

# La Lib RRObjective

---

## 1) Introduction

La lib RRObjective permet de gérer les objectifs à atteindre dans un niveau. Elle a été conçue de façon générique pour s'intégrer dans tous types de projets. On peut également créer des nouveaux objectifs plus dédiés à un projet comme c'est le cas ici avec l'objectif de suivi de courbes.

Son intégration se déroule en trois temps, d'abord rajouter un RRObjectiveManager dans la scène, et y rajouter tous les objectifs que l'on souhaite dans le niveau.

Ensuite, il faut rajouter le script RRObjectivePlayer sur le prefab de joueur qu'on va instancier par joueur, ce script permet de s'enregistrer et de récupérer un identifiant de joueur qui nous permettra de faire progresser les objectifs de ce joueur.

Enfin dans nos scripts de projets, il suffira de dire que tel joueur évolue sur tel objectif.

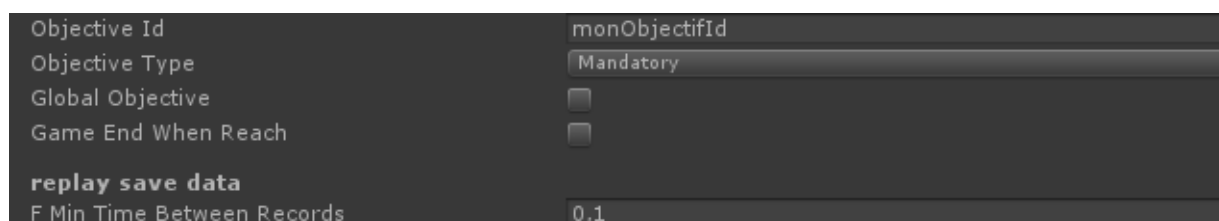
## 2) Dans Unity

### A) Ajouter RRObjectiveManager

Ajouter RRObjectiveManager à votre gameObject, ce script n'a aucune variables, ce qui va importer ce sont les objectifs que vous allez rajouter à ce gameObject.

### B) Ajouter des Objectifs

A la suite du Manager nous allons rajouter des objectifs, tous les objectifs ont des paramètres globaux :



Objective Id : Très important, c'est le nom par lequel vous retrouvez cet objectif dans le jeu, lorsque vous voudrez le mettre à jour ou enregistrer une callback

Objective Type : le type de l'objectif, peut être requis pour la victoire « Mandatory », peut être optionnel si on veut ajouter des objectifs secondaires, ou peut être éliminatoire (plus de temps, plus de vies ...).

Global Objective : signifie si l'objectif est commun à tout le monde (un temps par exemple) ou si il est dédié à chaque joueur (collecte de pièces d'or).

Game End When Reach : pour mettre fin à la partie une fois l'objectif atteint, on vérifiera à ce moment là si tous les objectifs requis sont remplis ou non, on peut causer une défaite en accomplissant un objectif de victoire.

FMinTimeBetweenRecords : Sers pour l'enregistrement des données, dans un but de proposer un replay on va sauvegarder chaque évolution de l'objectif. Cependant pour éviter d'utiliser trop de mémoire il est possible de définir un temps minimum entre 2 enregistrements. Enregistrer une position 60 fois par seconde serait trop gourmand par exemple.

## C) Les différents types d'objectifs

### ObjectiveInt :

Permet de définir des objectifs jouant avec des entiers, On doit récupérer X coffres ... Nous aurons 3 paramètres à compléter

Initial Value	0
Goal Value	1
Is Incremental	<input checked="" type="checkbox"/>



La valeur initiale, la valeur à atteindre, et enfin si on incrémente ou décrémente (si le joueur doit gagner ou se débarrasser d'un élément)

### ObjectiveFloat :

La même chose que précédemment sauf qu'il s'agit d'un nombre flottant. Avec les mêmes paramètres à renseigner.

### ObjectivePath :

Celui-ci est particulier car il est déjà plus dédié à notre jeu. On doit lui renseigner les couloirs de navigations de la scène, et lors de l'enregistrement du joueur, il affectera un de ces couloirs à un script PathFollow attaché à l'objet de notre player.

▼ Halls	
Size	1
Element 0	 Lane0 (PathHall) 
Goal Value	1

En ce qui concerne la GoalValue, 1 signifie la fin de la courbe, si vous avez un circuit fermé vous pouvez indiquer le nombre de tours.

**Il est à noter que les Objectifs manipulent des objets ObjData, généralement chaque Objectif aura sa propre data et convertira cet objet dans le type souhaité.**

### 3) En script

Maintenant que notre scène est configurée, nous devons créer nos scripts qui vont communiquer avec ces objectifs.

Notre ObjectiveManager est un singleton, ce qui signifie que nous pouvons interagir de n'importe quel endroit du script par l'appel `RRObjectiveManager.instance` à partir de là nous avons accès à plusieurs fonctions :

```
public void Pause()
public void Play()

public void AddObjectiveUpdateDlg( string sObjId, Objectives.OnObjectiveUpdate OnObjectiveUpdate )
public void SubObjectiveUpdateDlg(string sObjId, Objectives.OnObjectiveUpdate OnObjectiveUpdate)

public void AddObjectiveReachDlg(string sObjId, Objectives.OnObjectiveReach OnObjectiveReach)
public void SubObjectiveReachDlg(string sObjId, Objectives.OnObjectiveReach OnObjectiveReach)

public void GetInitialAndTargetValue(string sObjId, out Objectives.ObjData initial, out Objectives.ObjData target)
public void UpdateObjective( string sObjId, int nPlayerId, Objectives.ObjData add )
```

Pause et Play servent à activer/désactiver le module.

Nous avons ensuite des fonctions pour enregistrer et désenregistrer des callbacks, cela nous permet de mettre à jours des feedbacks.

Nous avons une fonction pour récupérer les valeurs initial et d'objectif, et enfin une fonction pour incrémenter notre objectif.

Exemple :

```
objectiveManager.AddObjectiveUpdateDlg("Coins", OnCoinsUpdate);
objectiveManager.AddObjectiveReachDlg("Path", OnPathComplete);
```

Nous enregistrons une fonction lorsque nous mettons à jour nos pièces, et une autre lorsque le chemin est complété.

Ces fonctions :

```
private void OnCoinsUpdate(int playerId, string sObjectiveId, Objectives.ObjData oldValue, Objectives.ObjData newValue)
{
    ObjectiveInt.IntObjData intObjData = (ObjectiveInt.IntObjData)newValue;
    m_coinValueText[playerId].text = intObjData.m_value.ToString() + "/" + m_nCoinsTarget;
}

private void OnPathComplete(int playerId, string sObjectiveId )
{
    m_lapValueText[playerId].text = "DONE";
}
```

Dans la mise à jour des pièces, on convertit notre ObjData en intObjData pour récupérer la valeur de pièces, et on actualise le hud.

Pour la complétion du chemin nous actualisons un message dans le Hud.

Pour l'update nous avons besoin de savoir l'Id du joueur, on peut rajouter dans notre script une référence au RRPlayer comme ici pour le coinsCollector

```

public class CoinsCollector : MonoBehaviour
{
    [SerializeField]
    RRObjecivePlayer objectivePlayer;

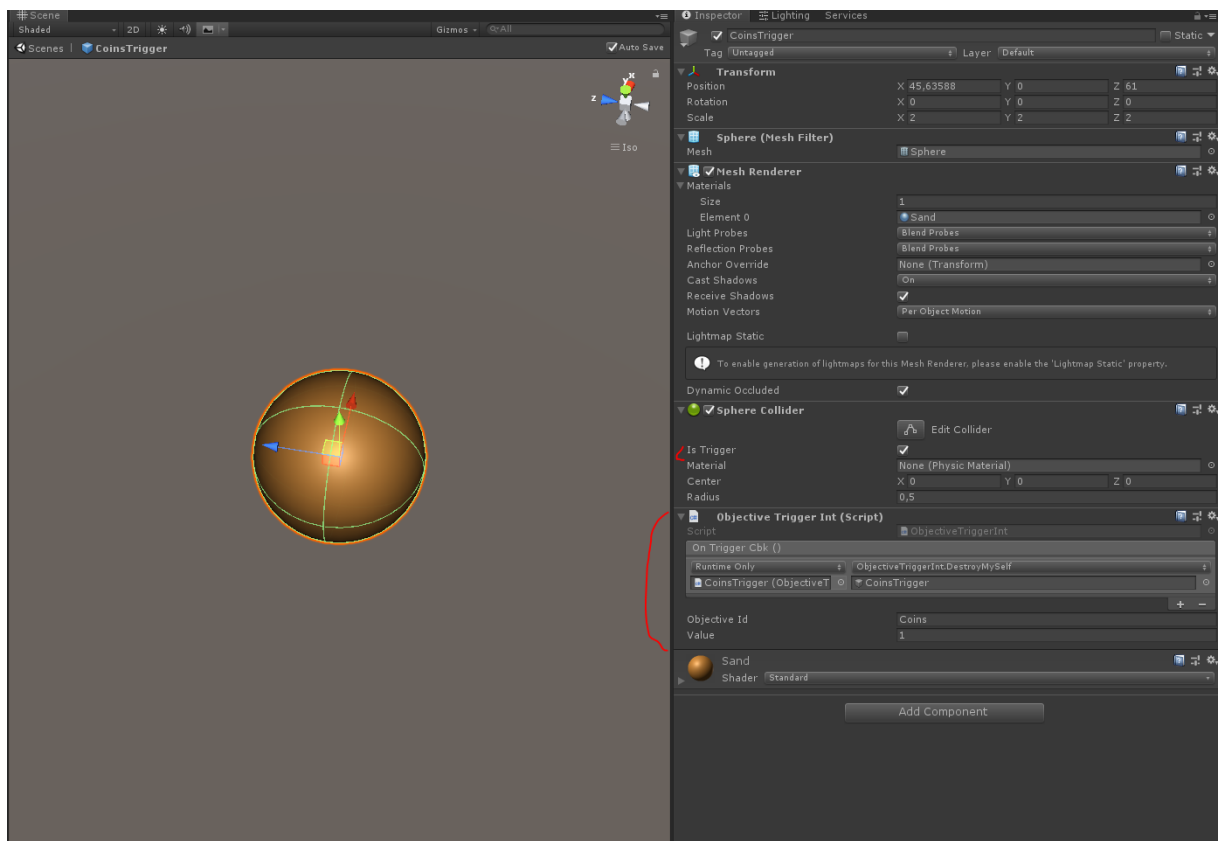
    private void OnCollisionEnter(Collision collision)
    {
        if (collision.gameObject.tag != "Player")
        {
            GameObject.Destroy(collision.gameObject);
            RRObjeciveManager.instance.UpdateObjective("Coins", objectivePlayer.playerId, new ObjectiveInt.IntObjData(1));
        }
    }
}

```

Lorsque nous collectons une pièce (ici lorsque nous rentrons en collision avec autre chose qu'un joueur) Nous mettons à jour l'objectif « Coins », pour le joueur qu'on référence, et cela de 1.

## 4) Les classes Triggers Update

Dans un souci de faciliter l'intégration nous avons créé des scripts pour mettre à jour les objectifs une fois ces objets touchés. Nous avons une classe mère « ObjectiveTrigger » si on veut rajouter ultérieurement d'autres objectifs, et nous avons les 2 scripts opérationnels [ObjectiveTriggerFloat](#) et [ObjectiveTriggerInt](#). Ces scripts prennent comme paramètres les événements à appliquer une fois ce trigger touché (par exemple la destruction de l'objet, le script contient une fonction DestroyMySelf prête à l'emploi). Mais surtout on doit renseigner l'id de l'objectif à mettre à jour ainsi que son incrément.



Dans notre exemple voici le prefab pour les coins. On le note en trigger et on lui ajoute l'ObjectiveTriggerIntScript. Dans les callbacks on lui dit de se détruire lui-même, et on indique qu'il faut incrémenter l'objectif Coins de 1.

## 5) Pour le multijoueur

Dans cette version le module n'est pas multi-joueurs online, pour cela nous préconisons 2 tâches :

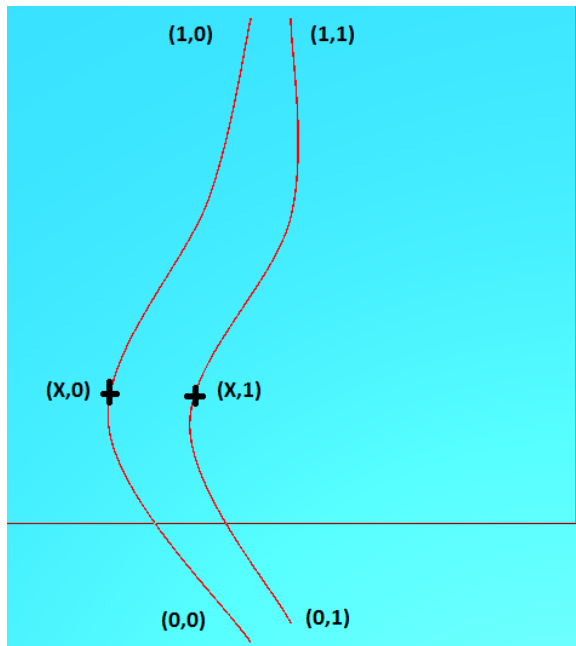
- Lors de l'enregistrement d'un joueur, il faudrait que son id soit une variable synchronisée, qu'il ait le même id sur chaque client
- Le module est conçu pour récupérer l'événement en un point et de dispatcher l'information. On peut donc avoir sur chaque client un module avec les mêmes callbacks de diffusion, Il faudra juste diffuser l'appel de l'événement sur le réseau, on peut comme pour le replay rajouter un temps entre 2 diffusions sur le réseau afin de réduire l'usage de la bande passante.

## 6) Le suiveur de Courbe

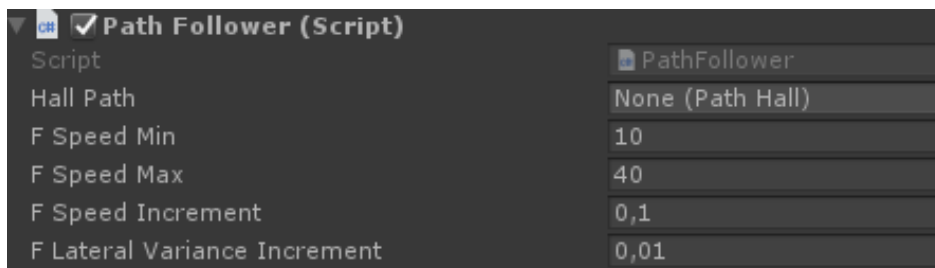
Le principe de ce script est de se situer dans un couloir, le couloir est défini par 2 courbes. On va avoir une valeur de progression, ainsi qu'une valeur de latéralité. Ces deux valeurs varient entre 0 et 1.

Une valeur de progression de 0 sera le début du couloir, et de 1 la fin du couloir

Une valeur de latéralité de 0 signifiera qu'on sera sur la courbe de gauche, et une valeur de 1 sur la courbe de droite



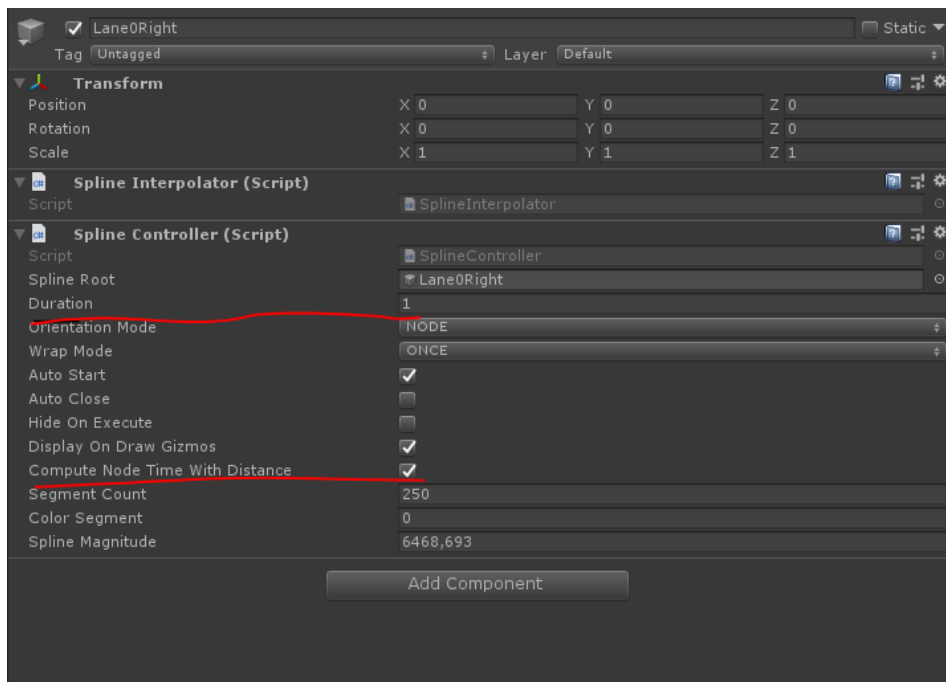
Vous devez rajouter sur votre objet joueur un script PathFollower. **Attention il doit être sur le même GameObject que le ObjectivePlayer !!** Le Path Follower Script prend plusieurs paramètres



On notera que les vitesses sont en unités monde, et que le moteur fera lui-même la conversion pour passer en valeur de progression entre 0 et 1, pour garder une cohérence lorsque l'on change de scènes (et de couloirs)

### La création d'un couloir :

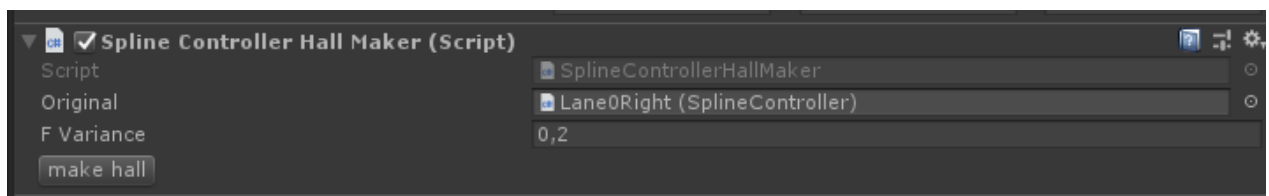
Un couloir est composé de 2 courbes, on va d'abord construire la première en mettant un composant Spline Controller sur un game Object, et ensuite créer les nodes enfants de ce game Object.



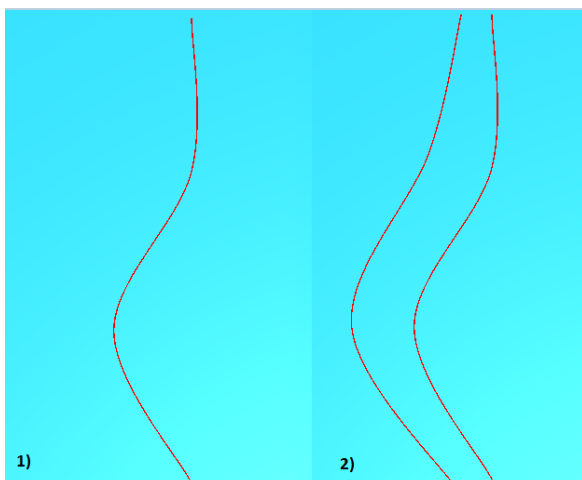
**ATTENTION ! Pour dessiner une courbe vous devez avoir au moins 2 positions (game object enfants) non confondus (pas la même position), sans ça vous n'aurez aucune courbe d'affichée.**

Pensez bien à mettre la duration sur 1, ainsi que de cocher le Compute Node Time With Distance.

Une fois cette courbe éditée, rajoutez le script SplineControllerHallMaker, référencez votre courbe dans le champ Original, réglez la variance (l'espace entre vos courbes) et cliquez sur make Hall

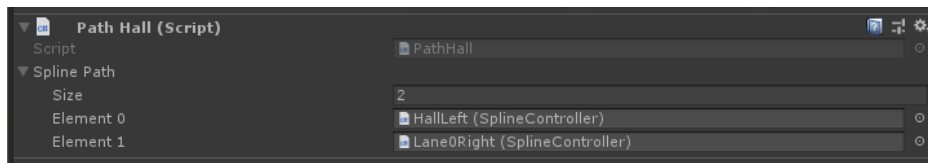


Vous devrez passer de l'image 1 à l'image 2



Vous pouvez rectifier les couloirs en manipulant les nodes enfants des SplineController.

Une fois votre couloir satisfaisant, vous devez rajouter un composant « Path Hall » et rajouter vos deux courbes dans son tableau de splines



Voilà votre PathHall est prêt pour l'objectif de suivi de chemin.

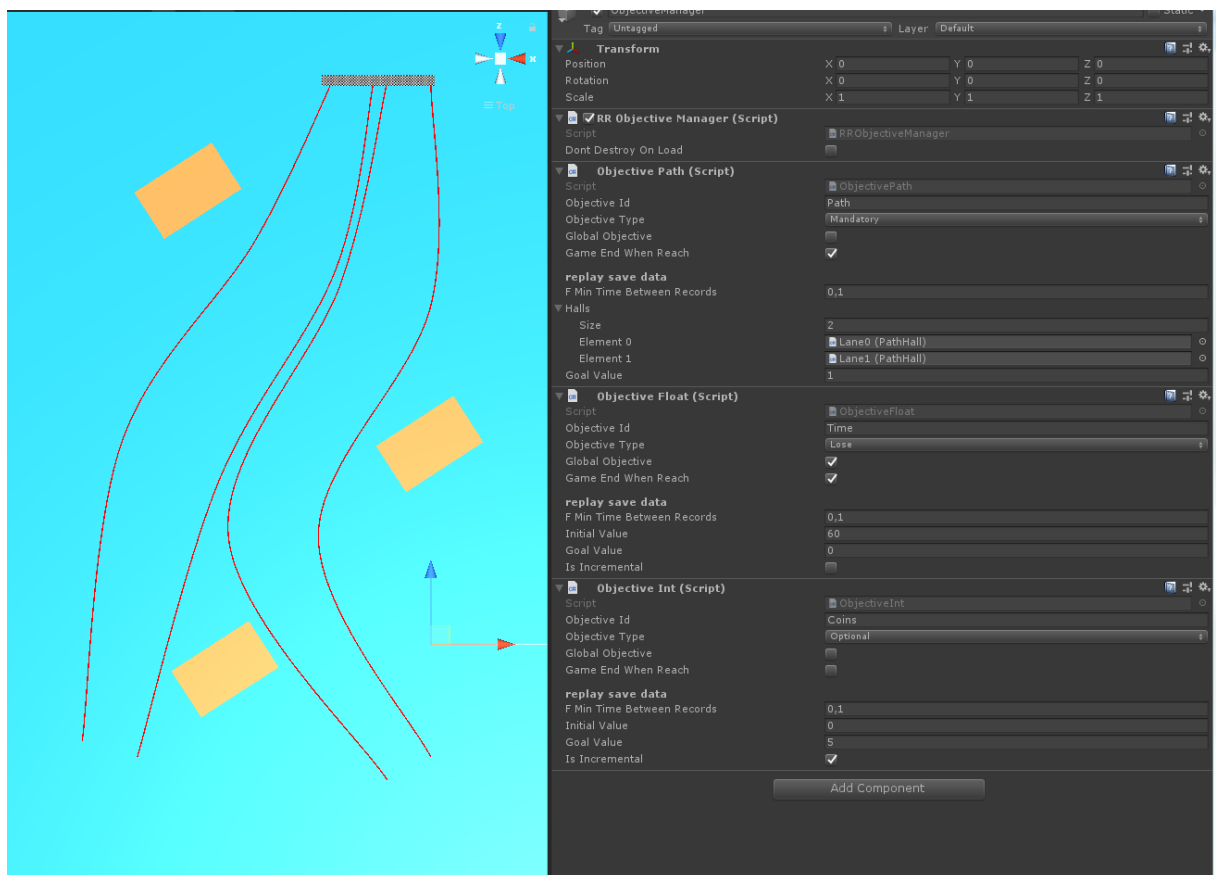
## 7) Les scènes d'exemples

La lib est présentée avec 2 scènes d'exemples, dans ces scènes les bateaux se déplacent avec les touches z,q,s,d pour le bateau de gauche et les touches directionnelles pour le bateau de droite.

Lors de l'exécution, on peut passer d'une scène à l'autre avec la touche « Esc »

### sampleLine :

Cette scène possède deux couloirs bien distincts où le but est d'arriver premier au bout de son couloir. On y a ajouté un objectif secondaire de ramassage de pièces. (Le finir ou pas n'aura pas d'incidence sur la victoire).



Voici son Objective Manager, ainsi que les 2 couloirs bien visibles sur la scène.

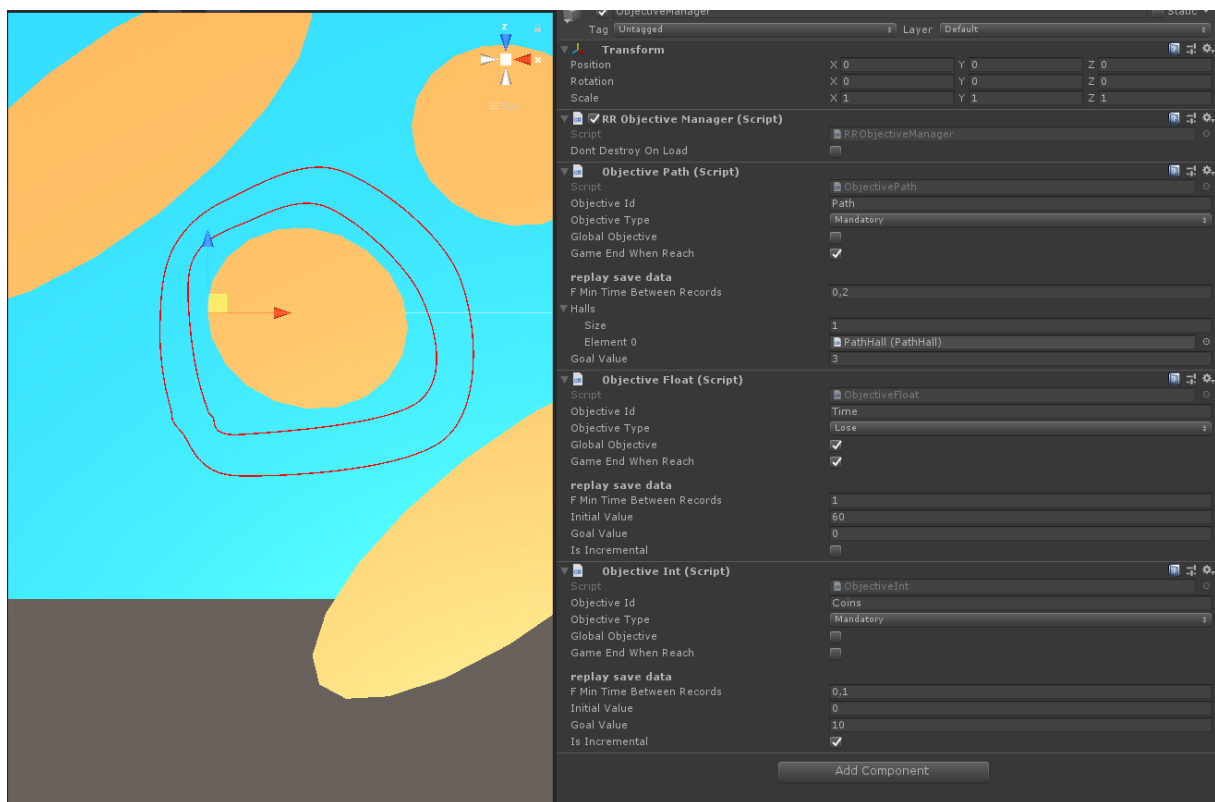


On y a ajouté 3 objectifs :

- Path qui est l'objectif de chemin, où on a référencé les 2 couloirs. Il est noté Mandatory (essentiel) et il met fin au jeu une fois atteint.
- Time : qui est l'objectif de temps, il part de 60 et finis à 0. C'est un objectif de défaite et il est global.
- Coins : Objectif optionnel qui est de ramasser 5 pièces.

### SampleCircle :

Cette scène possède qu'un seul couloir qui boucle, on peut donc y enchaîner les tours. Nous y avons mis l'objectif de collecte en requis pour la victoire. Ce qui signifie que si le joueur franchit la ligne avant d'avoir ramassé les pièces, il perdra.



Voici son Objective Manager, ainsi que son couloir unique.

On y a ajouté 3 objectifs :

- Path qui est l'objectif de chemin, où on a référencé le couloir. On a mis un objectif de 3, ce qui signifie 3 tours.
- Time : qui est l'objectif de temps, il part de 60 et finis à 0. C'est un objectif de défaite et il est global.
- Coins : Objectif requis pour la victoire, à noter qu'il ne met pas fin à la partie. D'un point de vue logique il ne doit y avoir qu'un seul objectif de victoire qui met fin à la partie, il est très rare de réaliser 2 objectifs simultanément.