Slay and Conquer

Paul Boursin David Hamelin Simon Pellicer Hugo Spinat

5 mai 2019

Table des matières

0.1 Présentation du jeu

Pour notre projet d'informatique, nous avons décidé de créer Slay And Conquer, un jeu de stratégie basé sur *Slay*, en ajoutant des mécaniques du jeu *Civilization*. C'est un jeu tour par tour dont l'objectif est de capturer toutes les villes ennemies.

La carte est composée d'hexagones réguliers, de hauteur variable. Monter est plus difficile que descendre ; Etre situé en hauteur est un avantage majeur, comme dans une vraie bataille médiévale.

Chaque joueur débute avec une ville. Les villes ont deux utilités :
— Elles génèrent une quantité fixe d'or à chaque début de tour.
— Elles permettent de créer des unités.

Les unités ont un coût à la création (précisé sur le menu), ainsi qu'un coût de maintenance qui se prélève au début de tour. S'il ne reste pas assez d'or pour la maintenance d'une unité, elle désertera.

Si une unité se déplace sur une ville et y reste jusqu'au tour suivant, la ville est conquise. L'unité ne pourra pas bouger pendant ce tour.

Il existe trois types d'unités :
— Guerrier.
— Archer.
— Cavalier.
Chacune de ces unités a un coût et des capacités différents.

Les unités peuvent se battre entre elles. Les guerriers et cavaliers se battent au corps-à-corps, alors que l'archer peut se battre à distance.

Lorsque une unité en affronte une autre au corps-à-corps, les points d'attaque de l'attaquant et les points de défense du défenseur déterminent les dégâts infligés. Le défenseur effectue ensuite une riposte, qui est similaire à une attaque, excepté que les points de riposte remplacent les points d'attaque.

Quand un joueur ne possède plus ni unité, ni ville, il perd. Le joueur restant gagne la partie.

1 Choix des technologies

1.1 *C++*

Notre choix pour le langage d'implémentation du jeu s'est porté sur *C++*, un langage de plus bas niveau que *Java*.
Le *C++* est beaucoup plus complexe que *Java* ; il n'y a par exemple pas de *garbage-collector*, c'est à dire que la gestion de la mémoire se fait manuellement :une mauvaise manipulation de pointeur peut créer des corruptions ou des fuites de mémoire.

Nous avons choisi ce langage pour deux raisons :
— Le *C++* est un standard dans l'industrie du jeu vidéo. Nous avons décidé de nous donner un défi supplémentaire en utilisant ce langage sur un projet non trivial.
— Les autres technologies que nous souhaitions utiliser sont faites pour être utilisé en *C++*. L'intégration avec *Java* aurait pu être possible, mais fastidieuse (et aurait privé *Java* de sa portabilité).
De plus, nous avions tous un peu d'expérience en *C++*, et nous avions pris la précaution de vérifier la faisabilité du projet, notamment l'intégration des librairies que nous utilisons dans le projet.

1.2 *Vulkan*

Nous avons décidé très tôt que notre jeu serait en 3D ; Nous avions plusieurs choix. Il aurait été possible d'utiliser une librairie de haut niveau (de type *JavaFX*). Ce type de librairie permet d'afficher directement des primitives 3D, sans se soucier de la carte graphique ou autre. Cela

offre une abstraction au détriment de la performance.

Nous avons décidé de prendre une librairie plus délicate à utiliser, mais beaucoup plus performante : *Vulkan*.

Vulkan est le successeur de *OpenGL* ; Il est beaucoup plus proche de la carte graphique, et permet de maximiser les performances. Contrairement à *JavaFX*, il n'y a quasiment pas d'abstraction : Le code (minimum) pour afficher un simple triangle (sans texture) est de 800 lignes.

Paul a énormément de connaissance en *OpenGL* et en *Vulkan*, et a notamment créé pour *l'INRIA DigitalBrain.org*, en utilisant WebGL (*OpenGL* ES, mais via le navigateur). Il a également de l'expérience en Vulkan (rendu traditionnel, et raytracing). Il s'est chargé de l'écriture du moteur de rendu Vulkan de notre jeu.

1.3 Entity-Component-System

Après avoir vu la conférence d'un développeur du jeu vidéo *Overwatch* à la GDC 2017, Nous avons décidé d'utiliser un nouveau paradigme pour notre jeu vidéo : l'*Entity-Component-System* ou ECS.

Le paradigme Objet sert à résoudre le problème des répétitions, en répartissant le code dans des classes. On structure le code en un arbre d'héritage. Ce paradigme est extrêmement puissant ; Cela permet d'exprimer des structures complexes comme des extensions de structures plus simple (Une *LinkedList* comme enfant de *List* par exemple).

Le problème est que ce paradigme n'est pas très pratique pour les jeux vidéo ; Le comportement et les caractéristiques d'une entité peuvent changer. Par exemple, dans notre jeu, il est possible de sélectionner une ville, ou une unité. On pourrait créer une classe *Selectable*, avec un booléen *protected isSelected*, et une fonction *select()* qui inverse le champ *isSelected*. Le problème de cette approche est que *City* et *Unit* sont également des classes descendantes de *Entity*. Doit-on faire de *Selectable* un enfant de *Entity* ? Ou l'inverse ? Et maintenant, imaginons qu'une entité ne puisse pas être sélectionnée ; Je dois ajouter un champ *isDisabled* à la classe *Selectable*, modifier la fonction *selected()*... Le graphe d'héritage devient très rapidement difficilement compréhensible, et encore plus difficilement maintenable.

L'ECS résout le problème en divisant la logique du programme en trois catégories :
— Entités.
— Composants.
— Systèmes.
Les entités sont des "conteneurs" de composants ; Les composants contiennent des états ; Les systèmes itèrent sur les composants, et peuvent modifier l'état de ces derniers (ainsi qu'ajouter ou supprimer des entités et composants).

Pour résoudre le problème de sélection, avec l'ECS, c'est très simple ; On crée une classe vide *SelectionComponent*, on l'instancie et on l'attache à l'entité qu'on veut sélectionner. Il suffit de faire *reg.has<SelectComponent>(entite)* pour savoir si l'entité est sélectionnée. On peut même itérer sur les entités sélectionnées directement.
Ce n'est qu'un exemple très simple d'application de l'ECS ; Nous profitons beaucoup de ce paradigme dans notre projet.

Nous utilisons la librairie *entt*, qui utilise les *templates* du C++17 moderne ; La librairie est extrêmement simple à utiliser, et très lisible. Pour voir un exemple, allez voir le fichier : *src/logic/systems/gold_system.cpp*

2 Implémentation des fonctionnalités

2.1 Modèle du jeu

Le point d'entrée du jeu est le fichier *main.cpp*. Ce dernier ne fait qu'initialiser les différents composants, en donnant les paramètre importants (taille de la carte, nombre de joueurs).

Le cœur du modèle se trouve dans *game.cpp* : Il contient tous les systèmes, le registre de *entt*, (*"reg"*), les méthodes importantes y sont présentes (gestion des tours, gestions des clics, sauvegarde du jeu...), ainsi que des fonctions importantes qui servent aux différents systèmes.

La classe virtuelle (équivalent de *abstract* en Java) *System* est la classe parente des systèmes. Elle permet aux systèmes d'avoir une référence vers *Game*.

2.2 *Pathfinding*

Le *pathfinding* effectue un parcours en largeur de l'arbre des mouvements possibles. Les cases sur lesquelles on passe sont marquées d'un *PathfindingComponent*, qui stocke le poids au moment de l'itération. Cet algorithme est assez naïf, mais suffisamment efficace.

2.3 Génération procédurale du monde

La génération procédurale du monde se fait avec l'addition de deux bruits de Perlin :
— Un bruit de Perlin faible amplitude, haute fréquence (pour les plaines).
— Un bruit de Perlin haute amplitude, faible fréquence (pour les montagnes).

2.4 Intelligence Artificielle

On itère sur les mouvements disponibles pour les unités en possession de l'IA, et on définit un objectif en fonction de la situation stratégique de la carte.

2.5 Interface Graphique

On utilise *dear Imgui* pour avoir une interface graphique immédiate : toute l'interface est dessinée directement à chaque frame, sans *callback* ; On donne les entrées à imgui, et au moment de dessiner un bouton, on peut vérifier si celui-ci a été activé. Cela simplifie énormément le design du programme, puisque toute l'interface est gérée à un seul endroit.

2.6 Moteur de rendu *Vulkan*

Le moteur de rendu *Vulkan* fait plus de 3000 lignes de code. Beaucoup sont dues à la verbosité énorme de *Vulkan*, et à toutes les subtilités nécessaires pour dessiner une image efficacement.

3 Problèmes rencontrés

3.1 Interface graphique

Nous pensions pouvoir réutiliser la partie librairie graphique de *Qt* pour faire une interface graphique plaisante, mais malheureusement cela n'a pas été possible. Nous avons du utiliser *dear Imgui* à la place, ce qui a été plus simple que nous pensions.

3.2 Déploiement

Nous utilisons *Qt* pour la musique de victoire, les paramètres, et l'initialisation de la fenêtre ; Le problème est que nous utilisons une version spécifique de Qt avec la prise en charge de *Vulkan*. De plus, cette version de Qt dépend de versions spécifiques d'autres librairies. Créer une version portable (utilisable par plusieurs distributions *Linux* différentes) a été un vrai challenge.

4 Pistes d'extension

4.1 Pile réseau

Il devrait être possible d'ajouter une pile réseau pour pouvoir jouer des parties en lignes, étant donné qu'il est possible de sérialiser l'état de la partie.

4.2 Plus d'unités, plus de mécanique

L'avantage de l'*ECS* est que nous pouvons rajouter rapidement et sans trop de difficultés des nouvelles unités, et des nouveaux concepts au jeu. Nous n'avons pas créé toutes les unités que nous voulions ; Mais le design est flexible, donc nous pourrons le faire dans le futur.

5 Conclusion

Même si nous n'avons pas fait tout ce que nous avions prévu dans notre cahier des charges, je pense que le projet est globalement une réussite. Les objectifs cruciaux (Jeux jouable, I.A, Génération Procédurale) ont été atteints. Nous avons pu utiliser ce que nous avons appris les deux dernières années et aller au delà ; Utiliser l'*ECS* en parallèle du paradigme objet permet de comprendre mieux les forces et les limitations du paradigme objet.