RPG Tools & ScriptableObject

Sommaire

I/ Introduction

Définition : Les scriptableObject sont des class permettant de stocker de grand quantité de donnée et ou il n'est pas besoin de les associer à un gameobject.

Objectif de la R&D : Manier les Scriptable Objet de telle façon à ce que l'utilisateur du tools puissent réaliser un RPG de A à Z. Cela comprend une recherche à la fois sur les scriptable Object mais également sur les fonctionnalité d'édition de Unity 3D.

Source
https://unity3d.com/learn/tutorials/modules/beginner/live-training-archive/scriptable-objects

II/ Editeur :

Les class GUI, GUILayout, EditorGUI, EditorGUILayout sont utilisés pour dessiner des widget à l'écran, ces widget peuvent être ordonnées en fonction de la class choisi. Les class Layout vont ordonner les widget automatiquement alors que les autre devront prendre en paramètre une position et une taille prédéfinie. Les class Editor sont une surcouche au class non Editor.

Les different champs (write/read) et forme (read) :

- Text
- Bool (toggle)
- int
- float
- Object (GameObject, Texture, ScriptableObject).
- PopupEnum
- Popup (List de string)
- Box
- …

Signature :

string TextField(Rect position, string label, string text, GUIStyle style)
string TextArea(Rect position, string text, GUIStyle style)
bool Toggle(Rect position, string label, bool value, GUIStyle style)
int IntField(Rect position, string label, int value, GUIStyle style)
float FloatField(Rect position, string label, float value, GUIStyle style)
Object ObjectField(Rect position, string label,Object obj, Type objType, bool allowSceneObjects)

Class GUI

BeginGroup() / EndGroup() : Definie une Zone ou l'utilisation d'autre element GUI ou GUILayout est positionner en fonction de celui ci.

BeginScrollView / Endscrollview

Class GUILayout

BeginArea / EndArea : Definie une Zone ou l'utilisation d'autre élément GUILayout est positionné en fonction de celui ci.

Class GUILayout

BeginHorizontal / EndHorizontal : Positionne un widget de manniere horizontal

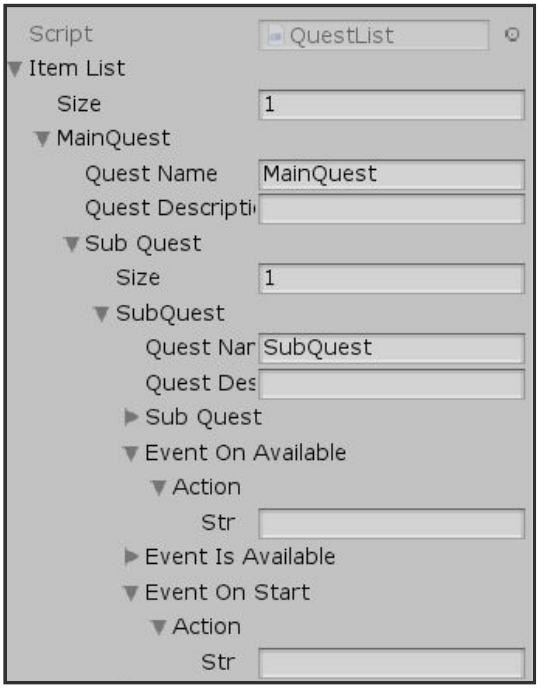
BeginVertical / EndVerticall : Positionne un widget de manniere Vertical
BeginScrollView / Endscrollview.

III/ Scriptable Object

A/ Contrainte des scriptable Object

Les scriptable Object permette beaucoup de chose mais comme les autre asset les scriptable Objet ont besoin d'être sérialiser et deserializer. Ainsi cela pose divers problème pour certain utilisation.

1) **Interdépendance des classe :**
   La première se situe au niveau des des interdépendance des class qui pose une limite à la sérialisation de Unity qui ne permet pas de gérer ce cas précis. En effet Unity pose une limite au niveau de la sérialisation profonde d'une classe (Max : 7)...



Par exemple ici on a une list de quest de Type Quest qui continent elle même une liste de SubQuest.

Ansi Unity ne pourra pas serializer/deserializer une fois la limite atteinte et nous renvoie systématiquement cette avertissement.

2) **Polymorphisme :**
   La seconde touche le Polymorphisme qui devient obsolète sur un scriptable Object. En effet Unity ne peut gérer le polymorphisme et l'héritage sera perdu au moment de la sérialisation car unity n'arrivera pas à reproduire l'objet sous forme polymorphique.

3) Instance

C'est dernière utilisation peut être aussi liée au 2 premier. En effet lorsque nous voulant instancier un object en utilisant le mot clé new unity ne pourra avoir la connaissance de cette opération ainsi tout les variable présente dans une classe ne seront que des variable qui ne font pas référence vers un autre. Ainsi toute comparaison entre deux objets devient alors impossible.

Exemple :

```
class ConnectionPoint{}
class Node
{
    public ConnectionPoint point;
    public Node(ConnectionPoint _point)
    {
        point = _point;
    }
}
class Connection
{
    public ConnectionPoint point;
}
class Container
{
    Container()
    {
        connection = new Connection;
        connection.point = new ConnectionPoint();
        Node node = new Node(connection.point);
    }
    Connection connection;
    Node node;
}
```

Dans cette exemple on peut voir que la class **Container** créer une **Connection** et créer une **ConnectionPoint**.

Puis créer une **Node** en lui passant en paramètre un **ConnectionPoint.**

Cependant pour **Unity** c'est variable ne sont pas des référence comme le peuvent etre les **GameObject** mais de simple variable.

Ainsi au moment de la désérialisation **Unity** dans tout les classe va recréer la classe **ConnectionPoint** mais à aucun moment l'un de fera référence à l'autre.

C'est tout comme pour le **polymorphisme** les information de la class seront perdu et **Unity** ne gardera que les information du type de la classe indique au moment de la déclaration.

B/ Solution

Bien entendu au vu de nos recherche on a pu trouvais divers moyen pour contourner ce problème.

1) Système d'id.

Pour contourner le problème des interdépendance on peut utiliser une système d'id ou on remplace la liste de class par une liste d'id.

```
class QuestExample
{
    List<int> subQuestID;
}

QuestExample GetQuest(int id)
{
    return QuestExampleList[id];
}
```

Ainsi ici pour accéder au sous quête on passera par un getter qui prend en paramètre un id.

Cette fonction retournera la quête qui sera déterminé par une identifiant représentant la position de la liste de quête.

2) Scriptable Object

Et oui les Scriptable Object sont eux même une solution a tout les problème cité plus haut.
En effet les Scriptable Object étant considéré comme des asset peuvent etre recuperer par unity lors de la désérialisation. On va pouvoir ainsi sauvegarder nos objets pour cela il y a divers chose a faire.

Tout d'abord il faut faire hérité la class qu'on veut sauvegarder par "ScriptableObject". Les class enfant pourront alors être hérite de la classe de base et seront aux aussi considérer comme des scriptable Object.

Voici le processus :

- ScriptableObject.CreateInstance<Type>()

Fonction template Permettant de créer une instance de Scriptable Objet en fonction de son Type.
Il faut alors utiliser une fonction d'initialisation à la place des constructeur si il y a des paramètre, qui en soit n'est pas un gros problème mais cela casse un peu le principe des constructeur.

Pour changer le nom de l'objet il faut passer par la variable name du scriptableObject.

ScriptableObjet.name = "Main Quest";

- AssetDatabase.AddObjectToAsset(Object, ScriptableObjetAsset);
  Ensuite il faut l'ajouter à l'asset dans laquelle on veut le sauvegarder.

- AssetDatabase.ImportAsset(AssetDatabase.GetAssetPath(quest));

Pour terminer il ne faut pas oublier d'importer pour qu'il soit bien modifier sinon le nom de l'objet sera sauvegarder seulement lorsque qu'on save le projet Unity.

Affichage de l'Asset :

Projet RPG Tools Editor

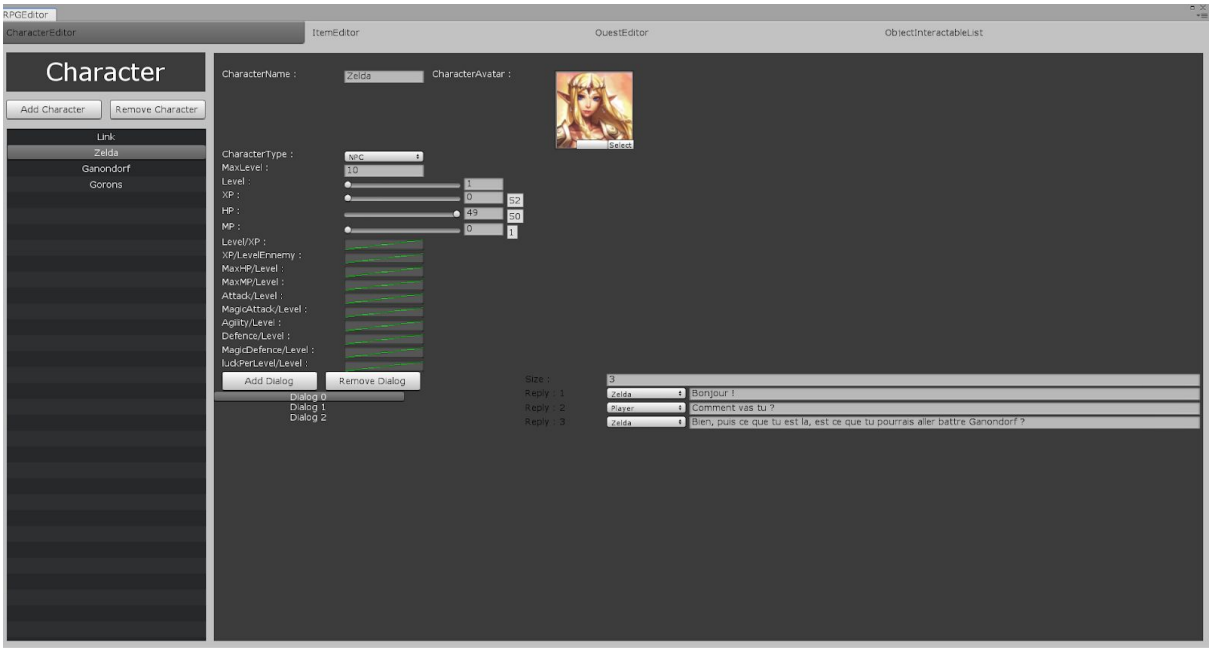
A/ Introduction

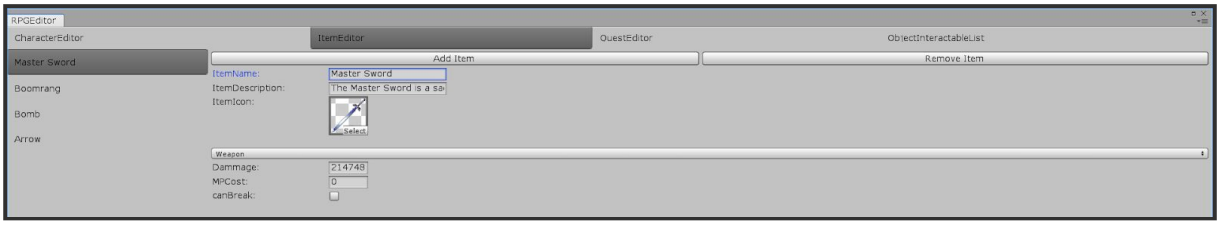
Maintenant que la recherche de R&D a été expliqué, on vas vous parler du petit projet qu'on a pu réaliser.

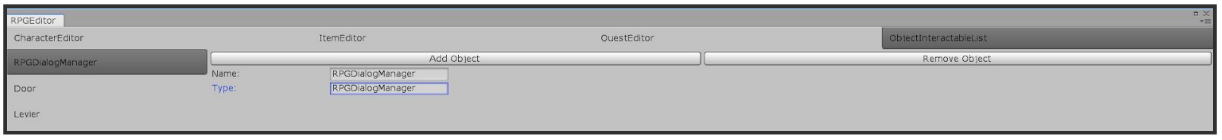
Le projet ce compose de la manière suivante :

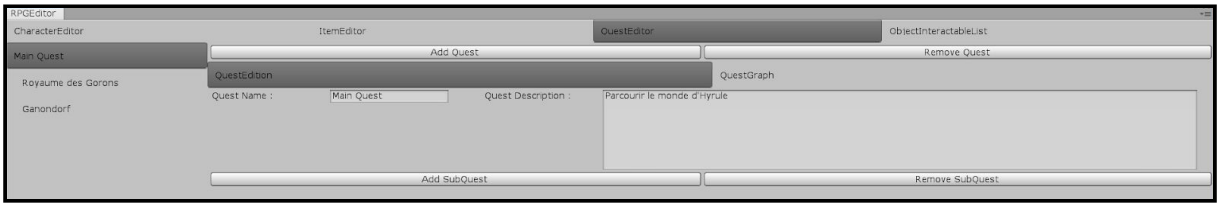
- Un editeur RPG Tools Window (Custom Editor)
- Des scriptable Object sur différent entité
- Une interface graphique composée de Node.

Appercue de la window RPG Tools :







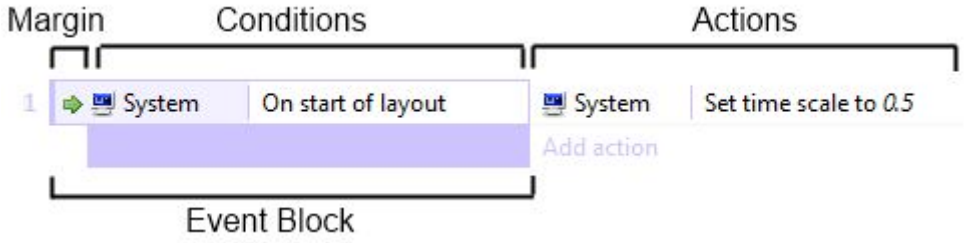


Comme on peut le voir cette editeur permet de pouvoir gérer les personnage, les item, les objets pouvant interagir avec le joueur (porte, levier...).

Le point important de ce Tools sont le système de quête mise en place, ce sont ces quête qui permettent de gérer entièrement le jeu, comme la mise en place de dialogue en fonction de la quête en cour.
Pour pouvoir gérer cela de manière simple et compréhensible par tout le monde, on a mis en place un graph composé de node permettant de gérer le jeu. Ce graph se base sur un système similaire au blueprint de Unreal Engine 4.

C/ System d'action et d'évent

Le système implémenté pour pouvoir gérer l'état du jeu est un système d'évent et d'action. Ceux system est inspiré du moteur de jeu construct



Event :
Une event est une condition qui sera vérifié en jeu, si cette event vraie alors on appellera l'action associer.

```
[System.Serializable]
public class EventRPG : ScriptableObject
{
    public EventRPG nextEvents = null;
    public MyAction action = null;

    public List<Parametre> parametres;

    public void Init()
    {
        parametres = new List<Parametre>();
        parametres.Add(new Parametre(typeof(string)));
        parametres.Add(new Parametre(typeof(string)));
    }

    public virtual void Update()
    {
        object entity = RPGManager.Instanse.GetEntity(parametres[0]);
        if(entity != null)
        {
            Type type = entity.GetType();
            MethodInfo methodInfo =  type.GetMethod(parametres[1].parameterString);
            if((bool)methodInfo.Invoke(entity, GetParametreArray(parametres)))
            {
                if (action != null)
                    action.Update();
            }
        }

        if (nextEvents != null)
            nextEvents.Update();
    }
```

Composition Event :

- Un objet qui peut soit être une quête, soit un character, soit un objet interactable.
- Une fonction retournant un booléen
- Une liste de paramètre
- Event suivant
- Une action

Action :
Une action est une fonction qui sera exécuté si L'event qui le possèdent est vraie.

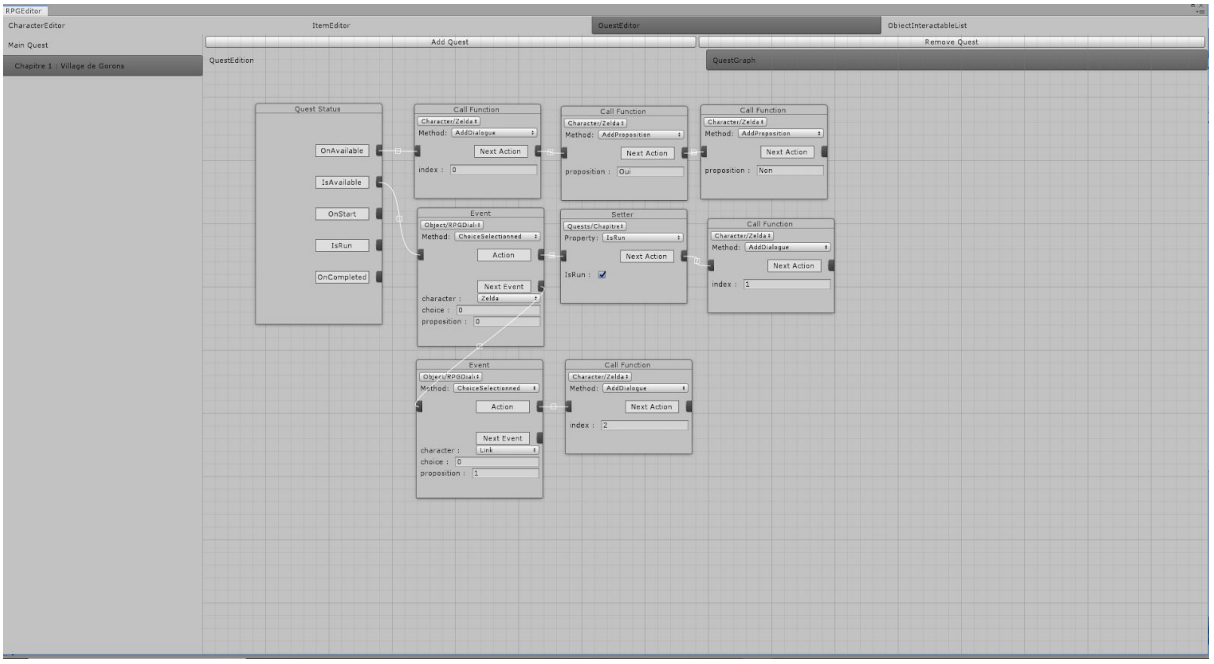
Composition Action :

- Un objet qui peut soit être une quête, soit un character, soit un objet interactable.
- Une fonction sans retour
- Une liste de paramètre
- Action suivant

Les fonction associer au Event et au Action sont appelés grâce à l'utilisation de la réflexion.

D/ Graph Editor

Un Graph est associé a chaque quête. C’est ici que vous pourrez définir les diferente event et action du jeu qui seront associés à la quête.



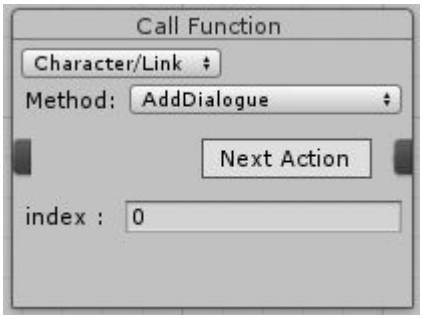
Un graph est composé d’une liste de node. C’est node peuvent être de quatre type.

1) Quest Status Node

La Node Quest Status permet de faire divers opération en fonction de l'état de la quête. Basiquement ce sont des sorte de delegate.

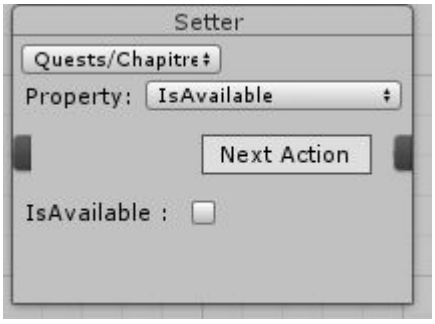
- OnAvailable désigner le faire qu’on rend un action disponible. Elle est specialement faire pour un npc qui donne des quête au joueur. En voyant cette dernière disponible il pourra la proposer au joueur.
- Is Available : Pareille que la précédente mais est activer en boucle.
- OnStart : Lors du commencement d’une quête.
- IsRun : Quête en cours
- OnCompleted : Quête Finit.

2) Call Function Node



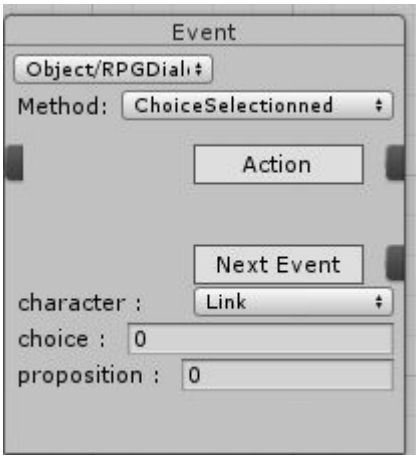
La Node Call Function permet Comme son nom l'indique d'appeler une action. Elle peut être précédé par un event ou une action. Et peut être suivi par une autre action qui sera appelé juste après cette dernière.
Pour appeler un fonction, il faut d'abord choisir l'objet qui contient la fonction, puis la méthode parmi la liste et enfin indiquer les paramètre de cette dernière.
Si une fonction contient des paramètre non gerer par le tool qui ne l'affichera pas.

3) Setter Node



La Node Setter permet de setter des valeur au different properties.

4) Event Node



La Node Event permet Comme son nom l'indique Est une condition. Elle peut être précédé par une autre condition. Action désigne l'action à appeler si la condition est vraie et next event sera appelé dans n'importe quelle cas qu'elle soit vraie ou non.

E/ Runtime :

Concernant les acteur en jeu, elle sont gérés par une classe de type RPG Component qui une fois la fonction Awake appelée, l'ajouter au RPGManager classe permettant de stocker tous les different acteur d'un RPG(item,object interactable, Quest).

Le Quest Manager s'occupera quant à lui s'occupera de vérifier les different event et action en fonction de l'état des quêtes.

Les classe RPGCharacter , RPGItem et RPGObjectInteractable herite de RPG Component et on chacun une variable dans leur type respectif qu'on devra setter dans l'éditeur. (Character, Item,ObjectInteractable).

Ainsi les fonction dans le graph seront pris directement de ces class.
Pour les objets Interactable il faudra indique le type de la classe en jeu dans l'éditeur d'objets.

IV/ Manuel D'utilisation

Tout d'abord rendez vous dans l'onglet window d'unity puis RPG Editor.

Dans Character vous pouvez ajouter des personnage et modifier leur statistique ainsi que leur ajouter une liste de dialog.
Un character peut être soit un NPC, soit un Player ou soit un ennemies.

Pour l'associer en jeu vous devrez créer un GameObject lui ajouter component de type RPGCharacter(RPGPlayerCharacter ou RPGNPCCharacter en fonction du type de personnage).
Ensuite associez lui un character le champs character du component.

Les item sont un peu similaire au niveau de leur association en jeu. Le type sera alors RPGItem.
Ensuite associez lui un item le champs item du component.

Les ObjetInteractable pour eux sont un peu plus particulier car en effet ce sont des objets qui peuvent être de différent type (door, manager, spawner, collider).

Vous devrez alors faire heriter votre nouvelle classe de RPGObjectInteractableComponent. Et devrez indiquer le type dans les propriétés de votre objet dans l'onglet ObjectInteractable du RPGEditor pour qu'il soit reconnaissable dans le graph pour accéder au diferente fonction.

Ensuite associez lui un ObjectInteractable le champs ObjectInteractable du component.

Pour éditer le graph de quête il vous suffira des faire un click droit sur le graph, un menu apparaîtra et vous pourrez sélectionner le type de node que vous voulez ajouter.

Pour les relier il vous suffira de cliquer sur les point de connection et de les relier au node que vous voulez connecter.

Attention, vérifier bien de connecter les bonne node entre elle.

Quest Status -> Event ou Action.
Event <- Event -> Action ou Event
Event ou Action <-Action -> Action.
Et pour supprimer une node clic droit sur la node et sélectionne remove dans le menu qui apparaît.