Mode d'emploi - Partie développeur

13 mai 2018

1 Introduction

Ce document vous permettra d'utiliser l'interface de ce logiciel, ainsi que tous les scripts qui lui sont attachés, de manière intuitive.

1.1 Interface graphique

La première partie de ce document se penchera sur l'utilisation de l'interface graphique, disponible directement en jeu. Nous passerons en revue les fonctionnalités qui s'offrent à vous.

1.1.1 Menu principal

Nous verrons tout d'abord le fonctionnement et les possibilités présentes sur le menu principal du jeu.

1.1.2 Editeur de comportement

Puis nous nous attarderons sur l'éditeur de comportement, et toutes les fonctionnalités indispensables à la création d'équipes, et de leur comportement.

1.2 Scripts

Nous irons ensuite plus en profondeur, et nous intéresserons aux scripts permettant le fonctionnement de l'interface dans sa globalité. Nous verrons, dans l'ordre, les scripts :

— du menu principal
— de l'éditeur de comportement

2 Interface graphique

L'interface graphique vous permettra de naviguer à travers les menus, de modifier des options inhérentes au jeu, de créer de nouvelles équipes, et surtout de modifier le comportement de chacune de leurs unités. Commençons par le commencement, avec le détail des fonctionnalités du menu principal.

2.1 Menu principal

Le menu principal donne accès aux options majeures du jeu.

- Lancer une partie. Cela lancera une partie, avec les équipes désignées, et les paramètres enregistrés.
- Accéder à l'éditeur de comportement. Cela ouvrira l'écran d'éditeur de comportement, que nous détaillerons plus tard.
- Modifier les paramètres du jeu. Permet d'accéder aux paramètres du jeu, tels que le réglage du volume, le choix de mode de jeu, la langue, etc...
- Désigner les équipes des joueurs. Permet de choisir quelles équipes participeront au combat.
- Les scores. Affiche directement le score et l'ELO de l'équipe concernée.
- Quitter le jeu.

2.2 Éditeur de comportement

L'éditeur de comportement met en place de nombreuses fonctionnalités indispensables au jeu. Il se décompose en cinq parties majeures.

- La "ToolBox". Permet la création d'un nouveau comportement, le chargement du comportement d'une unité sélectionnée, la sauvegarde d'un comportement, l'annulation d'une suppression, la reconstitution d'une annulation et le retour au menu principal.
- Les propriétés des unités. Partie purement informative, qui vous affichera le modèle de l'unité sélectionnée, ainsi que ses statistiques.
- La sélection d'équipes et d'unités. Ces deux listes vous permettront de choisir sur quelle équipe vous allez travailler, et sur laquelle de ses unités.
- Liste des pièces. Cette liste contient tous les éléments vous permettant de créer un comportement. Cinq types de pièces se proposent à vous :
  - Contrôles : La pièce permettant de contrôler les choix pris, sous la forme d'un "if".
  - Conditions : Tout ce qui permettra d'amorcer une action d'une unité.
  - Actions : L'action engendrée après validation de la condition, qui termine le tour de l'unité concernée.
  - Messages : Les différents messages qu'une unité peut envoyer aux autres unités de son équipe.
  - Action non terminale : Même comportement qu'une action, à la différence qu'elle ne termine pas le tour de l'unité.

3 Scripts

Nous allons maintenant aborder le cœur de l'interface, autrement dit, les scripts. Ceux sont eux qui font tourner toutes les fonctionnalités décrites précédemment.

3.1 Menu principal

— **Bouton Jouer (PlayButton.cs)**

Ce script a pour but de lancer la partie avec tous les paramètres définis par l'utilisateur(équipes choisies, nombre d'unités, volume du son, map et mode sélectionnés, etc...)

— **StartGame()** : Récupère le nombre de joueurs définis par le Dropdown NumberPlayersDropDown. Crée le path correspondant au mode de jeu sélectionné. Récupère la couleur correspondante au numéro du joueur, avec le nom de l'équipe, ainsi que les comportements de ses unités, puis ajoute cette équipe au script TeamManager().
Récupère les settings présents dans le GameObject GameManager, récupère le numéro de la scène à lancer, et la lance.

— **Bouton Paramètres (SettingsButton.cs)**

Ce script gère tous les paramètres présents dans la fenêtre Paramètres.

— **ApplySettings()** : Fonction qui appelle toutes les fonctions qui s'occupent de modifier les paramètres.

— **numberResources()** : Ne permet à l'utilisateur d'entrer une valeur uniquement comprise entre 1 et 40.

— **manageVolume()** : Récupère la valeur du slider, puis la passe à l'AudioSource, ainsi qu'au GameManager. Ce détail a son importance, puisque c'est ça qui va permettre de conserver la valeur d'une scène à l'autre.

3.2 Éditeur de comportement

3.2.1 ToolBox

— **Bouton Charger (LoadFile.cs)**

Ce script s'occupe de charger le comportement de l'unité actuellement sélectionnée dans le DropDown prévu à cet effet.

— **createBehaviorFromXML()** : La première chose à faire est d'effacer le comportement actuellement chargé, s'il existe. A l'aide du path, et du nom de l'équipe, nous allons pouvoir appeler une méthode présente dans l'interpréteur, pour transformer le fichier .xml correspondant à l'équipe courante, en un comportement, qui sera stocké dans un dictionnaire.
Nous allons maintenant récupérer la position de la pièce StartPuzzle, pour pouvoir placer les pièces correctement dans l'éditeur. Nous avons donc notre point de départ.

Il ne reste plus qu'à placer les pièces. Nous allons donc récupérer la liste d'instructions associé à chaque unité, et les placer dans leur ordre d'apparition. (Un "if" pour commencer, suivi de sa ou ses conditions, suivies d'une ou plusieurs actions)
Pour finir, on réinitialisera la scrollbar de l'éditeur, pour permettre à l'utilisateur de voir directement le début du comportement.

— **Bouton Sauvegarder (SaveFile.cs)**

Ce script permet de sauvegarder le comportement de l'unité sélectionnée, créé et/ou modifié dans l'éditeur.

- **createBehavior()** : Nous allons créer un comportement, copiant le comportement présent dans l'éditeur. Pour commencer, on va répertorier toutes les pièces "If" présentes. Ensuite, pour chaque pièce If de la liste, nous allons créer l'instruction comprenant les conditions et actions, grâce à l'interpréteur.
  Une fois chaque instruction créée pour chaque "If", nous pouvons appeler la fonction createXML().

- **createXML()** : Appelle la fonction behaviorToXml(), présente dans l'interpréteur, qui prend en paramètres le nom de l'équipe courante, le chemin où écrire le fichier .xml, le nom de l'unité, et la liste d'instructions correspondante.

— **Bouton Annuler (createPuzzle.cs)** Va permettre d'annuler la création ou la suppression d'une pièce. Lors de la création d'une pièce, cette pièce sera ajoutée dans une liste "listPieces", qui nous permettra de garder une trace de l'ordre dans lequel les pièces ont été créées.

- **Undo()** : Cette fonction requiert une liste de pièces non vide pour fonctionner, c'est donc ce que l'on va vérifier en premier. Si la liste "listPieces" n'est pas vide, alors on stocke la dernière pièce de cette liste dans une variable "pieceToUndo".
  Maintenant, il faut déterminer quelle action utilisateur nous devons annuler (création ou suppression).
  Si "pieceToUndo" est active, alors nous devons simuler sa suppression, en la passant inactive. Il faudra également l'ajouter dans la liste "recoverList", que nous verrons plus tard, pour reconstituer une annulation.
  Si, en revance, la pièce est inactive, il faut alors la faire réapparaitre à l'écran, en la passant en active.
  !!!!!!!!!!!!!!!!!!!!!!!A COMPLETER!!!!!!!!!!!!!!!!!!!!!!!

— **Bouton Redo (createPuzzle.cs)** Permet de restituer une annulation préalablement faite. Lorsqu'on clique sur le bouton Annuler, on va conserver l'action annulée dans une liste, qui nous servira donc à la restituer lors d'un clic sur le bouton Redo.

- **Redo()** : Cette fonction fonctionne grâce à l'utilisation de la liste

"recoverList", qui se remplit à chaque fois qu'une action est annulée. Elle est remplie de pièces qui ont été supprimées. Pour restituer l'annulation, il faut donc récupérer la dernière pièce de la liste, la passer en active(donc visible), et surtout ne pas oublier de la retirer de la liste.

3.2.2 Sélection des équipes / unités

— **Bouton Créer une nouvelle équipe (createTeam.cs)** Ce bouton sert à ajouter une nouvelle équipe à la liste des équipes déjà existantes. La création se passe dans une pop-up qui apparaît une fois le bouton cliqué.

— **validateName()** : Cette fonction a deux utilités. Tout d'abord, vérifier le nom de l'équipe entré par l'utilisateur. Nous n'acceptons que les caractères suivants : a-zA-Z0-9
Pour se faire, nous récupérons la chaîne de caractères entrée par l'utilisateur, puis nous allons, pour chaque caractère de cette liste, le convertir en sa valeur ASCII, et l'ajouter à la liste listInt. Le but est de vérifier quel caractère a été entré, et regarder son code ASCII, pour déterminer s'il est valide ou non. Ne surtout pas oublier de vérifier le cas du nom de l'équipe commençant par le caractère "espace" : (listInt[0] == 32).
Dès lors qu'un nom valide est entré, il faut vérifier que ce nom n'est pas déjà utilisé par une autre équipe. Pour cela, nous récuperons le nom de toutes les équipes présentes au path indiqué, puis comparons leur nom à notre nouvelle équipe.
Si l'équipe n'existe pas déjà, nous pouvons alors la créer. La création du fichier.wbt associé à l'équipe se passe dans l'interpréteur.
Finalement, il ne faut pas oublier d'ajouter cette nouvelle équipe à la liste des équipes disponibles dans le DropDown :

```
dropOption.Add(teamName);
teamDropDown.AddOptions(dropOption);
```

et de fermer la pop-up de création de nouvelle équipe.

— **Bouton supprimer une équipe (DeleteTeam.cs)** Ce bouton permet de supprimer l'équipe actuellement sélectionnée dans le DropDown des équipes, ainsi que les fichiers .wbt, .meta, et .stat associés.

— **Delete()** : Pour la suppression, nous ouvrons une pop-up qui demande une confirmation de l'utilisateur, pour s'assurer qu'il ne s'agissait pas d'un missclick.
Nous commençons par récupérer le nom de l'équipe concernée, puis nous allons parcourir la liste des fichiers .wbt, .meta, et .stat des équipes. Dès que nous trouverons les fichiers de la bonne équipe, nous les supprimons.

— **Updating()** : La dernière étape à ne pas omettre est de mettre à jour la liste des équipes, privée de l'équipe qui vient d'être supprimée. Nous parcourons donc la liste des fichiers des équipes, puis nous ajoutons chaque nom d'équipe dans une liste.
Nous réinitialisons les valeurs du DropDown, puis lui ajoutons la liste que nous venons de créer, qui est la liste des équipes mise à jour. La première équipe de la nouvelle liste deviendra alors l'équipe sélectionnée. Il nous reste à charger le comportement de son unité sélectionnée, pour ne pas obliger l'utilisateur à charger manuellement le comportement de l'unité de la nouvelle équipe courante.

3.2.3 Liste des pièces

La génération des modèles des pièces se fait de manière dynamique. Chaque type de pièce a un prefab associé.

— **ConditionEditorScript.cs** : Ce script contient une méthode pour chaque type de pièces disponibles (Conditions, Actions, etc...). Voyons de plus près l'une de ses fonctions (leur comportement est semblable).
  — **UpdateCondition()** : Nous commençons par récupérer une structure UnitPerceptAction, contenant le nom d'une unité, ainsi que toutes les conditions, actions, et messages inhérents à l'unité.
  A partir de là, nous pouvons créer, à l'aide du prefab correspondant, une pièce, avec le label contenu dans notre strucutre. Nous allons parcourir la structure pour créer toutes les pièces disponibles pour l'unité, et modifier leur placement, en leur ajoutant un vecteur, pour qu'elles ne se superposent pas.

```
new Vector2(0, -button.GetComponent<RectTransform>().rect.↩
    height)+ deltaVect;
```

  Dans cette fonction, nous nous occupons uniquement de la liste Conditions présente dans notre structure. Il y a une fonction pour chaque liste de la structure.

— **createPuzzle.cs** : Lors de l'appel à ce script, nous récupérons le prefab et le label associé au type de pièce concerné. Par exemple, pour une pièce "Conditions", nous récupérerons le prefab des pièces "<Conditions">, ainsi qu'un label nommé "PERCEPT". Le traitement de ce label se fait dans le script ConditionEditorScript.cs.
Nous plaçons ensuite la pièce créée
  — **create()** : Lors d'un clic sur le modèle de pièce que l'on veut ajouter au comportement courant, cette fonction est appelée. Une position par défaut est définie dans l'éditeur, qui déterminera où la pièce sera placée lors de sa création.
  A chaque pièce créée, nous l'ajoutons dans la liste listPieces, utilisée pour la fonction Annuler.

3.2.4 Editeur

Cette zone est la zone la plus importante de l'éditeur de comportement, puisque c'est elle qui permet la création d'un comportement pour une unité. Voyons comment tout cela est géré.

- **ManageDrapAndDrop.cs** : Ce script s'occupe de la gestion du déplacement des pièces, ainsi que de leur placement, sur une grille aimantée.
  - **OnMouseDrag()** : Une fois qu'un clic a été fait sur une pièce, cette fonction s'occupe de mettre à jour les coordonnées de la pièce, de telle sorte qu'elles soient égales à celles du pointeur de la souris. Pour valider les nouvelles coordonnées de la pièce, il faut appeler la fonction OnMouseUp().
  - **UpgradeGridPosition()** :
  - **OnMouseUp()** : Cette fonction appelle la fonction UpdateGridPosition(), puis attribue à la pièce courante sa nouvelle position. Une fois cela fait, il faut vérifier que la nouvelle position de la pièce est toujours un emplacement valide, d'un point de vue comportement. La fonction appelle pour ça le script StartPuzzleScript.cs, que nous verrons plus bas.
- **StartPuzzleScript.cs** : Ce script va nous permettre de vérifier si une pièce "If" est bien placé en dessous de la pièce Start.
  - **UpdateAllValidPuzzles()** : Cette fonction, appelée dans ManageDragAndDrop.OnMouseUp() va nous servir à vérifier si la position de la pièce "If" est correcte. Pour commencer, nous passons la variable booléenne "isValid" de toutes les pièces à false, puis nous allons vérifier leur placement.
    Pour se faire, nous allons appeler la fonction UpdatePuzzle() du script IfPuzzleScript.cs.
    Ensuite, si la pièce Start a bel et bien une pièce "If" juste en dessous d'elle, alors on passe la valeur isValid du "If" à true.
- **IfPuzzleScript.cs** : Ce script s'occupe de vérifier le placement de toutes les pièces actuellement sur l'éditeur. Si leur positionnement n'est pas correct, leur couleur sera grise. Sinon, une pièce Contrôle sera verte, une pièce Condition sera bleue, une pièce Action Non Terminale sera orange, une pièce Action sera Rouge, et une pièce Message sera Jaune.
  - **UpdateCondPuzzle()** : Ce script parcours toutes les pièces présentes sur l'éditeur. Si la pièce observée est une pièce de type Condition, et qu'elle est placée à droite d'une pièce "If", au niveau de sa première ligne, alors on l'attribue à une variable.

```
if (currentGridPos + new Vector2(1,0) == puzzleGridPos && ↩
    typePuzzle == PuzzleScript.Type.CONDITION)
```

Cela nous permettra, dans UpdatePuzzle, de savoir que nous avons une pièce Condition placée à notre droite, sur la bonne ligne. Le comportement des fonctions UpdateIfPuzzle() et UpdateActPuzzle

se comportent similairement. Le seul changement est la place dans l'éditeur. Une pièce "If" devra être située directement en dessous de notre "If" courant, et une pièce Action / Action non terminale / Message devra se trouver directement à droite de notre pièce "If", sur sa deuxième ligne.

— **UpdatePuzzle()** : Ce script récupère les pièces adjacentes à notre pièce actuelle, à l'aide des scripts UpdateIfPuzzle(), UpdateCondPuzzle, UpdateActPuzzle(). Il met ensuite à jour les valeurs des pièces adjacentes, à savoir leur variable isValid, ainsi que leur variable NextPuzzle, qui leur permet de savoir quelle pièce leur est adjacente.

Remerciements

Icônes

New file icon : Creative Stall
Load file icon : Ralf Schmitzer
Save icon : To Uyen
Undo : Michael Kussmaul
Redo : ImageCatalog
Sound Icon : Yaroslav Samoyl
Muted sound icon : Dev Patel

Polices

Enchanted Land
Dennis Ludlow