Editeur de niveaux
Prog & Play

Benjamin BONTEMPS

30 juin 2016

Résumé

Ce document a pour objectifs de récapituler l'essentiel du travail réalisé dans le cadre du projet de conception d'un éditeur de niveaux pour le jeu sérieux Prog&Play et de permettre sa maintenabilité en cas de nécessité. Il suppose acquis les concepts de base liés à Prog&Play, à Spring (moteur sur lequel est basé le jeu) et au développement en Lua.

Table des matières

1 Architecture globale

Cette section présente les choix effectués quant à l'architecture adoptée pour la réalisation de l'éditeur de niveaux.

Le core est composé d'un unique widget (launcher.lua) permettant de sélectionner le jeu de base sur lequel les niveaux vont être créés, ainsi que deux archives permettant de générer les éditeurs spécifiques ainsi que d'exporter le jeu final.

Le launcher est composé d'un widget central permettant d'afficher l'interface du launcher et de réagir en fonction des boutons sur lesquels l'utilisateur clique (editor_spring_direct_launch.lua). A noter que ce widget communique également avec les autres widgets et gadgets définis ci-dessous.

L'éditeur à proprement parler est composé de :

- Un widget central permettant d'afficher les éléments de l'interface utilisateur et de capter les interactions avec cette dernière ou avec le moteur pour effectuer des actions en conséquence. (editor_user_interface.lua)
- Un widget permettant de récupérer la liste des commandes que les différentes unités du jeu sélectionné dans le pré-launcher peuvent exécuter. (editor_commands_list.lua)
- Un widget permettant de récupérer la liste de tous les widgets présents. (editor_widget_list.lua)
- Un widget affichant un écran de chargement pour empêcher l'utilisateur d'interagir avec l'interface lorsqu'elle n'est pas complètement chargée (editor_loading_screen.lua)
- Un widget permettant de ne pas afficher les boutons permettant de donner des ordres aux unités. (hide_commands.lua)
- Un gadget permettant l'utilisation de code synchronisé (Par exemple, la création d'une unité sur le terrain). (editor_gadget.lua)
- Deux fichiers contenant toutes les chaînes de caractères à afficher. (EditorStrings.lua et LauncherStrings.lua)
- Un fichier définissant une classe "machine à états" et ses différentes instances. (StateMachine.lua)
- Un fichier contenant des fonctions utilitaires. (Misc.lua)
- Des fichiers de description de tables Lua. (Actions.lua, Conditions.lua, TextColors.lua)

Ces modules dépendent pour la plupart des bibliothèques suivantes :

- **Chili UI** - Framework d'interface graphique
- **dkjson.lua** - Permet de convertir une table lua en string au format json et inversement.
- **xml-serde.lua** - Permet de convertir une table lua possédant une syntaxe spécifique en string au format xml et inversement.
- **RestartScript.lua** - Permet de redémarrer le jeu avec certains paramètres supplémentaires.

2 Chili UI

Afin de simplifier le développement de l'interface graphique, j'ai choisi d'utiliser un framework d'interface graphique développé spécifiquement pour Spring dénommé Chili UI.

Ce framework a l'avantage de ne pas avoir à utiliser l'API OpenGL de Spring et les différents listeners pour afficher et permettre l'interaction avec des fenêtres, boutons etc. Tout y est déjà implémenté, ce qui a permis de réduire considérablement le temps requis pour obtenir les premiers résultats. Le problème majeur est que le framework est extrêmement mal documenté, et la meilleure façon de comprendre son fonctionnement pour s'en servir correctement voire même le modifier pour l'adapter à des besoins particuliers est de se référer au code source présent sur github [1].

Les modifications que j'y ai personnellement apporté concernent les boutons, qui possèdent un état supplémentaire *chosen* permettant d'avoir un feedback visuel lorsqu'on les sélectionne (les modifications de cet état sont gérées "à la main", mais le feedback visuel est géré automatiquement par le framework). Une autre modification a également été réalisée : il s'agit de l'ajout d'un membre "onReturn" de type *function* sur les editbox qui permet de donner un comportement à une pression sur la touche entrée lorsque l'utilisateur a le focus sur une editbox.

Le widget central de l'éditeur comporte de nombreuses fonctions reprenant les fonctions *New* du framework en n'utilisant qu'un certain nombre de paramètres, ce qui permet une définition plus compacte de chaque élément de l'interface. Le point négatif est qu'il faut soit connaître par cœur l'ordre dans lequel les paramètres doivent être renseignés dans le prototype de la fonction soit se référer en permanence à la définition des fonctions. Ces fonctions sont définies dans le widget central de l'éditeur dans la catégorie *Chili UI functions*.

Il faut faire extrêmement attention à une chose en particulier en ce qui concerne le framework : il ne faut surtout pas modifier les éléments d'interface qui ne sont pas affichés à l'écran (c'est-à-dire les éléments d'interface qui n'ont ni Screen0, ni un descendant de Screen0 en tant que parent). Ceci ne produit pas d'erreur lors de l'exécution, mais les éléments d'interface concernés ne s'actualiseront plus correctement, ce qui est très gênant d'un point de vue utilisateur (les opérations seront effectuées, mais l'utilisateur n'aura aucun feedback visuel).

3 Le core

Le core est la base de l'éditeur. Il permet de sélectionner un jeu de base sur lequel l'utilisateur va ensuite créer ses niveaux. Lorsque le jeu en question est sélectionné, le pré-launcher va s'occuper de créer une archive contenant les fichiers permettant le fonctionnement de l'éditeur tout en y modifiant le fichier ModInfo.lua contenant les informations de dépendance pour y ajouter le jeu sélectionné. Ensuite, il va redémarrer Spring avec l'archive nouvellement créée.

Si l'archive correspondant à l'éditeur pour un certain mod est déjà créée, il va simplement redémarrer Spring en utilisant cette archive. Ceci ne pose pas de problèmes d'un point de vue utilisateur, mais pendant une phase de développement, il ne faut pas oublier de supprimer l'archive créée à chaque modification.

Le core possède également une seconde archive qui contient toutes les données nécessaires pour faire fonctionner les missions créées avec l'éditeur. En effet, lorsque l'utilisateur exportera son scénario sous la forme d'un jeu, le launcher extraira du core l'archive *game* et il y ajoutera les missions, le scénario ainsi que les traces expertes sélectionnées.

1. https ://github.com/jk3064/chiliui/

4 Le launcher

La launcher se présente en 4 menus distincts : Création d'un nouveau niveau, Edition d'un niveau existant, Edition d'un scénario et Exportation d'un jeu. Pendant l'écran de chargement du launcher, le gadget de l'éditeur instancie chacune des unités du mod de base, récupère la liste de leurs commandes et la transfère au widget editor_commands_list.lua qui effectue un tri dessus. Les commandes seront ensuite passées à l'éditeur grâce au ModOptions.

4.1 Création d'un nouveau niveau

Rien de très compliqué ici : on s'occupe simplement de lister les cartes présentes dans le répertoire maps/ à la racine de Spring. Lorsque l'utilisateur clique sur le nom d'une carte, Spring redémarre sur la carte sélectionnée.

4.2 Edition d'un niveau existant

De même, on liste ici les niveaux correspondant au jeu sélectionné qui sont dans le répertoire pp_editor/missions/. Lorsque l'utilisateur clique sur le nom d'un niveau, on va chercher dans la description de ce niveau la carte sur laquelle le niveau a été créé, puis on redémarre Spring sur la bonne carte et on indique dans le ModOptions que l'éditeur va devoir charger ce niveau.

4.3 Editeur de scénario

Pour chaque niveau créé correspondant au jeu sélectionné, on créé une fenêtre contenant le nom du niveau, un bouton pour l'état d'entrée et n boutons pour les n états de sortie. En cliquant sur un bouton correspondant à un état de sortie, on stocke dans les variables *selectedOutputMission* et *selectedOutput* l'état de sortie sur lequel on vient de cliquer. Si ces variables ne sont pas nil et que l'on clique sur un état d'entrée, on le stocke dans *selectedInput*.

La fonction MakeLink() s'effectue à chaque frame et s'occupe de répertorier un lien dans la table Links si selectedOutputMission, selectedOutput et selectedInput ne sont pas nil. Un objet Chili personnalisé permet d'afficher ces liens lorsqu'ils existent.

En ce qui concerne la sauvegarde et le chargement des scénarios, ceci se fait en utilisant la bibliothèque xml-serde qui nous permet de transformer la table Links en fichier xml et vice-versa.

4.4 Exportation du jeu

Pour l'exportation de l'archive du jeu final, l'utilisateur va simplement choisir un des scénarios qu'il a préalablement créé via l'éditeur de scénario. Il peut également spécifier s'il ne veut exporter que les niveaux liés au scénario ou bien tous les niveaux du répertoire <Spring>/SPRED/missions. Une fois que le scénario en question a été sélectionné, on effectue la même opération que pour charger un scénario dans l'éditeur de scénario, puis on lit le contenu de la table Links pour définir les niveaux à exporter, et enfin on créé une archive contenant les fichiers nécessaires au fonctionnement du jeu ainsi que le scénario, les niveaux qui le composent (ou tous les niveaux) et le cas échéant les traces sélectionnées par l'utilisateur.

5 L'éditeur de niveaux

5.1 Editor User Interface

Ce widget est l'élément central de l'éditeur de niveaux. Le choix d'un widget en tant qu'élément central se justifie par le fait que la majorité des interactions se font de façon asynchrone. L'unicité du widget vient du fait qu'il n'est possible de n'avoir des variables communes à plusieurs widgets qu'en passant par la table WG, ce qui aurait été gênant étant donné le très grand nombre de variables devant être utilisées à de nombreux endroits distincts. Les sections suivantes expliquent rapidement le contenu de chacune des parties ; se référer au code et aux commentaires pour plus de précisions.

5.1.1 Initialisation

L'initialisation de la quasi-intégralité de l'interface se fait par les fonctions d'initialisation décrites dans la catégorie *Initialisation functions*. La grande majorité des fenêtres y est initialisée, en les affichant toutes d'un coup (pour les raisons évoquées à la section 2), puis elles sont ensuite masquées pour n'afficher que la fenêtre correspondant à l'état actuel de la machine à états globale (voir 5.3).

Sont présentes également deux fonctions s'occupant de l'initialisation de fonctions liées aux changements d'état lors de la sélection d'un type et d'une équipe d'une unité pour la placer ensuite sur le terrain.

Les fonctions des catégories *Top bar functions* et *Forces window buttons functions* s'occupent de changer l'état courant de la machine à états globale et de choisir quelles fenêtres doivent être affichées. Il s'agit pour la plupart de fonctions appelées lors de la pression sur des boutons ou sur des touches du clavier.

5.1.2 Unités et groupes d'unités

Les fonctions présentes dans la section *Unit/Selection state functions* gèrent à la fois le placement des unités (choix du type et de l'équipe), la sélection, le positionnement, l'orientation, les attributs et l'appartenance à un groupe d'une ou plusieurs unités.

Les états de la machine à états des unités permettent d'adapter le comportement des fonctions listeners d'évènements souris ou clavier. Ainsi, les fonctionnalités de sélection, rotation, déplacement... des unités se font dans les fonctions de callback des événements souris.

5.1.3 Zones

Les fonctions de cette section concernent la création, suppression, affichage, sélection et déplacement des zones logiques. Ces zones sont utilisées dans les déclencheurs.

De même que précédemment, les états de la machine à états régissent les interactions souris, qui sont donc gérées dans les listeners du widget.

5.1.4 Forces

Les fonctions de cette section s'occupent d'afficher les paramètres liés aux équipes et de gérer les alliances entre les différentes équipes actives.

5.1.5 Déclencheurs

Les fonctions de cette partie concernent la gestion du système de déclencheurs (création, édition, suppression des évènements, conditions et actions ainsi que la gestion des variables).

5.1.6 Paramètres de la carte

Les fonctions de cette partie sauvegardent les paramètres globaux du niveau, et transforme le texte du briefing en texte coloré et avec des retours à la ligne si les tags sont présents.

5.1.7 Sauvegarde / Chargement

Les fonctions de cette partie s'occupent de la gestion des fichiers de sauvegarde (.editor), de la transformation de l'état du niveau en une table lua, et des save states (ctrl+z).

5.1.8 Fonctions de dessin

Les fonctions de cette partie permettent de dessiner des ellipses sur le terrain, de gérer les différents curseurs et l'état de certains boutons.

5.1.9 Listeners

Les fonctions de cette partie correspondent aux callbacks des événements souris et clavier.

5.2 Gadget

Le gadget permet surtout d'effectuer toutes les opérations nécessitant d'être en code synchronisé. Il se décompose en 3 méthodes :

RecvLuaMsg Cette fonction permet de recevoir les messages envoyés depuis les widgets vers le gadget et d'effectuer des opérations en adéquation. Globalement, elle récupère un message de la forme "*operation++param1++param2++...*", split cette chaîne de caractères, effectue un switch sur "*operation*", effectue éventuellement des calculs avec les paramètres, stocke les paramètres transmis et calculés dans des variables, puis mets un booléen à *true*.

GameFrame Cette fonction est appelée à chaque frame. Elle s'occupe lorsque l'on est dans le launcher d'instancier toutes les unités du mod de base sélectionné et d'en récupérer leurs commandes. Lorsque l'on est dans l'éditeur, elle s'occupe de supprimer toutes les unités de base créées automatiquement à la première frame, puis à chaque frame suivante elle effectue un test sur les booléens mis à *true* ou non par la méthode précédente. Si un des booléens est à *true*, on effectue des opérations avec les variables stockées précédemment, et on remet le booléen à false.

RecvFromSynced Permet de récupérer les messages envoyés depuis la partie synchronisée du gadget vers la partie non synchronisée pour ensuite utiliser des méthodes des widgets.

5.3 Machines à états

Les machines à états servent à savoir les actions à effectuer lorsqu'une seule commande (clic souris par exemple) peut effectuer plusieurs actions distinctes. En réalité, il s'agit d'une metatable possédant plusieurs membres correspondant aux états ainsi que deux méthodes, un getter et un

setter sur les états. Au final, les machines à états n'ont pas été bien exploitées puisqu'il s'agit ici juste de savoir dans quel état on se trouve (et une simple variable changeant de valeur à chaque changement d'état aurait fait le même travail). On pourrait augmenter l'intérêt d'avoir des machines à états en définissant des fonctions de transition à chaque fois que l'on appelle le setter, mais il n'y a pas d'utilité immédiate.

5.4 Fonctions utilitaires

Les fonctions implémentées dans ce fichier sont soit des fonctions classiques qui ne sont pas présentes de base en lua (arrondir un nombre, etc.) soit des fonctions utilisées à plusieurs endroits dans le projet (récupération des informations des équipes par exemple).

5.5 Fichiers de description

Les fichiers de description des actions et des conditions contiennent toutes les informations nécessaires pour définir les actions et les conditions présentes dans l'éditeur. Pour rajouter une action ou une condition, il faut donc ajouter un élément dans la table correspondante possédant les éléments suivants :

- **type** : id de l'action ou de la condition à ajouter.
- **filter** : catégorie à laquelle l'action ou la condition appartient (s'ajoute automatiquement à la liste des filtres de l'éditeur).
- **typeText** : description présente dans la combobox lors du choix du type de l'action ou de la condition.
- **text** : texte à trou de l'action ou de la condition.
- **attributes** : "trous" du texte à trou.

Chaque élément de la table **attributes** est composé de :

- **text** : identifiant dans le texte à trous
- **type** : type de paramètre. Les types disponibles sont :
  - **unit** (une seule unité à pick dans la scène)
  - **unitset** (un ensemble d'unités = une seule unité, un groupe, une équipe, les unités validant une condition ou les unités créées par le dernier appel à une action)
  - **unitType** (un type d'unité)
  - **team** (une équipe)
  - **player** (une équipe contrôlée par un joueur
  - **position** (une position = un couple x,z de coordonnées)
  - **zone** (une zone logique)
  - **group** (un groupe d'unités)
  - **numberComparison** (un opérateur de comparaison (au moins, au plus, etc.) + un nombre)
  - **numberVariable** (une variable numérique)
  - **booleanVariable** (une variable booléenne)
  - **comparison** (un opérateur de comparaison)
  - **condition** (une condition parmi celles définies dans l'événement actuellement modifié)
  - **toggle** (activé/désactivé)
  - **command** (un ordre)
  - **text** (une chaîne de caractères)
  - **textSplit** (texte pouvant être transformé en table grâce au séparateur "||")
  - **boolean** (true/false)

    — **widget** (un widget parmi les widgets visibles)
— **id** : case dans laquelle la valeur sélectionnée va être stockée.
— **hint** (optionnel) : message d'aide pour l'utilisateur.

Une fois qu'une nouvelle action ou condition a été définie dans l'éditeur, il ne faut pas oublier d'implémenter son comportement dans les fichiers s'occupant de l'exécution du jeu.