Thibaut HUNCKLER - 2013

Projet Unity3D

-

Virus Inside

Ce document a pour but d'expliquer le fonctionnement du jeu Virus Inside.

Virus Inside est un jeu d'infiltration/plateforme, développé par moi-même en environ deux semaines (à temps plein). Son développement a été pour moi l'occasion de développer différents outils et de créer un « moteur » adapté surchargeant Unity3D pour me faciliter la tâche par la suite.

Ainsi, je m'attarderai tout d'abord à la présentation plus précise de Virus Inside, puis décrira les outils développés avant d'expliquer le fonctionnement global du jeu.

I/ Présentation

Virus Inside est jeu en 2,5D (gameplay 2D dans un environnement 3D), mettant le joueur dans la peau d'un virus informatique devant corrompre le cœur de la machine dans laquelle il s'est infiltré. Pour cela, il devra éviter différents ennemis, et rester caché pour atteindre son but. Le gameplay s'axe donc sur l'infiltration, mais aussi sur la plateforme, puisque les reflexes du joueur sont mis à l'épreuve avec des phases nécessitant vitesse et timing.

Le jeu intègre aussi un aspect scoring, avec un système de points s'incrémentant lors de la récupération d'items, et se décrémentant lorsque le joueur perd. Le temps de jeu est aussi pris en compte, et donne un challenge intéressant pour le speed running.

Virus Inside est un jeu prévu au départ sur PC/Android (car oui, j'avais oublié que nous devions le rendre en version Web), mais est aussi jouable en version Web par respect des consignes.

II/ Les outils

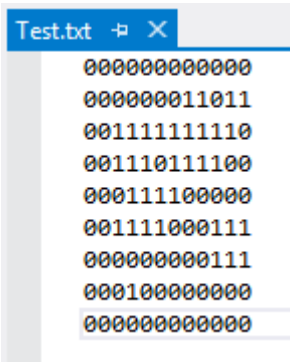
Afin de développer le jeu dans les meilleures conditions, j'ai préféré commencer le développement du jeu par la mise en place de plusieurs outils. En effet, je me suis dit que ce projet pourrait me servir de tremplin pour commencer un projet finalement plus vaste.

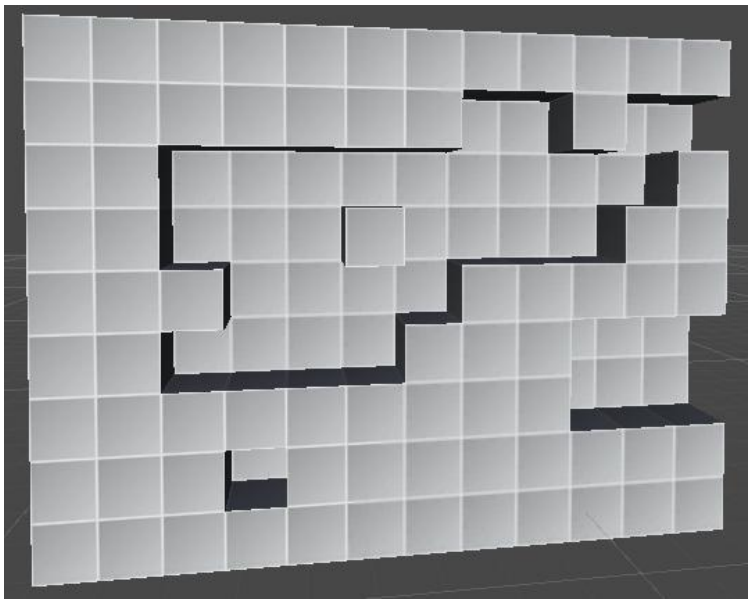
Le but était donc simple : me faciliter la vie pour le futur, le but de tout programmeur (feignant et donc productif) en somme. Pour cela, l'essentiel consistait à la création/modification facile de nouveaux niveaux, et à leur mise en place rapide.

II/ a) Outil de création de niveaux

Pour la création des niveaux, j'ai mis en place un outil intégrable dans l'interface Unity3D, chargeant un fichier et créant le niveau en fonction des informations que ce fichier contient.

Voilà ce à quoi ressemble le contenu d'un fichier et le résultat obtenu :





L'outil crée donc le niveau en se basant sur des cubes, mais prend garde à ne créer que les faces utiles, afin de simplifier au mieux la complexité de la scène (bon, ce ne sont que des cubes, mais quand on peut optimiser, pourquoi se priver ?). De plus, il se charge de créer les boîtes de collision de la manière la plus optimale possible (= en créant le moins de BoxCollider possible). Cela représente un gain de temps (juste) ENORME, car pour le niveau 2 par exemple, il aurait fallu créer plus de 73 Box Collider à la main… !

II/ b) Gestion de la caméra

La gestion de la caméra représente un atout majeur, puisqu'il permet des effets très agréables aux joueurs, et offre la possibilité de n'afficher au joueur que ce que l'on souhaite, et en fonction d'où il se trouve.

J'ai donc mis en place un outil permettant une gestion assez fine de la caméra. Le comportement de la caméra est donc défini en fonction de la zone dans laquelle le joueur se trouve. Elle peut être fixe ou mobile. Si elle est fixe, elle se place à une position prédéfinie jusqu'à ce que l'autre joueur aille dans une autre zone. Si elle est mobile, elle suivra le joueur en restant à une certaine distance. Elle peut être bloquée à une certaine valeur en x ou en y.

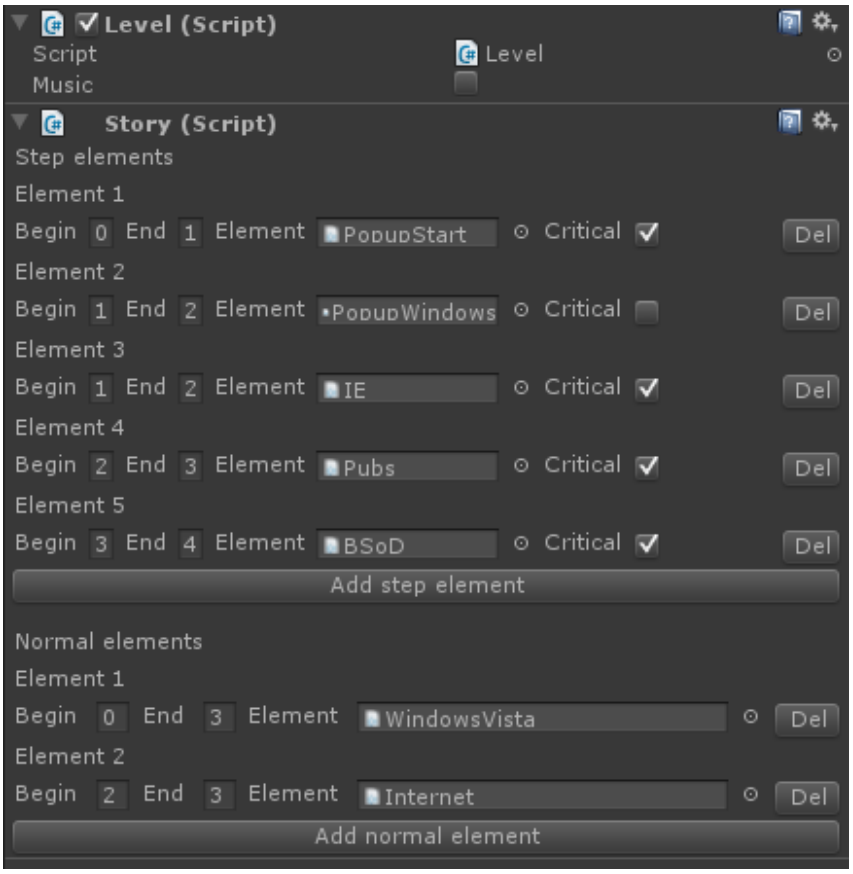
Pour la définition de ces zones et du comportement de la caméra de celles-ci, j'ai crée un script modifiant l'Inspector du script se chargeant du stockage des zones et comportements (ce dernier script tient au courant le script gérant directement la caméra si le joueur change de zone, pour lui donner les nouvelles valeurs).

Voici ce à quoi ressemble l'Inspector :

On peut donc choisir de créer une nouvelle zone fixe ou une nouvelle zone mobile à tout moment. Il faut ensuite définir leur limites en x et en y. Puis, dans le cas d'une zone mobile, il faut définir la distance à laquelle doit être la caméra, puis si elle doit être fixée en x ou en y. Dans le cas de la zone fixe, fournir une position par l'intermédiaire d'un Transform. Pour aider ce procédé, le bouton « Create spot » permet de créer automatiquement un spot là où se trouve la caméra dans la scène. Ainsi, il suffit de placer la caméra comme on le souhaiterait, puis d'appuyer sur ce bouton pour créer un GameObject avec la bonne position et la bonne orientation. Il ne reste alors plus qu'à la fournir à la zone fixe.
Pour faciliter la visibilité, chaque zone peut être définie comme la « actuelle », elle s'affiche alors en jaune dans l'Inspector, et le gizmo affiché dans la scène devient une zone jaune remplie (et non seulement un cadre bleu ou rouge).

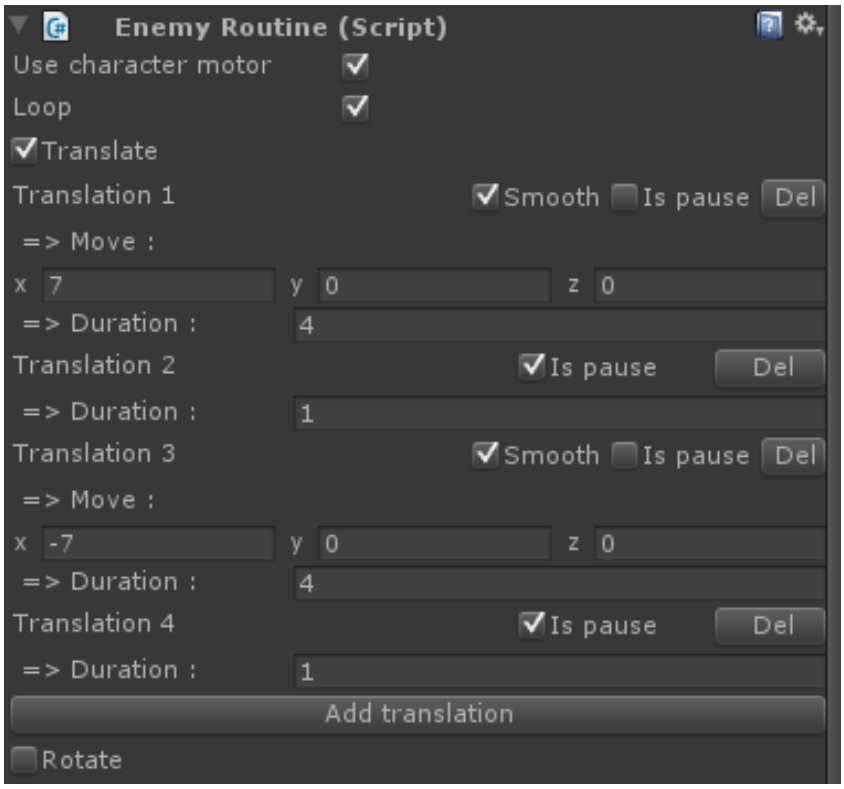
II/ c) Gestion d'histoire/d'événements

Cet outil m'a permis de pouvoir définir une succession d'événements au sein d'un même niveau. Ces événements possèdent chacun une étape de départ et d'arrivée. Ils peuvent être définis comme critiques ou non. Tous les événements critiques qui se terminent à la même étape doivent être finis afin que l'histoire passe à cette fameuse étape, et que de nouveaux événements se lancent. Lors du changement d'étape, les événements non critiques qui se terminent à cette étape sont considérés comme vu et sont supprimés.
Dès qu'il n'y a plus de nouveaux événements à lancer, c'est que le niveau est fini.



II/ d) Gestion des routines d'ennemis

Cet outil me permet de définir la routine des ennemis, comme une succession de translations ou de rotations, à lire une seule fois ou en boucle :



L'ensemble de ces outils ont été (et sont) des atouts de taille pour la suite du développement, et m'ont permis d'économiser énormément de temps. De plus, pour ce qui est des gestionnaires de caméra et d'histoire, ils sont suffisamment génériques pour pouvoir être utilisés pendant d'autres de mes projets.

III/ Fonctionnement du jeu

Afin de pouvoir clairement expliquer le fonctionnement du jeu, je vais donner une vue d'ensemble de mon code, en décrivant chaque pièce maîtresse.

Le « chef d'orchestre » du jeu est le **GameManager**. Il se charge du fonctionnement global du jeu. Il respecte le design pattern du Singleton, et peut être accessible par toutes les instances du jeu, car l'instance du GameManager est stockée (en read only) dans une variable statique.

Le **SaveManager** permet la sauvegarde des données du joueur, que ce soit sur PC/Tablette (fichier, non scripté pour le moment) ou sur navigateur Web (Clés de registre). La sauvegarde concerne l'avancement de la partie du joueur, et les scores réalisés lors de sa partie. Ainsi, si le joueur ouvre la page de navigateur du jeu alors qu'il avait déjà joué précédemment, il pourra reprendre sa partie au niveau où il était, et au dernier checkpoint de ce niveau qu'il avait rencontré. Il reprendra aussi son score et son temps de jeu, et les items déjà récupérés ne sont plus présents (pas de triche possible, désolé). Les données sont stockées dans le **ProgressData**.
La classe **CheckPoint** contient les informations de chacun de checkpoint du niveau courant, afin notamment de ne pas réactiver les items déjà récupérés.

Le **CameraManager** se charge de la gestion de la caméra, elle communique avec le **CameraMovements** pour changer le comportement de la caméra en fonction de la zone dans laquelle se trouve le joueur (voir II/b)). Les différentes zones et comportements reliés sont gérés grâce aux MovableCameraZone, qui peuvent être des FixedCameraZone ou des MovableCameraZone (les noms sont plutôt explicites).

L'**InputManager** s'occupe de la gestion des inputs. J'ai eu l'occasion de développer cette classe et ce qui l'entoure pendant mon temps libre avant ce projet. Globalement, la classe permet une gestion simple et générique des inputs, et ce quelque soit le périphérique et la plateforme de jeu. L'inputManager suit lui aussi le design pattern du singleton et peut être accessible de toutes autres instances du jeu. Cela permet d'accéder aux différentes actions réalisées par le joueur à partir d'autres scripts, sans se soucier du périphérique utilisé.

La classe **Level** est appliquée à chaque niveau. Elle permet de superviser la gestion de l'histoire et des éléments le composant. Elle previent également le GameManager lorsque le niveau est terminé.
La classe Story permet l'enchaînement d'événements d'histoire. Elle est mise à jour par la classe Level, et la tient au courant si tous les événements du niveau ont été joués (= fin du niveau).
La classe **StoryEvent** représente ces événements d'histoire (voir II/c)).

La classe **Menu** permet le bon fonctionnement du menu principal et du menu pause. Elle est constituée de MenuScreen, eux-mêmes constitué de MenuButton. Ces MenuButton

permettent de passer de MenuScreen en MenuScreen, ou de prendre des décisions sur le jeu (reprendre la partie, lancer le jeu, etc.). Lors de ces décisions, le Menu prévient le GameManager pour orchestrer la suite des événements.

La classe **Element** possède toutes les informations pour tous les éléments du jeu. Les CollectibleElement sont nécessaires pour les items à récupérer, et les InteractiveElement pour les éléments avec lesquels le joueur peut interagir.

La classe **Character** possède toutes les informations pour tous les personnages du jeu, le Virus comme les ennemies.
De même, les classes Character_Animator, Character_Interaction, Character_Motor et Character_Physic gèrent respectivement les animations, les interactions les mouvements et la physique des personnages.
Les classes Hero et Hero_X surchargent ces fonctions pour le fonctionnement plus spécifique du Virus (Il en est de même, dans une moindre mesure, pour les ennemis).