chacune de ces étapes de simulation. Le détail des étapes d'une mise à jour est présenté dans la Figure 2.

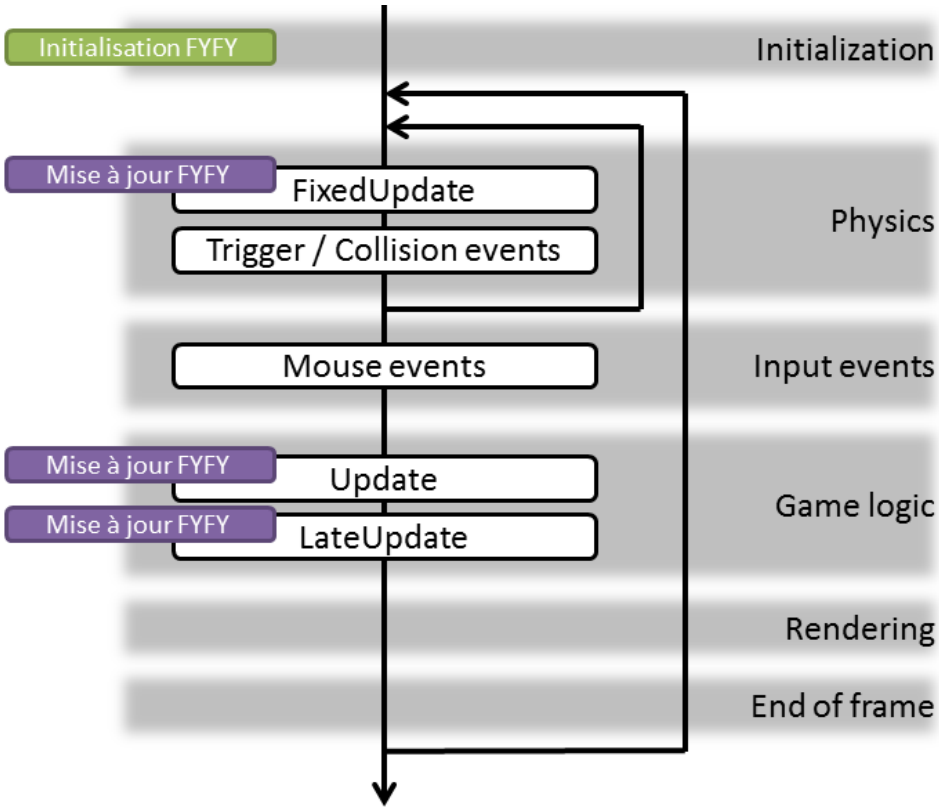

Figure 1

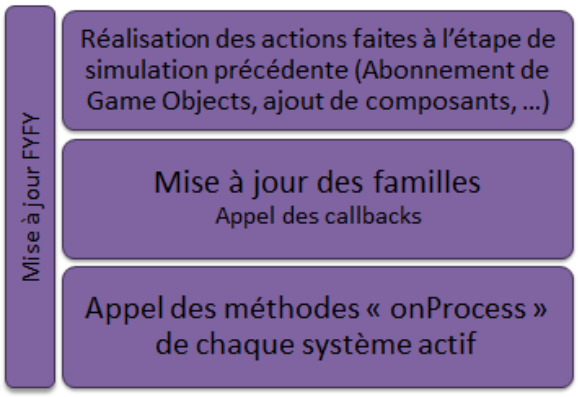

Figure 2

3 Intégration de FYFY à un projet Unity

Pour utiliser FYFY dans votre projet, vous devez intégrer les fichiers suivant : **FYFY.dll**, **FYFY.xml** et **FYFY_Inspector.dll**. Ces différents éléments ajoutent le menu **FYFY** (voir Figure 3) dans l'interface d'Unity vous permettant d'ajouter le GameObject **Main_Loop** à votre scène. C'est grâce à ce GameObject que vous pourrez associer vos systèmes aux différentes étapes de simulation et définir les GameObjects à abonner à Fyfy au démarrage de votre jeu.

Note : Pensez à définir la bibliothèque **FYFY_Inspector.dll** comme étant limitée au mode éditeur sans quoi vous obtiendrez une erreur du style : « Error building Player: Extracting referenced dlls failed. » lors de l'exportation de votre jeu. Pour ce faire sélectionnez la dll et dans l'Inspector cochez la case « Editor ».

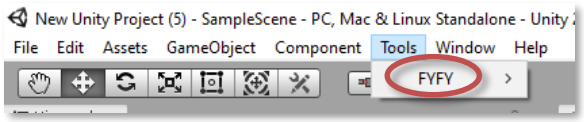

Figure 3

3.1 Ajout des compléments

En complément des trois fichiers précédents deux autres fichiers sont fournis : **0-FYFY__New Component-NewComponent.cs.txt** et **1-FYFY__New System-NewSystem.cs.txt**. Collez ces fichiers dans le dossier *C:\Programmes\Unity\Editor\Data\Resources\ScriptTemplates* (le chemin est à adapter en fonction de l'endroit où est installé Unity sur votre système). L'intégration de ces fichiers ajoute, **après redémarrage d'Unity**, un sous-menu FYFY au menu **Assets > Create** (voir Figure 4). Ce sous-menu FYFY permet de créer des composants et des systèmes pré-formatés pour une utilisation avec FYFY.

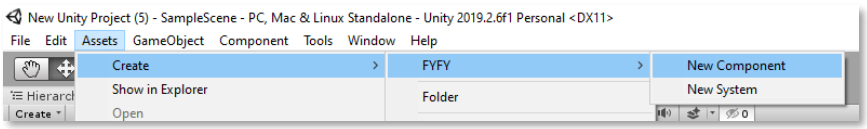

Figure 4

3.2 Intégration du Main_Loop

Afin de terminer l'intégration de **FYFY** à votre projet, vous devez créer la boucle principale qui vous permettra de gérer l'ordre d'exécution des systèmes et donc votre simulation. Via le menu **FYFY**, sélectionnez l'option **Create Main Loop**, comme illustré dans la Figure 5.

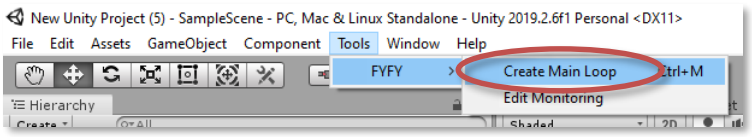

Figure 5

Par défaut, tous les GameObjects présents dans la fenêtre **Hierarchy** seront au3tomatiquement traités par FYFY au démarrage du jeu et abonnés à leurs familles respectives. Si vous souhaitez contrôler les GameObjects traités par FYFY (voir Figure 6), vous pouvez :

- exclure un ensemble de GameObjects en sélectionnant **Do not bind specified Game Objects on Start** et en glissant les GameObjects à exclure depuis la fenêtre **Hierarchy** dans le champ prévu à cet effet ;
- inclure uniquement un ensemble de GameObjects en sélectionnant **Bind only specified Game Objects on Start** et en glissant les GameObjects à inclure depuis la fenêtre **Hierarchy** dans le champ prévu à cet effet ;

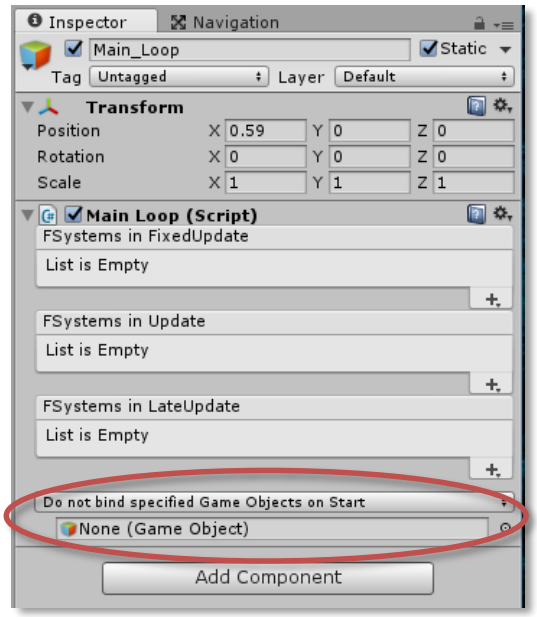

Figure 6

À noter que la vue du GameObject **Main_Loop** change dans l'**Inspector** lorsque vous lancez l'exécution de votre jeu (voir un exemple dans la Figure 7). Dans cette vue, vous ne pouvez plus abonner de nouveaux système aux différents contextes ou modifier la liste des GameObjects à exclure/inclure. En revanche trois outils de vous sont proposés :

- Le premier vous permet de gérer l'exécution de vos systèmes, chaque système peut être mis en pause de manière indépendante en cliquant sur les **interrupteurs vert/rouge** ;
- Le second (**FSystem profiler**) vous permet de visualiser le temps d'exécution de chaque contexte et ainsi identifier les systèmes gourmands en ressources en vue d'éventuelles optimisations ;

- Le troisième (**Bind tools**) vous permet au runtime d'ajouter ou retirer dynamiquement un Game Object à l'environnement FYFY.
- Le quatrième (**Families Inspector**) vous permet d'inspecter vos familles afin de connaître les GameObjects les peuplant. Très utile pour vérifier si les GameObjects présents dans les familles sont bien ceux attendus, sans quoi une révision de la déclaration des familles devra être réalisée dans les systèmes considérés pour améliorer le filtrage.

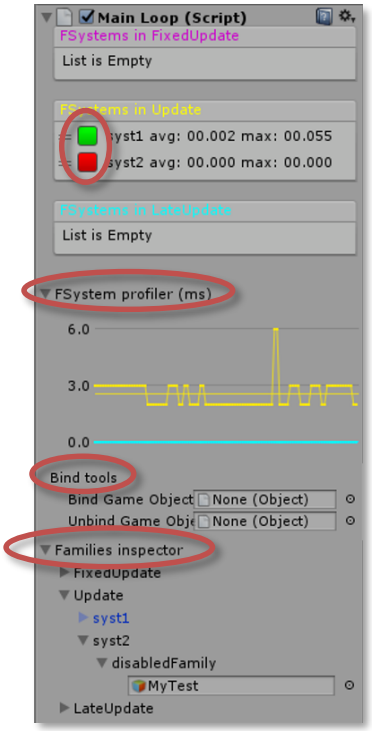

Figure 7

4 Notes sur l'implémentation d'un système

Un système est une classe héritant de **FSystem**. Si vous avez installé les compléments (voir section 3.1), utilisez le menu **Assets > Create > FYFY > New System** pour créer un nouveau système. Le template de création d'un système vous invite à implémenter trois méthodes : **onPause**, **onResume** et **onProcess**.

L'instanciation des systèmes est automatiquement gérée par le **Main_Loop**. Vous pouvez donc implémenter le constructeur d'un système si vous souhaitez faire un traitement particulier lors de sa création. Au moment de l'appel du constructeur les familles sont déjà peuplées vous pouvez donc les parcourir et traiter en conséquences les données de votre jeu.

Attention : Si vous utilisez le pluggin Monitoring (voir section 9.4) et que vous implémentez des constructeurs dans vos systèmes, vous devrez vérifier vos parcours de famille dans ces constructeurs. En effet le module Monitoring doit instancier et inspecter les systèmes de votre projet dans le contexte de l'éditeur pour en extraire les familles et vous les proposer dans la fenêtre d'édition du suivi. Hors en mode édition les familles ne sont pas peuplées, vous devrez donc adapter le code de vos constructeurs en conséquence.

5 Création / Modification / Suppression dynamique de GameObjects

Avec FYFY les GameObjects sont gérés au travers de la classe statique **GameObjectManager** fournie dans la bibliothèque **FYFY.dll**. Cette classe statique permet l'abonnement et le désabonnement de GameObjects au module Fyfy et la modification (ajout/retrait de composants, changement de propriétés) de GameObjects existants. Elle permet également le chargement de scènes.

Une spécificité importante du **GameObjectManager** est l'asynchronicité de l'ensemble de ces fonctions. En effet afin de maintenir la cohérence de la simulation lors de l'exécution de deux systèmes enregistrés dans la même étape de simulation, toute demande de modification d'un GameObject effectuée via le **GameObjectManager** est reportée et exécutée au début de la prochaine étape de simulation.

5.1 Abonnement et désabonnement d'un GameObject à Fyfy

Seuls les GameObjects abonné à Fyfy seront accessibles dans vos familles. Au démarrage du jeu les GameObjects respectant la règle de filtrage du Main_Loop seront automatiquement abonnés à Fyfy. Toutefois, vous pouvez à tout moment abonner un GameObject à Fyfy si vous souhaitez le gérer à l'aide du formalisme ECS. Ceci est utile lorsque vous créez dynamiquement des GameObjects en cours de simulation. Vous pouvez également réaliser le processus inverse à savoir désabonner un GameObject de Fyfy pour qu'il ne soit plus géré par Fyfy.

Pour abonner un GameObject à Fyfy vous devez utiliser la méthode **bind**, exemple :

```
// Abonnement d'un nouveau GameObject à Fyfy
GameObjectManager.bind (go);
```

Pour désabonner un GameObject de Fyfy vous devez utiliser la méthode **unbind**, exemple :

```
// Désabonnement d'un GameObject de Fyfy
GameObjectManager.unbind (go);
```

Note : Vous devez toujours désabonner un GameObject de Fyfy avant de le détruire.

5.2 Modification dynamique d'un GameObject

Modifier un GameObject consiste à ajouter/retirer dynamiquement des composants à ce GameObject ou à modifier ses propriétés comme son état (actif/inactif), son parent, son layer ou son tag, exemples :

```
// Ajoute le composant SphereCollider avec un rayon de 2 au début de la prochaine
// étape de simulation
GameObjectManager.addComponent<SphereCollider> (go, new { radius = 2f });
// ou
GameObjectManager.addComponent (go, typeof(SphereCollider), new { radius = 2f });
```

```
// Supprime le composant SphereCollider au début de la prochaine étape de
// simulation
GameObjectManager.removeComponent<SphereCollider> (go);
// ou
SphereCollider sc = go.GetComponent<SphereCollider>();
GameObjectManager.removeComponent (sc);
```

```
// Active/Désactive le GameObject au début de la prochaine étape de simulation
GameObjectManager.setGameObjectState (go, newState);
```

```
// Change le parent d'un GameObject au début de la prochaine étape de simulation
// Si le dernier paramètre est à true, les propriétés relatives du parent
// (position, rotation mise à l'échelle) sont modifiées pour que le fils conserve
// ses propriétés dans l'espace du monde
GameObjectManager.setGameObjectParent (go, parent_go, true);
```

```
// Change le layer d'un GameObject au début de la prochaine étape de simulation
GameObjectManager.setGameObjectLayer (go, newLayer);
```

```
// Change le tag d'un GameObject au début de la prochaine étape de simulation
GameObjectManager.setGameObjectTag (go, newTag);
```

5.3 Chargement d'une nouvelle scène

La classe **GameObjectManager** vous permet aussi de déclencher le chargement d'une nouvelle scène à l'aide des fonctions **loadScene**, exemple :

```
// Chargement d'une scène à partir de son index dans le Build Settings (Ctrl +
// Shift + b)
GameObjectManager.loadScene (index);
```

```
// Chargement d'une scène à partir de son nom
GameObjectManager.loadScene (sceneName);
```

Dans le cas d'un chargement de scène à partir de son nom, le paramètre **sceneName** peut ne contenir que la dernière partie du chemin d'accès sans l'extension « **.unity** ». Si un chemin complet est indiqué, il doit être identique à l'un de ceux présentés dans la fenêtre **Build Settings** (Ctrl + Shift + b). Si seulement le nom de la scène est donné, la première scène au nom identique dans la liste de la fenêtre **Build Settings** sera chargée. Si plusieurs scènes ont le même nom mais des chemins différents, vous devriez utiliser le chemin d'accès complet. A noter que le chemin n'est pas sensible à la case.

5.4 Cas du DontDestroyOnLoad

Unity fournit un mécanisme permettant de rendre persistant un GameObject lors d'un changement de scène, il s'agit du **DontDestroyOnLoad**. Pour utiliser ce mécanisme avec Fyfy, la classe **GameObjectManager** fournit la fonction **dontDestroyOnLoadAndRebind**, exemple :

```
// Rend le GameObject "go" persistant et le réabonnera automatiquement à Fyfy lors
// du prochain changement de scène contenant un MainLoop
GameObjectManager.dontDestroyOnLoadAndRebind (go);
```

6 Gestion des familles

Avec FYFY, les familles sont gérées au travers de la classe statique **FamilyManager** fournie dans la bibliothèque **FYFY.dll**. Cette classe statique permet de définir une famille via la fonction **getFamily**, il est alors possible de la parcourir et définir des écouteurs sur cette famille.

6.1 Définition d'une famille : les matchers

Pour définir une famille vous avez à votre disposition un ensemble de matcher qui vous permettent de définir des « filtres » sélectionnant les GameObjects abonnés à Fyfy qui satisfont l'ensemble des contraintes.

6.1.1 Les matchers de composants

Trois matchers différents sur les composants vous permettent de définir une famille : **AllOfComponents**, **AnyOfComponents** et **NoneOfComponents**.

6.1.1.1 AllOfComponents

Ce matcher vous permet de définir une famille composée des GameObjects contenant TOUT les composants définis dans le matcher, exemple :

```
// Définition d'une famille composée des GameObjects contenant les composants Move
// ET RandomTarget
Family myFamily = FamilyManager.getFamily(new AllOfComponents(typeof(Move),
typeof(RandomTarget)));
```

6.1.1.2 AnyOfComponents

Ce matcher vous permet de définir une famille composée des GameObjects contenant AU MOINS UN composant parmi ceux définis dans le matcher, exemple :

```
// Définition d'une famille composée des GameObjects contenant les composants Move
// OU RandomTarget
Family myFamily = FamilyManager.getFamily(new AnyOfComponents(typeof(Move),
typeof(RandomTarget)));
```

6.1.1.3 NoneOfComponents

Ce matcher vous permet de définir une famille composée des GameObjects NE contenant PAS les composants définis dans le matcher, exemple :

```
// Définition d'une famille composée des GameObjects ne contenant pas les
composants Move ET RandomTarget
Family myFamily = FamilyManager.getFamily(new NoneOfComponents(typeof(Move),
typeof(RandomTarget)));
```

6.1.2 Les matchers sur les layers

6.1.2.1 AnyOfLayers

Ce matcher vous permet de définir une famille composée des GameObjects APPARTENANT à un des layers définis dans le matcher, exemple :

```
// Définition d'une famille composée des GameObjects appartenant aux layers 5, 6 ou
// 7
Family myFamily = FamilyManager.getFamily(new AnyOfLayers(5, 6, 7));
```

6.1.2.2 *NoneOfLayers*

Ce matcher vous permet de définir une famille composée des GameObjects n'appartenant à AUCUN des layers définis dans le matcher, exemple :

```
// Définition d'une famille composée des GameObjects n'appartenant pas aux layers
// 5, 6 ou 7
Family myFamily = FamilyManager.getFamily(new NoneOfLayers(5, 6, 7));
```

6.1.3 Les matchers sur les tags

6.1.3.1 *AnyOfTags*

Ce matcher vous permet de définir une famille composée des GameObjects ASSOCIES à un des tags définis dans le matcher, exemple :

```
// Définition d'une famille composée des GameObjects associés aux tags "virus" ou
// "bactery"
Family myFamily = FamilyManager.getFamily(new AnyOfTags("virus", "bactery"));
```

6.1.3.2 *NoneOfTags*

Ce matcher vous permet de définir une famille composée des GameObjects NON associés à un des tags définis dans le matcher, exemple :

```
// Définition d'une famille composée des GameObjects non associés aux tags "virus"
// et "bactery"
Family myFamily = FamilyManager.getFamily(new NoneOfTags("virus", "bactery"));
```

6.1.4 Les matchers sur les propriétés

Les propriétés compatibles avec les matchers de propriétés sont définies sous la forme d'une énumération dans la classe dans la classe PropertyMatcher. Il s'agit de **ACTIVE_SELF**, **ACTIVE_IN_HIERARCHY**, **HAS_PARENT** et **HAS_CHILD**.

6.1.4.1 *AllOfProperties*

Ce matcher vous permet de définir une famille composée des GameObjects ayant TOUTES les propriétés définies dans le matcher, exemple :

```
// Définition d'une famille composée des GameObjects étant actifs ET ayant un
// parent
Family myFamily = FamilyManager.getFamily(
    new AllOfProperties(PropertyMatcher.PROPERTY.ACTIVE_IN_HIERARCHY,
                        PropertyMatcher.PROPERTY.HAS_PARENT));
```

6.1.4.2 *AnyOfProperties*

Ce matcher vous permet de définir une famille composée des GameObjects ayant AU MOINS une des propriétés définies dans le matcher, exemple :

```
// Définition d'une famille composée des GameObjects étant actifs OU ayant un
// parent
Family myFamily = FamilyManager.getFamily(
    new AnyOfProperties(PropertyMatcher.PROPERTY.ACTIVE_IN_HIERARCHY,
                        PropertyMatcher.PROPERTY.HAS_PARENT));
```

6.1.4.3 *NoneOfProperties*

Ce matcher vous permet de définir une famille composée des GameObjects n'ayant AUCUNE des propriétés définies dans le matcher, exemple :

```
// Définition d'une famille composée des GameObjects étant inactif et n'ayant pas
// de parents
Family myFamily = FamilyManager.getFamily(
    new NoneOfProperties(PropertyMatcher.PROPERTY.ACTIVE_IN_HIERARCHY,
                         PropertyMatcher.PROPERTY.HAS_PARENT));
```

6.1.5 Combinaison des matchers

Vous pouvez combiner les différents matchers afin de définir des filtres avancés, exemple

```
// Définition d'une famille composée des GameObjects inactifs ayant les composants
// Move ET RandomTarget MAIS PAS Velocity et appartenant au layer 3 ou 7
Family myFamily = FamilyManager.getFamily(
    new AllOfProperties(PropertyMatcher.PROPERTY.ACTIVE_IN_HIERARCHY),
    new AllOfComponents(typeof(Move), typeof(RandomTarget)),
    new NoneOfComponents(typeof(Velocity)),
    new AnyOfLayers(3, 7));
```

6.2 Parcours des familles

La méthode la plus simple et la plus sûre pour parcourir vos familles est d'utiliser une boucle **foreach**, exemple :

```
// Définition d'une famille
Family myFamily = FamilyManager.getFamily(...);
// Parcours de la famille
foreach (GameObject go in myFamily){
      // Exploitation du GameObject
      ...
}
```

Une méthode plus rapide d'exécution consiste à utiliser une boucle **for**, exemple :

```
// Définition d'une famille
Family myFamily = FamilyManager.getFamily(...);
// Parcours de la famille
int count = myFamily.Count;
for (int i = 0 ; i < count ; i++){
      GameObject go = myFamily.getAt(i);
      // Exploitation du GameObject
      ...
}
```

Attention : cette solution peut vous renvoyer des valeurs nulles si vous omettez de désabonner un GameObject de Fyfy avant de le détruire (voir section 5.1).

Vous pouvez aussi accéder au premier GameObject de la famille à l'aide de la méthode **First**. Si la famille est vide, cette méthode retournera la valeur **null**. A noter également que le premier GameObject de la famille peut évoluer au cours du temps, vous n'avez donc aucune garantie que deux appels successifs de cette méthode renvoient le même GameObject. Exemple d'usage :

```
// Accès au premier GameObject de la famille
GameObject go = myFamily.First();
```

6.3 Ajout d'écouteurs sur une famille

Vous pouvez définir des écouteurs sur l'entrée et la sortie de GameObjects d'une famille. Cette fonctionnalité est utile pour n'effectuer un traitement qu'une seule fois lors de l'ajout ou du retrait d'un GameObject d'une famille.

Attention : Les écouteurs sont appelés après la mise à jour des familles et avant l'appel des fonctions **OnProcess** des systèmes. Par conséquent, la modification d'un GameObject dans un écouteur (ajout d'un composant par exemple) qui serait annulé par le retrait de ce même composant dans un **OnProcess** n'entrainera pas de mise à jour des familles à la prochaine étape de simulation et donc de redéclanchement des écouteurs.

6.3.1 Ecouteur sur l'ajout d'un GameObject à une famille

Vous pouvez abonner des écouteurs sur l'ajout de GameObjects à une famille via la fonction **addEntryCallback** de la classe **Family**. Les écouteurs que vous abonnez via cette fonction doivent respecter la signature suivante :

```
void nomEcouteur (GameObject go);
```

Exemple :

```
// Définition d'une famille
Family myFamily = FamilyManager.getFamily (...);
// Enregistrement d'un écouteur sur l'ajout d'un GameObject à la famille
myFamily.addEntryCallback (myFunction);

// Définition de l'écouteur
void myFunction (GameObject addingGo){
    // traitement du GameObject passé en paramètre arrivant dans la famille
    ...
}
```

6.3.2 Écouteur sur le retrait d'un GameObject d'une famille

Vous pouvez abonner des écouteurs sur le retrait de GameObjects d'une famille via la fonction **addExitCallback** de la classe **Family**. Les écouteurs que vous abonnez via cette fonction doivent respecter la signature suivante :

```
void nomEcouteur (int gameObjectInstanceId);
```

Contrairement aux écouteurs sur l'ajout d'un GameObject à une famille, ce n'est pas le GameObject qui est passé en paramètre mais seulement son identifiant d'instance. En effet, un GameObject peut être retiré d'une famille s'il ne respecte plus les filtres définis par les matcher ou si le GameObject est désabonné de l'environnement FYFY en raison de sa destruction par exemple. Dans ce dernier cas, il serait impossible de passer le GameObject en paramètre de l'écouteur. Seul son identifiant est donc transmis.

Exemple :

```
// Définition d'une famille
Family myFamily = FamilyManager.getFamily (...);
// Enregistrement d'un écouteur sur le retrait d'un GameObject de la famille
myFamily.addExitCallback (myFunction);

// Définition de l'écouteur
void myFunction (int removingGoInstanceId){
    // traitement de l'identifiant d'instance du GameObject retiré de la famille
    ...
}
```

7 Lier les évènements d'UI avec les systèmes

Dans un approche classique d'Unity, l'utilisateur peut associer des fonctions définies dans des composants à des évènements d'UI directement dans l'**Inspector** (par exemple lors d'un clic sur un bouton). Pour permettre cette fonctionnalité très utile d'Unity dans un contexte d'ECS où les composants ne contiennent pas de fonctions, FYFY inspecte vos systèmes et maintient automatiquement des wrappers qui exposent les fonctions de vos systèmes pour les rendre accessibles dans l'**Inspector**. Les fonctions retenues par FYFY doivent respecter la signature imposée par Unity à savoir **être publiques, ne pas renvoyer de résultat et contenir au maximum un paramètre**. Si ces contraintes sont respectées l'utilisateur pourra associer dans l'**Inspector** la fonction à appeler lorsqu'un évènement se produit. Il devra pour ce faire rechercher dans le **Main_Loop** le wrapper associé au système contenant la fonction à appeler et sélectionner cette dernière fonction.

La Figure 8 illustre l'association de la fonction « **StarGame** » définie dans le système « **MenuSystem** » via le **Main_Loop** lors d'un clic sur un bouton.

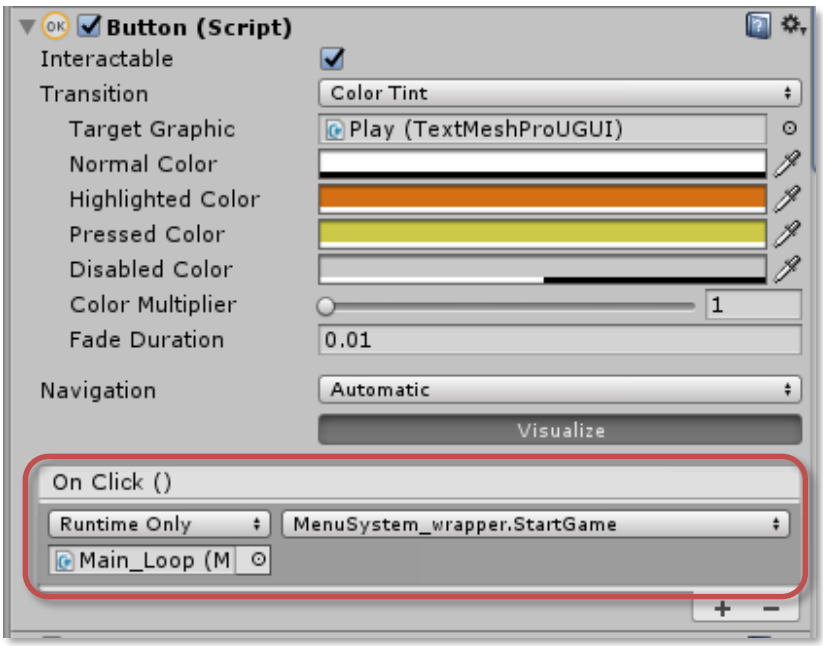

Figure 8

8 Utilisation des coroutines

Pour plus d'information sur l'intérêt des coroutines veuillez-vous référer à la documentation Unity : https://docs.unity3d.com/Manual/Coroutines.html

Dans Unity seul un MonoBehaviour peut démarrer une coroutine. Si vous souhaitez lancer une coroutine depuis l'un de vos systèmes vous pouvez passer par le MainLoop (voir illustration ci-dessous)

```
using UnityEngine;
using FYFY;
using System.Collections;

public class SystemExemple : FSystem {
    // Constructeur
    public SystemExemple ()
    {
        // Lancement de la coroutine définie dans le système via le MainLoop
        MainLoop.instance.StartCoroutine(coroutineExemple());
    }

    // Définition de la coroutine
    private IEnumerator coroutineExemple()
    {
        for (int i = 0; i < 100; i++)
        {
            Debug.Log(Time.frameCount + " " + i);
            yield return new WaitForSeconds(.1f);
        }
    }
}
```

9 Pluggins

Quatre modules complémentaires sont fournis pour profiter des fonctionnalités d'Unity avec le formalisme Entité-Composant-Système.

9.1 PointerManager

Le gestionnaire de pointeur (souris, touché) est fourni via la bibliothèque **PointerManager.dll**. L'intégration de ce module à votre projet Unity vous donne accès au composant **PointerSensitive**. Si vous ajoutez ce composant à un GameObject **contenant également un GUIElement ou un Collider**, il deviendra sensible aux pointeurs dans le sens où il contiendra un composant **PointerOver** si un pointeur est positionné au-dessus de lui. Vous pouvez alors exploiter ce composant **PointerOver** dans vos matchers pour créer une famille contenant les objets positionnés sous le pointeur. Il ne vous reste plus qu'à tester l'état du pointeur avec les fonctions classiques d'Unity (exemple : `Input.GetMouseButton(0)` retourne `true` si le bouton gauche de la souris est pressé).

Attention : Un GameObject avec un **RigidBody** « absorbe » tous les évènements générés par les **Colliders** de leurs enfants s'ils ne contiennent pas eux même un **RigidBody**. Exemple : Soit un GameObject contenant un RigidBody et un fils contenant lui-même un Collider et un PointerSensitive, lorsque le pointeur est positionné au-dessus du fils aucun PointerOver ne lui sera affecté car l'évènement sera traité par le RigidBody du parent. Vous devez donc soit supprimer le RigidBody du parent soit en ajouter un au fils. Exploitez cette propriété en conséquence.

9.2 CollisionManager

Le gestionnaire de collision est fourni via la bibliothèque **CollisionManager.dll**. L'intégration de ce module à votre projet Unity vous donne accès aux composants **CollisionSensitive2D** et **CollisionSensitive3D**. Si vous ajoutez l'un de ces composants à un GameObject **contenant également un Collider**, il deviendra sensible aux collisions dans le sens où il contiendra un composant **InCollision2D** (respectivement **InCollision3D**) durant toute la période de temps où la collision est effective. Vous pouvez alors exploiter les composants **InCollision** dans vos matchers pour créer une famille contenant les objets actuellement en collision. Il ne vous reste plus qu'à extraire de ces composants **InCollision** les GameObjects cibles (attribut **Targets**) et les données des collisions (attribut **Collisions**) à exploiter.

Note 1 : Les collisions ne sont détectées que si au moins un des GameObject intervenant dans la collision contient un **RigidBody** *non-kinematic*.

Note 2 : Dans Unity, **les collisions se propagent des fils aux parents**. Exemple : Soit un GameObject A contenant un CollisionSensitive et un fils (ce fils contient un Collider), si le GameObject fils entre en collision avec un GameObject B contenant également un Collider et un RigidBody, son père (le GameObject A) sera informé de la collision (via le composant InCollision) même s'il n'est pas lui-même directement en contact avec le GameObject B. Exploitez donc ces propriétés en conséquence.

Note 3 : Vous pouvez paramétrer les moteurs physiques d'Unity pour restreindre la détection des collisions à certains layers. Cette astuce est très utile pour limiter l'influence des **CollisionSensitive** aux seuls GameObjects appartenant à certains layers. Menu **Edit** > **Project Settings** > **Physics** / **Physics 2D** puis définissez la **Layer Collision Matrix**.

9.3 TriggerManager

Dans le cas où les données relatives aux collisions ne vous sont pas utiles vous pouvez noter les **Colliders** comme des **Triggers** (voir propriété **isTrigger** des **Colliders**). Définir un **Collider** comme un **Trigger** permet d'économiser des ressources. Unity se chargera simplement de générer des messages indiquant la présence de collisions sans faire appel au moteur physique pour résoudre les conséquences des collisions (points d'impact, forces d'impact...).

Le gestionnaire de trigger est fourni via la bibliothèque **TriggerManager.dll**. L'intégration de ce module à votre projet Unity vous donne accès aux composants **TriggerSensitive2D** et **TriggerSensitive3D**. Si vous ajoutez l'un de ces composants à un GameObject **contenant également un Collider**, il deviendra sensible aux triggers dans le sens où il contiendra un composant **Triggered2D** (respectivement **Triggered3D**) durant toute la période de temps où la collision est effective. Vous pouvez alors exploiter les composants **Triggered** dans vos matchers pour créer une famille contenant les objets actuellement en collision. Il ne vous reste plus qu'à extraire de ces composants **Triggered** les GameObjects cibles (attribut **Targets**) à exploiter.

Note : Les notes du **CollisionManager** sont également vraies pour le **TriggerManager**.

Cas d'utilisation 1 : Soit un GameObject A qui doit détecter les collisions avec tous les GameObjects de la scène contenant un Collider. Le GameObject A contiendra un TriggerSensitive + un RigidBody + un Collider avec la propriété isTrigger **activée**. Les autres GameObjects contenant un composant Collider **n'ont pas l'obligation d'avoir leur propriété IsTrigger activée**.

Cas d'utilisation 2 : Soit un GameObject A qui doit détecter les collisions avec le GameObjects B mais pas le GameObject C. Le GameObject A contiendra un TriggerSensitive + un Collider avec la propriété

isTrigger **désactivée**, le GameObjects B contiendra un Collider avec la propriété isTrigger **activée** et le Game Object C contiendra un Collider avec la propriété isTrigger **désactivée**.

Conclusion : En fonction des besoins la propriété **isTrigger** peut être définie soit sur le composant Collider du GameObject contenant le TriggerSensitive, soit sur le composant Collider du GameObject cible. Dans certains cas il est utile d'activer cette propriété sur les deux Colliders.

9.4 Monitoring

Le module de suivi a pour objectif de fournir un ensemble d'outils permettant de tracer les actions du joueur en vue d'une analyse avec l'outil **Laalys**. Ce module est donc composé de deux bibliothèques : **Monitoring.dll** et **Monitoring_Inspector.dll**.

Ces deux bibliothèques fournissent le composant **MonitoringManager**. Ce composant est un singleton, une seule instance de ce composant peut être définit dans une scène. Il est généralement ajouté au **Main_Loop**. Le **MonitoringManager** (voir Figure 9) permet de générer les réseaux de Petri et leurs caractéristiques en fonction des paramètres définis dans la fenêtre d'édition du suivi (voir section 9.4.1). Il permet également de définir le chemin d'accès à l'outil Laalys et les chemins d'accès aux différentes ressources pour une analyse des actions du joueur en cours de jeu (réseaux de Petri complets, filtrés et leurs caractéristiques).

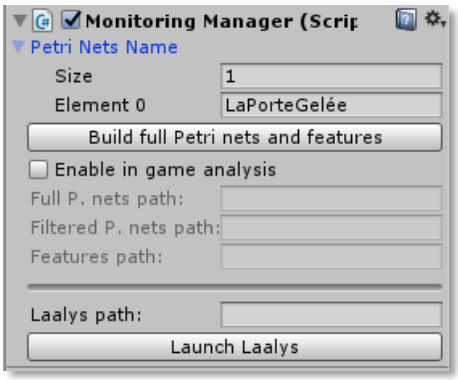

Figure 9

Note 1 : La liste **Petri Nets Name** vous permet de définir plusieurs noms de réseau de Petri complets. Cette option est utile si vous souhaitez décomposer votre suivi en plusieurs sous-parties. Par défaut le nom du réseau complet proposé est celui du nom de la scène chargée (ce nom ne peut être supprimé).

Note 2 : Si l'option **Enable in game analysis** est activée, le chemin d'accès à l'outil Laalys doit être défini (ex : ./foo/bar/LaalysV2.jar). L'environnement **Java** est requis pour l'exécution de l'outil Laalys.

Note 3 : Si vous implémentez des constructeurs dans vos systèmes, vous devrez vérifier vos parcours de famille dans ces constructeurs. En effet le module Monitoring doit instancier et inspecter les systèmes de votre projet dans le contexte de l'éditeur pour en extraire les familles et vous les proposer dans la fenêtre d'édition du suivi. Hors en mode édition les familles ne sont pas peuplées, vous devrez donc adapter le code de vos constructeurs en conséquence.

De même que pour la bibliothèque FYFY_Inspector.dll, vous devez penser à définir la bibliothèque **Monitoring_Inspector.dll** comme étant limitée au mode éditeur sans quoi vous obtiendrez une erreur du style : « Error building Player: Extracting referenced dlls failed. » lors de l'exportation de votre jeu. Pour ce faire sélectionnez la dll et dans l'Inspector cochez la case « Editor ».

9.4.1 Fenêtre d'édition du suivi

Les bibliothèques **Monitoring.dll** et **Monitoring_Inspector.dll** ajoutent au menu FYFY l'entrée **Edit Monitoring** (voir Figure 10) qui permet d'accéder à la fenêtre de configuration du suivi.

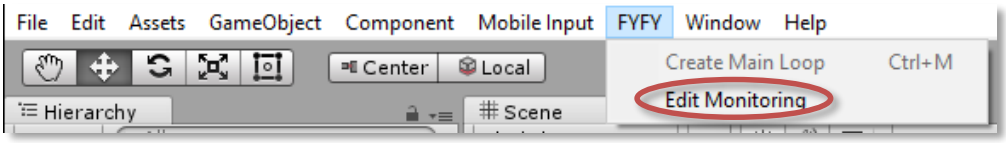

Figure 10

Vous pourrez à l'aide de cette fenêtre :

- Définir les GameObjects/Familles que vous souhaitez intégrer au suivi
- Définir à quel réseau de Petri complet l'élément suivi sera intégré
- Associer, à chaque élément suivi, un réseau de Petri au format **pnml**. Ce réseau de Pétri doit modéliser les actions possibles sur cet élément que vous souhaitez suivre (les transitions du réseau de Petri) et les états de cet élément contraignant ces actions (les places du réseau de Petri).
- Définir l'état initial et les propriétés des actions des éléments suivis
- Ajouter pour chaque action suivie des liens avec les états d'autres éléments suivis.

Note 1 : Intégrer des GameObjects ou des familles au module de suivi ajoute automatiquement le composant **ComponentMonitoring** (respectivement **FamilyMonitoring**) aux éléments suivis. Ces deux composants ne peuvent être paramétrés directement dans l'**Inspector** et sont configurés via la fenêtre d'édition du suivi (voir Figure 10).

Note 2 : Plusieurs **ComponentMonitoring** peuvent être associés à un même GameObject.

Note 3 : Le composant **FamilyMonitoring** hérite du composant **ComponentMonitoring**.

Le composant **MonitoringManager** vous donne également accès à un ensemble de méthodes statiques.

9.4.2 Fonction « trace »

La fonction **trace** vous permet notamment de tracer une action de jeu. Cette fonction requiert au minimum un **ComponentMonitoring** ou une famille, le nom de l'action à tracer et la source ayant généré cette action. Si l'analyse en cours de jeu a été activée, le résultat de l'analyse de l'action est retourné sous la forme de chaînes de caractères. Cas de la trace d'une action d'un **ComponentMonitoring** :

```
// Récupération du composant de suivi à partir d'un GameObject
ComponentMonitoring cm = go.GetComponent<ComponentMonitoring> ();
// Enregistrement d'une action d'un ComponentMonitoring réalisée par le joueur
string[] result = MonitoringManager.trace (cm, "gameActionName",
                                           MonitoringManager.Source.PLAYER);
```

Cas de la trace d'une action sur une famille :

```
// Définition d'une famille
Family myFamily = FamilyManager.getFamily (...);
// Enregistrement d'une action d'une famille réalisée par le joueur
string[] result = MonitoringManager.trace (myFamily, "gameActionName",
                                           MonitoringManager.Source.PLAYER);
```

9.4.3 Fonction « getMonitoringById »

La fonction **getMonitoringById** permet de récupérer un **ComponentMonitoring** à partir de son id. les id des **ComponentMonitoring** sont visibles dans la fenêtre d'édition du suivi (voir Figure 11) :

```
// Récupération du composant de suivi à partir de son id
ComponentMonitoring cm = MonitoringManager.getMonitoringById (0);
// Enregistrement d'une action d'un ComponentMonitoring réalisée par le joueur
string[] result = MonitoringManager.trace (cm, "gameActionName",
                                           MonitoringManager.Source.PLAYER);
```

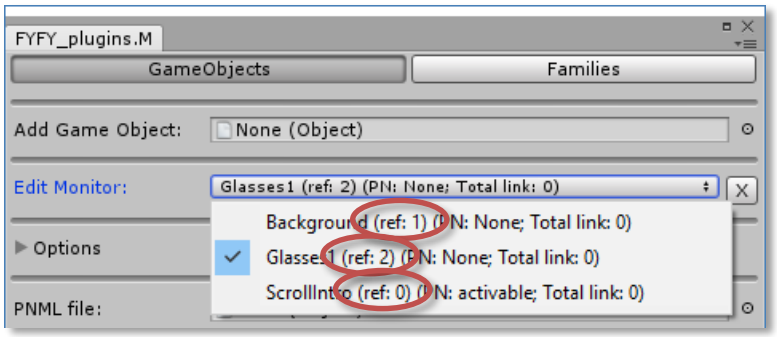

Figure 11

9.4.4 Fonction « getNextActionsToReach »

La fonction **getNextActionsToReach** permet d'interroger le module d'analyse (Laalys) pour obtenir la liste des X prochaines actions à réaliser en vue de rendre possible une action ciblée. Comme la fonction trace, cette fonction existe en deux versions. Une première pour les GameObject suivis :

```
// Récupération du composant de suivi à partir d'un GameObject
ComponentMonitoring cm = go.GetComponent<ComponentMonitoring> ();
// Récupération des 3 prochaines actions à réaliser en vue de rendre possible
// l'action nommée "target"
List<KeyValuePair<ComponentMonitoring, string>> result =
                    MonitoringManager.getNextActionsToReach (cm, "target", 3);
```

Une seconde pour les familles suivies :

```
// Définition d'une famille
Family myFamily = FamilyManager.getFamily (...);
// Récupération des 3 prochaines actions à réaliser en vue de rendre possible
// l'action nommée "target"
List<KeyValuePair<ComponentMonitoring, string>> result =
                MonitoringManager.getNextActionsToReach (myFamily, "target", 3);
```

9.4.5 Fonction « getTriggerableActions »

La fonction **getTriggerableActions** permet d'interroger le module d'analyse (Laalys) pour obtenir la liste de toutes les actions suivies disponibles :

```
// Récupération de toutes les actions suivies actuellement possibles
List<KeyValuePair<ComponentMonitoring, string>> result =
                                        MonitoringManager.getTriggerableActions ();
```