LE TOOL PROGRAMMING APPLIQUE AU LEVEL EDITOR

Comment la programmation outil peut-elle faciliter la création de niveaux dans le jeu vidéo ?

Emeline BERENGUIER
Master jeux et médias interactifs numériques
CNAM-ENJMIN
Angoulême – France
emeline.berenguier@gmail.com

INTRODUCTION

Le développement d'un jeu fait appel à beaucoup de compétences très différentes les unes des autres, convergeant toutes vers un même objectif : celui de produire un jeu à la hauteur des exigences de chacun. Durant ce processus, je me suis rendue compte de plusieurs aspects du développement peu connus et pourtant pas moindres. D'un point de vue programmation, le développement de features n'est pas la seule tâche à effectuer. Il est aussi demandé de prévoir en amont les méthodes d'intégration et de production d'un point de vue technique.

Durant mes projets en équipe, j'ai pu remarquer la nécessité de développer des outils pour faciliter la production. Que ce soit un outil pour automatiser l'intégration sonore ou graphique, des outils pour gérer le logiciel de versionning ou des outils pour optimiser la production en améliorant des logiciels de production.

Cependant, le développement d'un outil est un investissement long terme, pas toujours pertinent selon l'ambition du projet, ainsi je n'ai pas eu l'occasion de développer des outils avancés dans le cadre d'un projet de jeu.

Je profite de ce sujet d'expérimentation pour traiter ce sujet plus en détail, essayer de comprendre les enjeux liés au développement outil, sa pertinence, que ce soit d'un point de vue technique ou d'un point de vue impact sur la production.

Cependant, le développement outil est très vaste : cela peut aller d'outils d'animation (Inverse Kinematic, Dynamical Bones…) à des outils de traduction en passant à des outils de génération de meshes. Le monde de l'outil est aussi grand qu'il y a de besoins dans le jeu vidéo.

J'ai donc décidé de me limiter à un domaine très précis et de l'explorer plutôt qu'essayer divers sujets sans leur approfondir. Pour ce sujet, j'ai choisi de traiter l'outil appliqué au Level Design, étudier les outils correspondants, les limites, les possibilités, et les attentes.

Durant ce projet, je vais étudier le développement d'un outil appliqué au Level Design, en partant de zéro. J'ai pour volonté de creuser le sujet de manière pertinente, en gardant un certain recul et une hauteur de vue pour ne pas tomber dans le dogme de la technique pure, mais de combiner besoins actuels et réponses techniques. Ce travail a pour but de définir les attentes dans une production classique et à approfondir les problématiques techniques liées.

ETAT DE L'ART

Le Tool Programming est une spécialité dans le secteur de la programmation jeux vidéo de plus en plus recherchée. Il n'est pas rare de voir des annonces dans dans les offres d'emploi.

Cependant, lorsqu'on regarde les annonces de Tools Programmer, les demandes sont assez vagues et différentes selon les studios, que ce soit au niveau des langages ou des compétences sociales. Certaines offres demandent même des notions d'ergonomie.

Cette différence peut être expliquée par les besoins du studio : le domaine de l'outil est vague, un Tool Programmer peut être sollicité pour corriger un outil déjà existant ou en développer un autre, il n'y a pas de profil type.

J'ai contacté une Tool Programmer à Ubisoft pour mieux comprendre les attentes. A son travail, les outils les plus utilisés sont Qt, Docker et le langage SQL.

Bien sûr, cela n'est qu'un exemple parmi d'autres. Certaines entreprises sont spécialisées dans la création d'outils pour moteur de jeu, comme l'Asset Store d'Unity ou le Market Place d'Unreal.

CONTRIBUTION

Durant mon travail de recherche, j'ai été amenée à traiter différents sujets, tous très intéressants. Cependant, j'ai cherché à les développer sans m'éloigner du sujet, ils seront donc abordés dans la mesure où cela permet d'approfondir l'outil appliqué au Level Editor.

0. Préface : Qu'est-ce qu'un bon outil ?

Avant toute chose, il est essentiel de définir la direction dans laquelle je souhaite travailler. Le but de ce travail est d'étudier l'outil appliqué au Level Editor. Cependant, un outil réussi est avant tout un outil utile. Et les usagers de cet outil ne sont pas les développeurs, mais les Level Designers.
Avant d'entrer dans la partie technique du sujet, je vais d'abord aborder la question d'un point de vue ergonomique.

Après entretien avec une Level Designer de ma promotion sur la question « Qu'attends-tu lorsque tu utilises un outil de Level Editor ? », on peut retenir plusieurs points principaux de ses réponses :

- Prise en main facile et intuitive : l'outil doit être fluide à utiliser. (Problématique ergonomique)
- Les niveaux doivent être sauvegardables et jouables (Problématique technique)
- Grand choix de features (Problématique technique)
- Possibilité d'avoir une vision d'ensemble : le Level Designer fonctionne par « pièce de puzzle » et « thème » (Problématique ergonomique)

Cette définition n'est surement pas complète et pas satisfaisante d'un point de vue ergonomique, cependant elle sera acceptée durant ce travail de recherche, et servira d'objectif à atteindre lors des projets expérimentaux.

1. Prise en main de Qt

Qt est une des références en terme de développement outils. Qt est un framework en C++ qui permet de créer des GUI (Graphical User Interface) et des logiciels.

Avec le temps, différentes méthodes se sont développées pour utiliser Qt : Qt Creator et QML. Pour cette partie, nous utiliserons Qt Creator.

1.1 Introduction à Qt

Qt se prend assez facilement en main, malgré quelques particularités propres à lui (exemple : exécuter `qmake` avant de faire sa première compilation). Je me suis familiarisée avec le vocabulaire de Qt en faisant quelques petites fenêtres basiques. Ma première fenêtre consiste à entrer un texte et afficher une fenêtre avec le texte entré selon le bouton cliqué précédemment (fig. 1 et fig. 2). Cela m'a permis d'apprendre à créer des boutons et à les connecter à des fonctions avec le système d'events interne à Qt : `SIGNAL / SLOT`, sender/receiver et `QObject::connect()`. Les fenêtres sont générées avec les `QMessageBox` et la saisie de données par le `QInputDialog`.

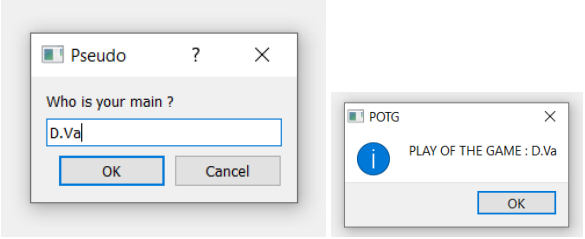


Figure 1 et 2 : Exemples de fenêtres créées avec Qt

La seconde fenêtre permet d'ouvrir l'explorateur, de sélectionner un fichier selon un type choisi et d'en afficher son nom dans une nouvelle fenêtre. La seule nouveauté est l'utilisation du `QFileDialog` qui permet d'ouvrir l'explorateur de fichiers de son ordinateur.

Une fois la logique prise en main, je peux commencer à tester Qt pour mon projet suivant : faire un Level Editor sous Qt.

1.2 Projet expérimental : Tile Editor sur Qt.

Le but de ce projet est très simple : faire un Tile Editor dans lequel le Level Designer peut changer le type des cases. De manière détaillée, le Level Designer sélectionne le type de terrain qu'il souhaite, et peut changer ensuite le type de la case du niveau. Les niveaux sont sauvegardables et chargeables dans l'éditeur.

Dans un premier temps, j'ai mis en forme l'application dans Qt Designer qui facilite le développement graphique de l'application (fig. 3). J'ai généré le niveau dans une grille entièrement constituée de boutons, qui changent visuellement une fois cliqués en fonction du type de terrain sélectionné par l'utilisateur.

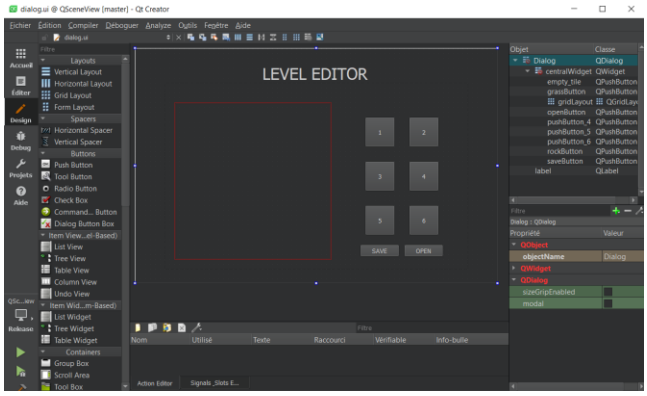


Figure 3 : Création de l'interface avec Qt Designer

Par ailleurs, j'ai remarqué que Qt Designer permettait d'automatiser certaines manipulations comme les connexions boutons – fonctions lors d'une interaction basique avec le bouton (click, relâché, survolé…).

Une fois l'éditeur fini, je suis passée à la partie sauvegarde du niveau. Cette partie demandait de nouvelles fonctionnalités auxquelles je n'avais pas encore touchées telles que `QFile` et `QTextStream` : la première étant le type des fichiers trouvables sur un ordinateur, la deuxième permettant de lire et modifier ses fichiers.

La sauvegarde stocke dans l'ordre les types des cases. La lecture lit le fichier de sauvegarde pour réaffecter le bon type à toutes les cases du niveau.

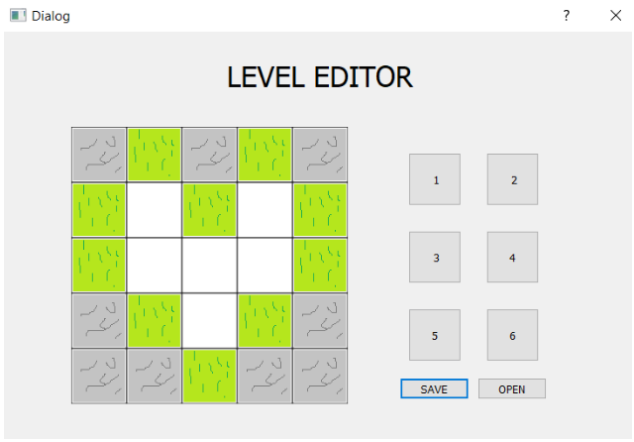


Figure 4 : Résultat final du Level Editor

1.3 Bilan de Qt.

Durant ce projet j'ai pu me rendre compte de plusieurs points importants à soulever :

- Qt est vieillissant, comme sa communauté : une partie de sa documentation (en particulier des exemples interactifs) n'est plus hébergée et les topics datent des souvent de trois ans ou plus. De plus, certaines fonctionnalités sont buguées sans possibilité de résolution

- QT QML est plus apprécié car plus facile à prendre en main et plus visuel : il s'appuie sur le langage JavaScript et fonctionne plus comme un langage de mise en page qu'un langage de script

- Qt reste un framework beaucoup plus tourné vers le développement de logiciels / applications que le jeu vidéo, très peu de features sont pensées pour en faire un jeu

- Qt est très difficile à coupler avec un jeu vidéo : aucune intégration possible avec les moteurs de jeu

2. Transition de Qt vers un moteur de jeu

Malgré qu'on ne puisse pas l'intégrer dans un moteur de jeu, il est quand même possible de faire des liaisons indirectes entre Qt et un moteur de jeu, c'est ce que nous étudierons dans cette deuxième partie.

2.1 Etude de cas

Plusieurs fonctionnalités, autre que le développement de fenêtres, sont incluses dans Qt. Il est possible de développer des outils pour accéder à un réseau (chat, navigateur…), ou à une base de données SQL tout en passant par la création de fenêtre contenant de la 3D avec OpenGL ou encore des modules de dessin 2D à la manière d'un Paint.

Il est donc en fait possible de se servir de Qt comme outil externe à la production. Par exemple, une idée d'outil serait d'accéder à des données en requête SQL comme les données d'un profil de joueur, les exporter sous forme d'un fichier et les lire dans un moteur de jeu pour faire des tests ou avoir une mise en situation. De manière plus générale, on peut séparer le travail en deux parties distinctes : d'un côté tout ce qui concerne le jeu d'un point de vue gameplay, et de l'autre les données à l'état pur, qu'on peut modifier.

C'est ainsi que je vais procéder, en essayant d'étudier une liaison possible entre le Tile Editor réalisé sur Qt et un moteur de jeu, ici Unity.

2.2 Projet expérimental : Charger un niveau de Qt dans Unity

Ce projet a pour but de tester la liaison entre Qt et Unity. Pour cela, je vais me baser sur le fichier de sauvegarde du niveau du Tile Editor pour le lire sur Unity.

Le fichier de sauvegarde est dans un format .txt, j'évoquerai ce sujet plus tard, mais dans ce cas de figure, je resterai sur cette version car mon but est de traiter la transition entre Qt et Unity, d'un point de vue technique comme ergonomique.

Le fichier de sauvegarde est donc un fichier .txt dans la forme la plus basique qu'il soit, facilitant sa lecture et son écriture dans les deux logiciels puisqu'ils ont tout deux des fonctions internes pour lire et éditer des fichiers textuels.

Dans un premier temps, il faut d'abord accéder au fichier. Comme Qt et son QFileDialog, Unity a aussi une fonction toute faite pour ouvrir des fichiers : EditorUtility.OpenFilePanel(). Les deux fonctions œuvrent pareil, on peut choisir le nom de la fenêtre et le type de fichier recherché.

Tout comme Qt, cette fonction retourne un fichier qui faut lire pour en extraire les informations. Comme j'utilise un fichier .txt, la lecture est simplifiée, Unity fournit la classe StreamReader, qui, avec la fonction StreamReader.ReadToEnd() permet de retourner le contenu du fichier de sauvegarde sous la forme d'un string.

Une fois la lecture finie, le programme remplit une grille de la taille du niveau de Qt et le niveau s'affiche correctement sur Unity (fig. 5).

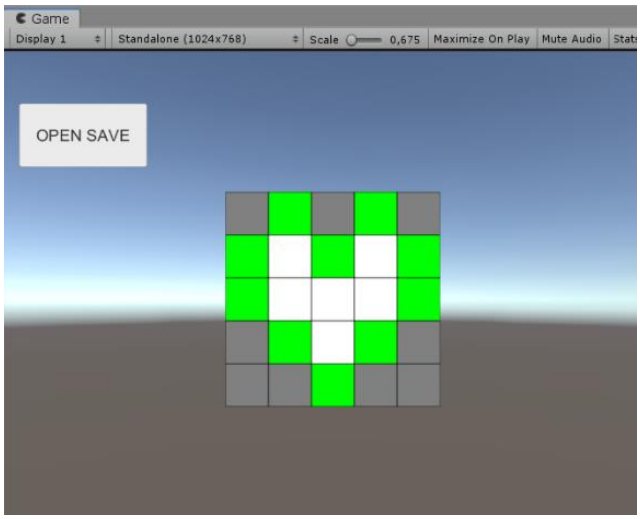


Figure 5 : Niveau sous Unity après ouverture du niveau généré par le Level Editor de Qt

J'ai donc maintenant la configuration suivante : une application, réalisée sous Qt, qui permet de créer un niveau, de le sauvegarder et de le modifier, et un programme dans l'éditeur d'Unity qui permet de l'ouvrir et de le tester.

2.3 Bilan

Ce projet de liaison n'était qu'un exemple parmi d'autres possibles avec Qt, cependant il permet de soulever des points indépendants de la fonctionnalité du logiciel en elle-même :

- La modification du niveau se fait entre deux logiciels : l'application de Tile Editor et l'éditeur d'Unity. Même si les temps de chargement, sauvegarde et lecture sont très rapides, pour ne pas dire instantanés, on perd de vue un des critères d'un bon outil : celui de la fluidité. Dans un workflow de production, le Level Designer aura à ouvrir deux applications en même temps, et à switcher de l'une à l'autre. L'expérience utilisateur n'est donc pas optimale.

- Pour tout ce qui est graphique, Unity est bien plus développé que Qt qui ne contient que peu de fonctionnalités citées plus haut dédiées au visuel. Le Tile Editor de Qt est extrêmement facile et rapide à refaire sur Unity, avec plus de fonctionnalités.

- Il est peu efficace de chercher à développer des outils de gameplay (ici du Level editor) sur des logiciels tiers au moteur sur lequel est testé le jeu. Une transition Qt-Unity serait plus pertinente pour des fonctionnalités auxquelles Qt serait plus adapté par rapport à Unity. On peut penser à une communication avec une base de données en SQL, car passer par Unity demanderait un temps d'attente inutile de compilation.

- Contrairement à Qt, on ne peut pas exporter d'outils propre à Unity sous forme de build. Le sens doit donc toujours être outil de production pour un projet fait sur Unity et non l'inverse.

3. L'outil dans un moteur de jeu

J'ai étudié la programmation outil avec un logiciel de création d'outils, je vais maintenant explorer l'outil dans un moteur de jeu. Pour continuer sur la lignée de mon travail, j'étudierai ici l'outil avec le moteur de jeu Unity en version 2019.1.

3.1 Etude de cas

Unity est un des moteurs de jeu public les plus puissants du marché. Un de ses grands points forts est le nombre de fonctionnalités mises en place pour réduire le temps de développement d'un jeu. Parmi ses services proposés, on peut penser à l'Unity Asset Store, une plateforme sur laquelle des

développeurs peuvent partager leurs assets ou outils de développement. Lorsqu'un outil devient très complet et très apprécié de la communauté, il arrive qu'Unity le rachète pour l'intégrer directement dans le moteur, tel est le cas par exemple de Pro Builder, un outil de Level Design permettant de créer des niveaux en Grey block.

Mais Unity n'a pas que son Asset Store à proposer, il développe aussi des outils internes au moteur pour des utilisations parfois assez poussés pour être considéré comme un moteur dans le moteur.

L'exemple en tête est ici le TileMap développé par Unity en 2017. Cet outil permet de créer des niveaux en tiles maps 2D de manière très rapide et complète (fig. 6).

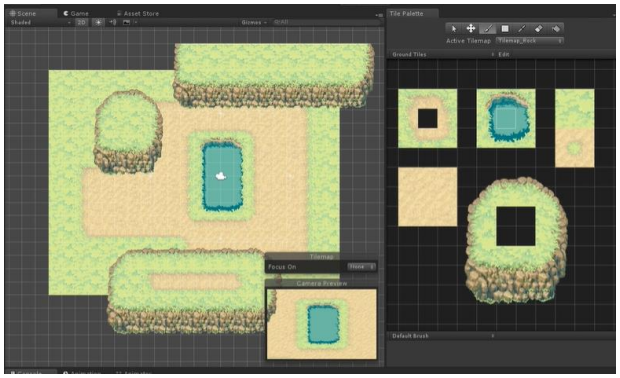


Figure 6 : Niveau sous Unity avec le TileMap

Unity propose donc une version très aboutie de mon Tile Editor fait avec Qt. J'ai fait des recherches pour analyser la fiabilité de cet outil, et voici les résultats obtenus : l'outil est facile à prendre en main, propose un nombre de fonctionnalités assez grands (gestion des collisions, pinceaux, animations, z-forward…), avec des automatismes implantés (en fonction de la position d'un bloc dans un niveau, il n'aura pas le même visuel (fig. 7)), une interface efficace et claire. En résumé, il est difficile, et inutile de faire mieux. Continuer à développer mon Tile Editor de Qt n'aurait pas de réel intérêt en termes de workflow.

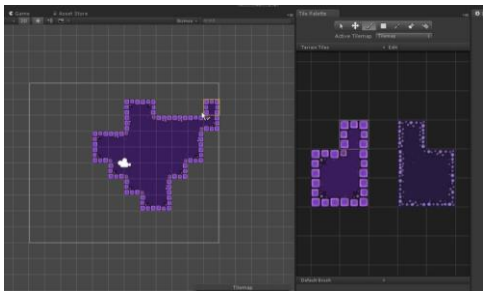


Figure 7 : Gestion des murs dans le TileMap

J'ai choisi de parler du TileMap d'Unity car il était dans la continuité de ce dont j'avais traité précédemment. Mais cet exemple peut s'appliquer à tout : les outils de Level Editor, que ce soit dans Unity ou sur l'Asset Store, ne manquent pas. Ils sont très complets et à des tarifs abordables voire gratuits. Quelle est donc la place laissée pour le développement d'outil interne ? Est-il pertinent ?

J'ai décidé pour cela de développer un outil plus spécifique sur Unity : un Planet Tile Editor.

3.2 Projet expérimental : Planet Editor sur Unity

Pour rester dans la lignée des Tile Editor, et proposer une recherche pertinente par rapport aux fonctionnalités déjà proposées, j'ai décidé de partir dans le développement d'un Planet Editor. Le fonctionnement est le suivant : sur une planète low-poly définie, le Level Designer peut modifier le type des facettes et y poser différents props en fonction du type de la cellule. Le niveau de la planète est sauvegardable, le Level Designer peut sauvegarder plusieurs sauvegardes de différents niveaux et les ouvrir quand il le souhaite.

J'ai d'abord commencé par créer la planète sur Blender (fig.8). La planète est une icosphère, pour donner un effet low-poly et faciliter la sélection des faces.

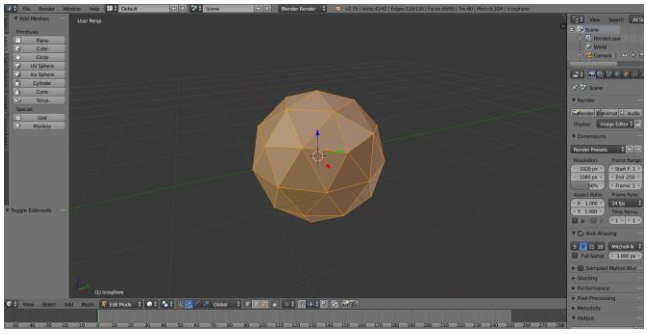


Figure 8 : Création de la planète sur Blender

Une fois importée dans Unity, il est alors possible de cliquer sur une des faces. Une des premières fonctionnalités que j'ai voulu implémenter était un inspecteur dynamique en fonction de si la cellule était cliquée ou non.

J'ai dû pour cela faire attention à plusieurs détails :

- Le premier a été de faire le choix entre faire un outil qui fonctionne dans l'Unity Editor, ce qui est plus pratique pour le Level Designer car inutile de lancer une compilation du projet, et un outil qui fonctionne en Runtime. Mon choix s'est porté sur le Runtime car il était impossible de développer uniquement pour l'Editor, cela me privait de fonctions basiques comme

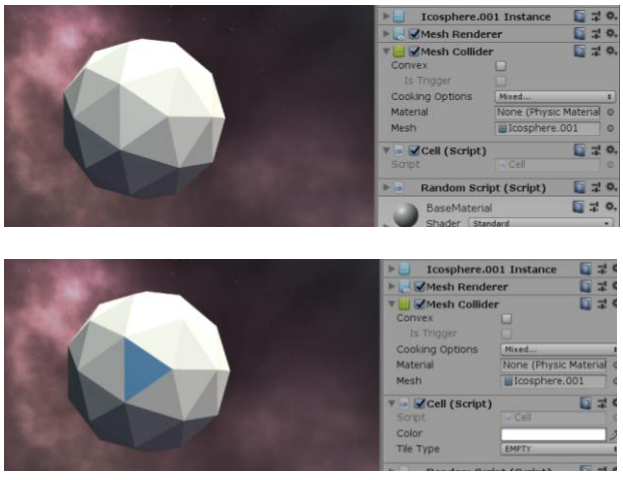
les Update(), qui m'étaient essentielles pour détecter les collisions.

- Unity autorise les développeurs à modifier l'inspecteur des GameObjects, c'est ce qu'on appelle les Custom Editor. Pour lier un Custom Editor à un script, il faut créer une nouvelle classe qui hérite d'Editor, et lui faire cibler le script avec l'instruction [CustomEditor(typeof(Script))]

- La communication entre la cellule et son inspecteur est particulière. L'inspecteur se modifie selon le fait que la cellule soit cliquée ou non (fig.9). Il est possible d'accéder aux variables de la cellule avec le mot-clé target, hérité d'Editor, qui permet d'accéder au script.

Figure 9 : Schéma explicatif d'un click sur une cellule

OnInspectorGUI() est une fonction héritée d'Editor qui est appelée à chaque fois que l'inspecteur est affiché. Une fois affiché, il va vérifier si la cellule est dans l'état cliqué, et si tel est le cas, il affichera certains éléments à l'aide de la classe EditorGUILayout. Cette classe propose différentes fonctions pour afficher des champs de variables dans l'inspecteur selon le type de la variable à modifier. Par exemple, dans les fig. 10 et 11, je veux modifier le type de sol de la cellule dans l'inspecteur. Le type de la cellule est de type enum, j'utilise donc EditorGUILayout.EnumPopup().



Figures 10 et 11 : Cellule cliquée et modification de son inspecteur

Lorsqu'on change le type dans l'inspector, le script détecte la modification et peut modifier la cellule. Ici, le fait de modifier le type change la couleur de la cellule.

Passons maintenant au positionnement des props.

Pour cette fonctionnalité, j'ai choisi d'explorer deux des caractéristiques d'un bon outil : l'organisation des features par « thème » et une prise en main « intuitive ». J'ai poussé l'affichage dynamique de l'inspecteur en proposant maintenant de sélectionner des props sous forme d'icône et de les poser sur la face sélectionnée.

Les props sont des prefabs déjà pré-créés, j'aurais simplement pu développer des boutons textuels que le Level Designer sélectionne pour les poser. Cela manquait de clarté, j'ai préféré prendre une approche plus visuelle : une palette de props.

La palette est un tableau de GUIContent, qu'on remplit avec l'icône des prefabs des props. L'icône est la même que celle qu'on voit dans les dossiers du projet Unity, elle est récupérée avec la méthode AssetPreview.GetAssetPreview(prefab), qui retourne une Texture2D, qu'on peut passer au constructeur du GUIContent. Les props doivent être cependant placés dans un dossier spécifique à Unity, le « Editor Default Resources », pour qu'un script héritant d'Editor puisse accéder aux Assets du projet. Les prefabs sont récupérés avec la méthode AssetDatabase.LoadAssetAtPath(String, Type) qui attend le chemin d'accès du dossier de prefabs et le type recherché.

Pour donner un aspect plus maitrisé à la palette, j'utilise GUILayout.SelectionGrid() qui me permet de savoir quelle est la case sélectionnée. On peut ensuite passer le prefab sélectionné au script de la cellule pour qu'il se gère de l'instancier et de le positionner.

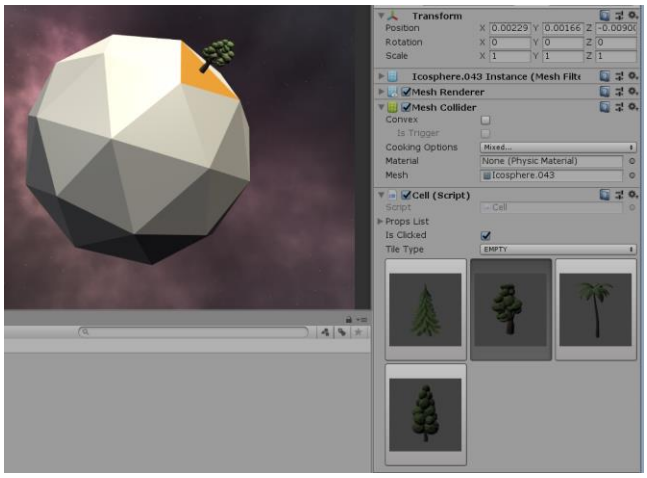


Figure 12: Affichage des props à positionner sur la planète

Cependant, j'ai voulu pousser l'affichage plus loin en affichant certains props en fonction du type de la cellule, par thématique (les palmiers avec l'eau, les sapins avec la roche…).

Comme dit plus haut, les prefabs sont récupérés avec la méthode AssetDatabase.LoadAssetAtPath(String, Type). Cette méthode attend un chemin sous forme de String. J'ai créé différents dossiers contenant différents props par thème, et à chaque fois que la cellule est cliquée, je vérifie le type. En fonction de son type, je modifie le chemin d'accès au dossier pour qu'il ne cible que la bonne catégorie (fig. 13). Si la cellule est vide, tous les props sont alors affichés.

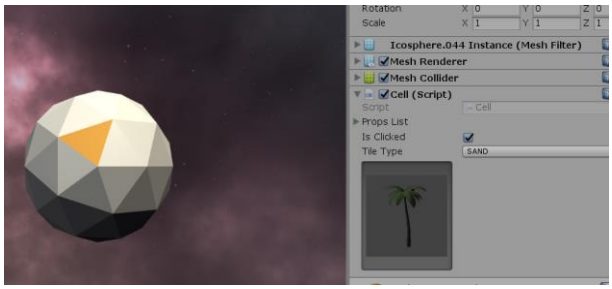


Figure 13: Palette dynamique en fonction du type de la cellule sélectionnée

Par ailleurs, pour faciliter l'édition, j'ai codé du multi-editing de cellules (fig. 14). Cependant, le multi-editing de script n'étant pas supporté par Unity, j'ai dû contourner en utilisant un autre moyen. Les cellules sélectionnées avec SHIFT + click sont entrées dans un tableau, et n'est affiché l'inspecteur que de la première cellule sélectionnée, à l'aide du Selection.objects. Comme les cellules sélectionnées sont vouées à avoir les mêmes modifications, les modifications faites dans l'inspecteur de la première cellule sont appliquées aux autres.

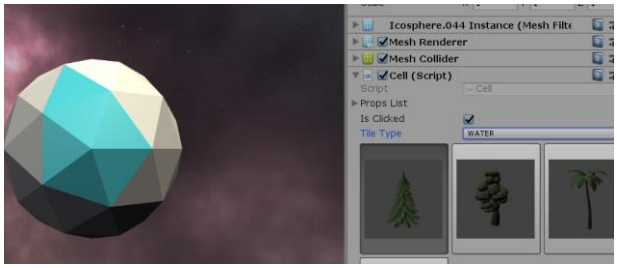


Figure 14: Multi-editing des cellules

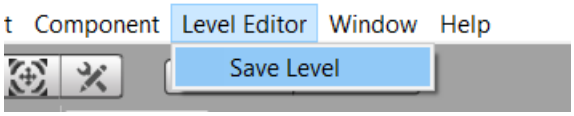
Après avoir creusé les possibilités offertes avec le Custom Editor, j'ai voulu étudier les Editor Window d'Unity. Comme les Custom Editor, Unity autorise les développeurs à créer leur propre fenêtre dans le moteur. J'ai décidé d'en créer une pour tout ce qui est sauvegarde du niveau.

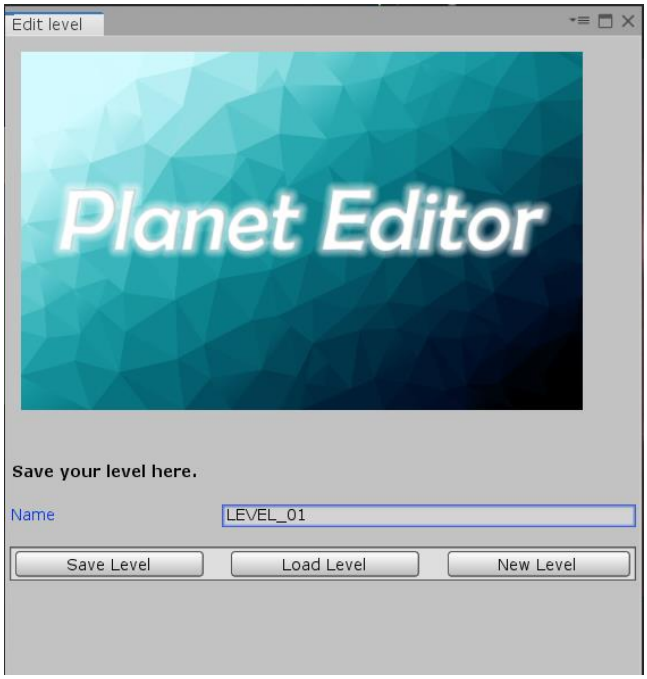
La fenêtre est très simple et contient 3 boutons : Charger un niveau, le sauvegarder ou en créer un nouveau. Elle propose aussi de changer le nom qu'on souhaite donner au niveau. Elle peut s'ouvrir Ingame en cliquant sur un bouton, ou en passant par la barre de menu d'Unity.

Une fenêtre est un script héritant d'EditorWindow. La gestion de son affichage, contrairement à Qt, se gère en script, et non pas un onglet propre à l'affichage UI, dans la fonction OnGUI(). L'affichage est plutôt laborieux car il faut indiquer toutes les intentions graphiques, rendant vite le tout illisible. Les propriétés graphiques se gèrent avec les classes GUILayout et GUI. Par exemple, pour rajouter du texte dans ma fenêtre, je passe par GUI.Label(), et pour séparer du texte GUILayout.Space().

Rajouter un bouton au menu se fait très simplement, il faut rajouter l'entête [MenuItem("Level Editor/Save Level")] au-dessus de la fonction chargée d'ouvrir la fenêtre.

On obtient le résultat souvent (fig. 15 et 16).





Figures 15 et 16: Fenêtre de sauvegarde et son ouverture

La gestion de la sauvegarde se déroule de la manière suivante : on ouvre l'explorateur de fichiers lorsqu'on veut charger un niveau, qui laisse le choix du niveau à récupérer. Pour

sauvegarder, le Level Designer n'a qu'à cliquer sur Save Level avec le nom de son choix (fig. 17).

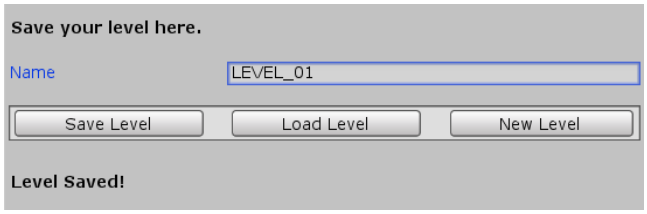


Figure 17: Phrase de feedback

Une phrase de feedback est là pour confirmer la démarche ou signaler un problème.

3.3 Bilan

Le développement de cet outil m'a permis de prendre conscience de plusieurs faits par rapport au développement outil sur Unity :

- L'outil développé est propre à Unity et est impossible à exporter. Contrairement à Qt, où j'ai pu faire une application qui peut être envoyée et utilisée par n'importe qui, sans posséder Qt, la quasi-totalité des fonctionnalités du Planet Editor ne peut fonctionner que dans le logiciel même.

- La plupart des fonctionnalités de l'outil font appel à des fonctions internes d'Unity, qui sont très spécifiques au moteur, propres à sa logique et par conséquent non retrouvables dans les autres moteurs et logiciels.

- Un outil de Level Editor utile dans Unity est un outil spécifique aux besoins du jeu. C'est-à-dire qu'on ne crée pas d'outil pour produire le jeu mais on crée un outil pour accélérer le développement de certaines features trop spécifiques aux outils d'Unity (par exemple, on ne peut pas poser des props automatiquement dans la bonne inclinaison sur Unity)

- Comme un outil utile sur Unity est un outil très spécifique, il est souvent propre au jeu et difficile à réutiliser dans un autre jeu, même s'il est développé sous Unity.

- Unity possède deux modes d'utilisation : le mode Editor et le mode Runtime. Il faut s'en souvenir pour développer un outil : il se peut que le Level Designer n'ait pas besoin de lancer le jeu pour avoir à la modifier, mais techniquement parlant, des fonctions peuvent être utilisées qu'en Runtime (fonctions héritant du MonoBehaviour tel qu'Update()). Il y a un équilibre à trouver entre la pertinence de coder un outil pour l'Editor et la difficulté de le développer en se passant de fonctions basiques.

- La gestion du double support Inspecteur – Game crée quelques problèmes, tel que la question du focus de la souris. Par exemple, lorsqu'on clique sur un bouton de l'inspecteur, il faut cliquer 2 fois pour que le jeu considère un click. Cela a dû demander une révision de la gestion d'inputs.

En conclusion, Unity laisse une grande liberté aux développeurs souhaitant développer leurs outils. Il faut cependant ne pas négliger la veille et le benchmarking avant de faire un outil dans Unity. De manière générale, dans un moteur de jeu, on développe plus des outils pour combler des lacunes du moteur que réellement un outil pour aider le Level Designer.

4. La sauvegarde

J'ai évoqué plus tôt les fichiers de sauvegarde des niveaux. Il était inutile de creuser le sujet dans le cadre de la recherche car c'est un sujet assez vaste qui demanderait plus de pages pour être traité convenablement. Cependant, je vais résumer ici mes recherches pour les sauvegardes de mes niveaux.

Je rappelle que la sauvegarde constitue une des clés de réussite du fonctionnement d'un outil. Un outil doit pouvoir sauvegarder et restituer rapidement un niveau pour ne pas faire perdre du temps au Level Designer. C'est un aspect technique à ne pas négliger, le plus utile et le plus risqué. C'est pourquoi je voulais traiter le sujet avec mes projets de recherche.

4.1 La sauvegarde du Tile Editor

La sauvegarde du Tile Editor fonctionne comme suivant : on écrit sous forme de caractère les données du jeu dans un fichier .txt (fig. 18). Le programme va chercher le fichier .txt de sauvegarde à un chemin fourni et écrire la sauvegarde avec le QTextStream.

La lecture se passe toujours par le QTextStream. On parcourt le fichier .txt avec la fonction QString.readLine() pour retourner le contenu sous forme de string.

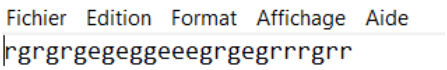


Figure 18: Contenu de la sauvegarde du .txt

Cette méthode a plusieurs avantages :

- Facilité de manipulation. Beaucoup de logiciels proposent des fonctions internes pour lire et écrire des fichiers .txt. Ces fonctions sont fiables et n'ont pas besoin d'une surcouche de code pour être utilisables

- Taille légère. Le fichier .txt est léger, très souvent inférieur à 1 Mo.

- Facilité de lecture : on peut facilement ouvrir le fichier de sauvegarde et en lire le contenu avant de vérifier / débuguer la sauvegarde.

Mais aussi de très gros inconvénients :

- La maniabilité des données : Pour accéder aux données, je dois toutes les lire et les stocker à la suite. Cela peut être acceptable pour un projet expérimental mais devient très vite difficile à gérer sur un plus gros projet. Une alternative serait de passer par le JSON ou XML qui permettent de mieux hiérarchiser les données. Une fois parsé, les formats en format JSON ou XML proposent un accès aux données plus efficaces qu'un fichier .txt.

- La sécurité. Comme citée plus haut, le fichier .txt est facile à ouvrir en passant par un logiciel tiers et à modifier. Cela signifie qu'un joueur peut ouvrir sa sauvegarde et se rajouter des bonus comme de l'argent ou de meilleures statistiques, ce qui est une faille de sécurité. Dans notre cas, ce ne sont pas des joueurs mais des Level Designers, ils n'ont donc pas volonté de modifier le niveau en passant par la sauvegarde. Il est préférable de conserver une sécurité supplémentaire pour éviter une fausse manipulation de leur part.

Cette méthode a été utilisée pour gagner du temps lors de la réalisation des projets expérimentaux, mais n'est pas satisfaisante. La faille de sécurité est trop importante pour la laisser tel quel. Je suis donc partie sur une autre méthode avec le Planet Editor.

4.2 La sauvegarde du Planet Editor

La première différence majeure entre la sauvegarde du Tile Editor et du Planet Editor est le format du fichier, dans le Planet Editor j'utilise un fichier binaire. En effet le binaire a de nombreux avantages :

- L'écriture binaire est la méthode non-compressée de données le plus compacte. Pour un même nombre de données, un fichier uniquement en binaire est plus léger que le fichier .txt

- Le fichier est illisible et très compliqué à ouvrir. On peut cependant l'ouvrir mais le contenu est tellement abstrait qu'il est impossible de le modifier en parfaite connaissance de cause.

Passons maintenant au fonctionnement de la sauvegarde. Tout d'abord, il faut savoir ce qu'on souhaite sauvegarder ou non. Par exemple, je n'ai pas besoin de sauvegarder l'orientation de la planète si le Level Designer l'avait fait tourner. Mais j'ai cependant besoin de connaître la position des props sur la planète.

Voici une liste de ce qui est écrit dans le fichier binaire lors d'une sauvegarde, dans l'ordre de sauvegarde :

- Le nombre de cellules de la planète (int)
- Le type de la cellule (enum)
- Le nombre de props de la cellule (int)
- Les props (GameObject)

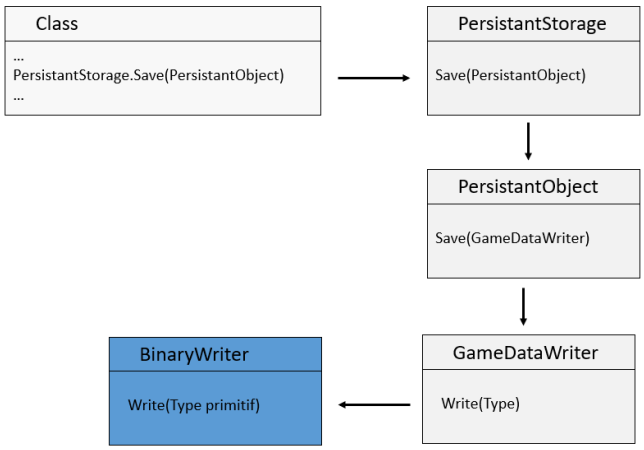


Figure 19: Schéma d'une sauvegarde générale dans le Planet Editor

Les principaux éléments de la sauvegarde sont le PersistantStorage, le GameDataWriter et le PersistantObject.

Le PersistantStorage est une classe statique. On l'appelle pour lancer la sauvegarde globale. Il se charge de créer une instance de GameDataWriter qui fait office de sauvegarde. Une fois l'instance créée, elle est passée en paramètre à l'objet auquel on souhaite sauvegarder ses données. Cette instance ne doit surtout pas être recréée et est unique durant le temps de la sauvegarde.

L'objet qu'on souhaite sauvegarder doit hériter de PersistantObject pour accéder aux fonctions de Save() et Load(). La fonction Save()

permet de sauvegarder le Transform de l'objet à savoir la position, rotation et la taille de l'objet. Ces fonctions sont overridées dans les classes filles pour enregistrer plus ou moins de données.

La fonction Save() de l'objet sauvegardé fait appel au GameDataWriter.Write() pour enregistrer différentes données (par exemple le transform.position).

L'utilité de passer par un GameDataWriter est de coder une surcouche du BinaryWriter plus simple à utiliser. En effet, le BinaryWriter, classe interne à Unity, ne peut enregistrer que des types primitifs (int, char, byte…). Il faut donc « décomposer » les types non primitifs des objets pour les sauvegarder. Et cette décomposition se fera dans le GameDataWriter. Par exemple, si je veux sauvegarder la rotation de mon PersistantObject, je la passe en paramètre à la méthode GameDataWriter.Write(). Cette fonction est surchargée pour enregistrer le plus de types possibles. On peut lui envoyer un Quaternion comme un GameObject.

La rotation d'un objet est un Quaternion, ce n'est pas un type primitif, on ne peut pas l'inscrire tel quel dans le fichier binaire. La fonction Write se charge alors de décomposer le Quaternion en 4 valeurs : x, y, z et w, toutes des float. Ces 4 valeurs sont ensuite sauvegardées dans le BinaryWriter

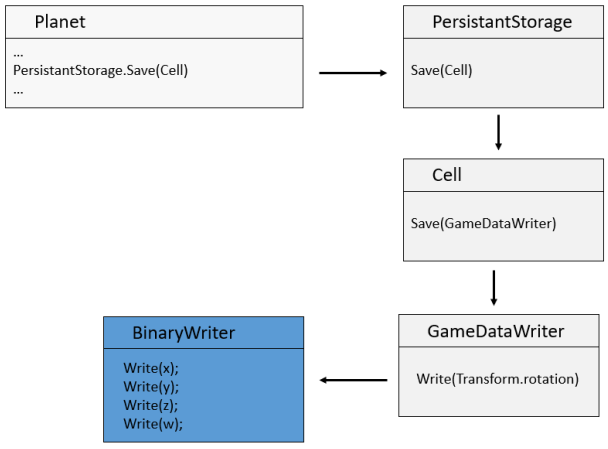


Figure 20: Schéma de la sauvegarde d'une rotation

Une fois la sauvegarde effectuée, on peut observer le contenu du fichier (fig. 21).

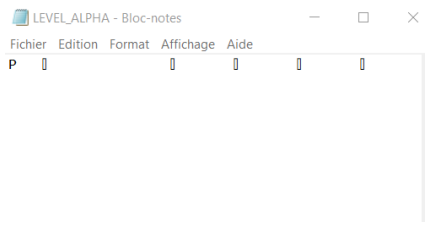


Figure 21: Sauvegarde du niveau du Planet Editor, ouverte avec le bloc-notes

Illisible par un éditeur de texte.

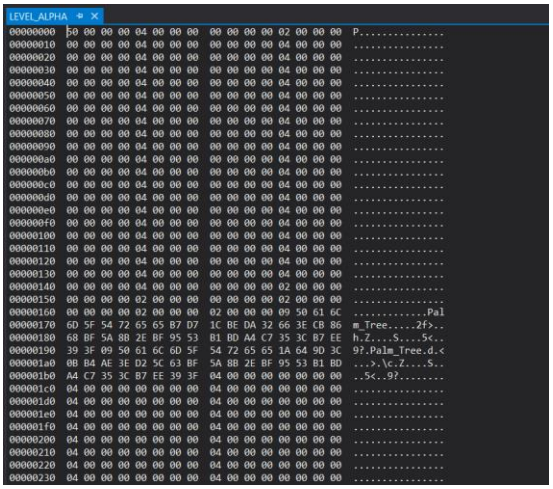


Figure 22: Sauvegarde du niveau du Planet Editor, ouverte avec Visual Studio

La sauvegarde est lisible dans un lecteur de binaire, mais incompréhensible.

J'ai donc un fichier de sauvegarde léger (< 1 ko), facile et rapide à modifier, et protégé.

4.3 Bilan

La sauvegarde est vraiment une partie importante lors du développement du logiciel/outil. Il y a plusieurs méthodes de sauvegarde possibles, plus ou moins adaptées à la situation.
J'ai fait le choix du binaire car je n'ai besoin de sauvegarder et de ne charger qu'une seule chose : le niveau. Dans d'autres situations, il aurait été pertinent de passer par un fichier JSON ou XML comme un éditeur de dialogue pour charger des parties plus spécifiques ou pour faciliter la lecture et écriture du document dans à avoir à tout réécrire.

CONCLUSION

Ce sujet m'a permis de traiter les problématiques liées au développement d'outils appliqués au Level Editor. Cela m'a fait considérer l'importance de l'ergonomie en plus du côté technique pour être vraiment efficace et apporter une vraie plus-value. Cependant, j'ai été déçue de Qt. Le logiciel est déprécié et offre trop peu de features par rapport à un moteur de jeu.

Bien que j'ai réussi à pousser le Planet Editor aussi loin que je le souhaitais, j'aurais voulu traiter d'autres exemples de Level Editor comme des éditeurs de dialogue ou du visual scripting, mais cela demandait un développement aussi vaste que ce travail d'expérimentation

Durant mes recherches, j'ai été amenée à m'intéresser à des sujets plus éloignés de la programmation outil. Certains ont retenu mon attention, je comptais en parler dans ce travail d'expérimentation mais le sujet était tellement vaste qu'il aurait nécessité un autre écrit de même taille. Par la suite j'aimerais bien traiter le sujet du Test Automatique et voir comment le coupler avec des logiciels de versioning et des moteurs de jeu.

REMERCIEMENTS

Je remercie l'équipe pédagogique de m'avoir laissé travailler sur ce sujet et mes camarades pour le soutien. Merci à Florian pour son support lors de mes sessions de travaux expérimentaux sur Qt, Sébastien pour son hébergement, Océane et Noëmie pour avoir répondu à mes questions sur les attentes du Tool Programming.

BIBLIOGRAPHIE

https://www.qt.io/.
https://docs.unity3d.com/Manual/index.html
https://catlikecoding.com/unity/tutorials/object-management/persisting-objects/

REFERENCE

https://github.com/Valefors/LevelEditorQt
https://github.com/Valefors/PlanetEditor