Rapport P2 Qt

Damian Petroff, Sergiy Goloviatinski, Raphaël Margueron

16 janvier 2018

Table des matières

Chapitre 1

Introduction

Chapitre 2

Analyse

2.1 Spécifications

Chapitre 3

Conception

Chapitre 4

Développement

4.1 Génération des sols

Pour la gestion du terrain, nous avons décidé d'avoir deux types de sol : l'eau et l'herbe. Sur l'herbe, il est possible d'ajouter des bâtiments et sur l'eau non.

Pour pouvoir avoir des cartes auto-générée, nous avons décide d'utiliser un algorithme de bruit de Perlin. Nous n'allons pas aller dans les détails du fonctionnement du bruit de Perlin, dans ce rapport. Mais son principe est le suivant : il permet d'avoir des nombres aléatoires qui sont proche les uns de autres en function d'une seed(pour l'aléatoire, initialisé au début) et de 1, 2 ou X dimensions (trois dans celui que nous avons utilisés, mais que 2 utilisés). Vu que les outputs de la function de bruit de Perlin sont les mêmes quand nous utilisons les mêmes inputs. Nous pouvons la considérer comme une fonction à plusieurs variables (qui sont les dimensions et la seed). Changer la seed donne des résultats complètement différent mais changer les autres variables change peu entre chaque valeur.

$$bruit = perlin(seed, x, y, z)$$

Les nombres aléatoires que nous récupérons de la function sont des nombres flottant entre 0 et 1. Leurs distribution sont sous la forme d'une distribution normal.

$$y = e^{-x^2}$$

Pour pouvoir générer la carte nous parcourons toutes les cases de la carte (selon x et y). Et "nous déplaçons sur le fonction de Perlin" d'un certain offset (qui est un multiple des index des cases).

4.2 Gestion des cases de la carte

Nous stockons les cases de la carte dans un $QArray$ d'une longueur $nbCases^2$

Une case à la coordonnée $X;Y$ dans la grille de cases représentant la carte correspond à l'élément se trouvant à l'indice $X + YnbCases$ dans le $QArray$

Une case de la carte est représenté par la classe $MapTile$ dérivé de la classe $QGraphicsRectItem$ et contient les attributs suivants :

- bOccupied (boolean) : dit si la case est occupée par un bâtiment ou de l'eau ou pas
- x (int) : coordonnée en x dans la matrice contenant les MapTile
- y (int) : coordonnée en y dans la matrice contenant les MapTile
- bId (int) : id du type de bâtiment (tous les hôpitaux ont la même bId p. ex, valeur par défaut : -10)
- unique BId (int) : id unique du bâtiment (chaque bâtiment posé aura une id unique différente, valeur par défaut : -10)
- mainTileX (int) : coordonnée en X de la case principale (en haut à gauche), utile pour les bâtiments d'une largeur ou hauteur supérieures à 1 (valeur par défaut : -10)
- mainTileY (int) : même chose que mainTileX mais la coordonnée Y
- buildingWIdth (int) : largeur en nombre de cases d'un bâtiment (valeur par défaut : -10)
- buildingHeight (int) : hauteur en nombre de cases d'un bâtiment (valeur par défaut : -10)
- bPix (boolean) : indique si la case a une texture ou pas (valeur par défaut : false)
- buildImage (QGraphicsPixmapItem*) : Pointeur vers la texture du bâtiment/route qui se trouve sur la case

4.3 Routes et bâtiments

Pour la gestion des routes et des bâtiments nous sommes parti sur une approche plutôt simple qui est d'avoir divisé en trois classe qui chacune des rôles spécifique. Une qui décrit les caractéristiques propre à chaque bâtiments : les constantes -> ConstantBuilding. Une seconde classe qui décrit une bâtiment spécifique à la partie -> Building. Et une troisième qui sert de gestionnaires de building, 'classe tableau' -> BuildingManager.

4.3.1 Constant Building

Constant Building contient une function de construction d'un tableau d'objet de 'soit même'. Chacun de ces objet représente un bâtiment avec toutes ces caractéristiques internes, par exemple : le prix de construction, la liste des bâtiments nécessaires pour sa construction ou encore sa catégorie dans l'affichage de l'HUD.

Liste de caractéristique de base de chaque bâtiment :

— Nom à l'affichage
— Catégorie
— Prix
— Largeur (au sol)
— Hauteur (au sol)
— Une liste de case ignoré (non-implémenté, pour avoir la possibilité d'avoir de bâtiment non rectangulaire)
— Type de pré-requis (Ou / Et)
— Une liste d'identifiant de bâtiment pré-requis.
— La sommes des pré-requis nécessaires.

Nous avons choisi d'avoir des objets au lieu d'une liste de constante, pour pouvoir avoir la possibilité d'y ajouter des fonctions pour ces objets ce qui nous a permis d'avoir un autre set de valeur basé sur les caractéristiques de base, voir ci-dessous. Cela nous permet ajuster facilement le jeu et d'avoir une sorte de progression linaire du jeu.

Liste des caractéristique dérivé :

— Prix/Secondes = $Prix/4$
— Efficacité = $(Prix/10)^{1.4} + 10$ (arrondi au multiple de 25 le plus proche)
— Rayon d'action = $Log10(PrixParSecondes * Efficacite + 1)$ (Arrondi au multiple de 5 le plus proche)
— Poids du Bâtiment dans les pré-requis = $Prix/10$

On remarque donc que le prix du bâtiment influe directement sur tous les caractéristique qui définisse si le bâtiment est performant ou non.

Fonctionnement des pré-requis : Les pré-requis sont un aspect que nous considérons comme l'élément primordiale permettant au jeu d'avoir une progression. Le principe est le suivant.

Pour qu'un bâtiment soit accessible à la construction il faut qu'il respect les règles suivantes si son type de pré-requis est 'ou' : La somme des poids de tous les bâtiments qui sont actuellement posé qui sont dans la liste des bâtiments en pré-requis du bâtiment désiré doit être supérieur ou égal à la somme des pré-requis nécessaires. Exemple : Pour poser un Laboratoire médical : Il faut au moins une somme de 6. La liste des bâtiments pré-requis et des poids pour le laboratoire est la suivante :

— Clinique : 1
— Hôpital : 5

Il y a donc au moins trois possibilités pour arrivé à remplir cette demande :

— 6x Clinique
— 1x Clinique et 1x Hôpital (Meilleur solution, au niveau du prix par seconde de le efficacité sur un grand rayon)
— 2x Hôpital

Pour les pré-requis en 'et' : Il faut en plus que pour les pré-requis en 'ou', avoir au moins un bâtiment de chaque type de la liste des bâtiments pré-requis. Exemple : Pour poser une Tour Eiffel : Il faut au moins une somme de 11.5 La liste des bâtiments pré-requis et des poids pour le laboratoire est la suivante :

— Hôpital : 5
— Caserne de pompier : 1.5
— Quartiers généraux : 5

Il y a donc au moins une possibilité pour arrivé à remplir cette demande, qui est d'avoir une bâtiment de chaque. A noter que dans ce cas, la somme de pré-requis n'influence pas pour ce bâtiment, mais on pourrai imaginer qu'un bâtiment aille une somme plus grande que la somme de chaque bâtiment pré-requis et la la somme aurai une influence.

4.3.2 Building

La classe building décrit un bâtiment posé dans une partie. Il est décris les caractéristiques suivantes :

— Un identifiant unique
— Un identifiant du type de bâtiment (de ConstantBuilding)
— Une position X/Y
— Un angle de rotation (De quel côté est posé le bâtiment, non-implémenté)
— Une population

On peut donc noter que cet objet est rendu assez 'léger' car la plus part de ce qui le décrit est géré par l'identifiant constant building. Nous pouvons donc avoir beaucoup de bâtiment et avoir une emprunte mémoire peu conséquente.

4.3.3 Building Manager

Cette classe permet de gérer une liste de bâtiment de la classe Building. On peut exécuté les actions suivante :
— Ajouter un bâtiments (qui est ajoutable selon les pré-requis)
— Supprimer des bâtiments
— Récupérer la population total de la ville.
— Récupérer le bonheur moyen de la ville.
— Récupérer la somme des prix par seconde.

Les fonctions de récupération retournent un valeur stocké mais la valeur stock est recalculé si le besoin l'est, par exemple le bonheur moyen ne varie pas selon le temps mais si un nouveau bâtiment est placé on doit donc recalculer le bonheur moyen.

4.4 Déroulement en fonction du temps de la partie

Le jeu se déroule en fonction du temps sur divers aspects, cela permet d'avoir une progression. Toute les secondes certain calcules sont effectuer pour donner l'impression au joueur que le temps défile. Voici les trois indicateurs que nous avons choisis de représenté :
— Bonheur : Représente une satisfaction de la population sur la ville construite.
— Argent : Le solde que la ville a à disposition pour faire de modifications.
— Population : Le nombre d'habitant qui réside actuellement dans la ville.

Malgré nos efforts à rendre le jeu jouable. Nous avons parfaitement conscience que le jeu n'est pas très bien équilibré à ce niveau.

4.4.1 Temps : Heure et Date

Pour pouvoir donner un aspect réaliste au jeu, nous avons choisi de prendre la date du jour comme date du début de jeu. Puis toute les secondes, le temps virtuel augmente de une heure. Nous avons donc pour 60 minutes de jeu effectives correspondent à 150 jours virtuel.

4.4.2 Bonheur

La jauge de bonheur global est un nombre qui varie entre 0 et 250. Il est basé sur la moyenne du bonheur de toutes la résidence de la ville.
Pour chaque résidence, un bonheur résidentiel est calculé. Les paramètres suivant influence le bonheur résidentiel :
— Les bâtiments public autours de la résidence.
— La proximité de ce bâtiment public.

Plus y il a de bâtiment efficace et proche d'un résidence pour le bonheur sera élevé pour cette résidence. Les paramètres suivant influence le bonheur global.
— Le bonheur résidentiel moyen
— Les impôts en vigueur actuellement

Concernant les impôts nous avons fixé un multiplicateur à 100% qui est au point neutre des impôts (8.0%). Et qui peut varier entre 0%-200%. Plus le impôts sont bas plus les habitants seront content et à l'inverse plus les impôts sont élevé moins les habitants sont content.
Pour ajouter un effet de lissage sur le temps du bonheur, nous avons implémenté un algorithme assez simple qui est le suivant :

$$bohneur = 20\% ancienBohneur + 80\% bohneurResidentielMoyen$$

L'ancien bonheur correspond au bonheur qui était en vigueur la seconde précédente.

4.4.3 Population

Nous avons actuellement une population fixe mais il avait été prévu d'avoir un accroissement de la population en fonction du bonheur. Ainsi que des logements qui se construisent automatique au bords des routes en conséquence.

4.4.4 Argent

Pour l'argent il y a principalement deux facteurs qui influence les revenus ou non de la ville.
— En négatif : Le prix par seconde des bâtiments public installés
— En positif : Le impôts prélevé aux habitants.
Pour les revenus des impôts, nous avons fais une règle un peu fantaisiste qui est que les habitants payent des impôts relatif à leur satisfaction à la ville. Ce qui impose au joueur d'avoir une ville heureuse pour pouvoir avoir de l'argent. Ce qui nous évite permis de contourné le faites que les flux de population ne sont pas géré. Car initialement les revenus n'était pas censé être influencé par le bonheur mais seulement par la population. Ce qui nous apportai une liaison en triangle entre les trois indicateurs. Selon ce schéma Bonheur -> Population -> Argent -> Bonheur(Indirectement) et la boucle est fermé.

4.4.5 Dérivé des indicateurs

Lors du déroulement en fonction du temps nous avons trouvé intéressant d'avoir d'affiche la variation de chaque indicateurs entre les deux derniers calculs (la dérivée en fonction du temps). Calculé de cette façon.

$$deltaValeur = nouvelleValeur - ancienneValeur$$

4.5 Gestion du son

Pour la gestion du son nous avons décider de pouvoir traiter facilement les musiques et les bruitages du jeu avec un mixeur. Nous avons donc crée une classe. L'HUD supérieur permet de la piloté à l'aide de trois sliders : Un pour le contrôle du volume de la musique, un pour celui des bruitages et un troisième englobant les deux, appelé master.

4.5.1 Gestion des sauvegardes

Pour la sauvegarde des parties, nous avons opté pour un système de enregistrement par fichier JSON. Ce qui nous permet d'avoir les données répartis sous forme hiérarchique et Qt possède toute une palette de fonction pour gérer des fichier JSON. Nous n'avions aucune connaissance de ce format.
Nous avions prévu deux types de fichier de sauvegarde : Un concernant une partie et un autre pour les configurations, celui-ci est le même pour n'importe quelles parties. Nous n'avons malheureusement pas eu le temps d'implémenter la sauvegarde de la configuration.
Quand aux données à sauvegarder, nous avons essayer de stocker un minimum d'information, pour éviter toute redondance. Comme par exemple le bonheur de la ville peut-être calculé avec nos fonctions. Il n'est donc pas sauvegarder dans le fichier. Un autre exemple qui est assez 'puissant' et nous permet d'éviter beaucoup d'information est la topographie de la carte, car en effet vu qu'elle est généré aléatoirement il nous suffi de stocker la seed et nous aurons toujours le même résultat.
Vous pouvez trouver un exemple de fichier de sauvegarde dans les annexes.

4.6 Launcher

Chapitre 5

Tests

5.1 Protocoles de tests

Chapitre 6

Bilan

Chapitre 7

Conclusion

Chapitre 8

Source

Pour le projet en général, nous avons principalement utilisé la documentation en ligne du framework Qt pour le développement.
Pour le utilisation du bruit de Perlin, nous avons consulté une série de vidéo tutoriel en javascript (avec le framework p5) de la chaîne Youtube "the CodingTrain" par Daniel Shiffman qui a couvert le sujet.
Pour ce qui est de l'algorithme de Perlin (en lui même) utilisé, il a été trouvé sur le compte Github de Sol-program : `https://github.com/sol-prog/Perlin_Noise`

Chapitre 9

Annexes