Projet de développement : SUPRTS

Documentation Technique

Sommaire :

1. 3D Engine :
2. Internal World Representation Structure:
3. Client/Server Protocol:
4. World Loop / Events Triggered:
5. World Persistence:
6. Le coût de la solution :

3D Engine:

Le moteur de rendu 3D est un framework de développement de jeux: Isogenic Game Engine. Ce framework basé sur l’API Canvas permet le rendu de jeux en 2D et 3D isométrique.

Pour des raisons de performances et de contraintes imposées, l’utilisation d’un framework s’est imposée plutôt que d’implémenter entièrement un moteur de rendu. En effet, la conception d’un moteur de rendu 3D, même isométrique est une tâche qui demande énormément de temps et des compétences spécifiques.

Au vu de nos estimations, il aurait fallu au moins 3 mois de travail (à temps plein) pour rendre le moteur opérationnel (développement et apprentissage de la 3D) et 1 autre mois pour valider les performances et corriger les bugs.

De plus, l’Isogenic Game Engine (IGE) convenait parfaitement à nos besoins :

* Rendu du monde en 3D isométrique ;
* Gestion des entités sur la carte ;
* Gestion du réseau de façon simple et efficace ;
* Intégration native dans nodeJS ;
* Synchronisation des clients et du serveur automatique.

Internal World Representation Structure:

Le monde est représenté par une structure hiérarchique de classes. Le point d’entrée est la classe World. Elle contient toutes les méthodes pour interagir avec le monde sans devoir descendre dans la hiérarchie.

Le monde est lui-même divisé en chunk. Un chunk est une section du monde de 10 cases sur 10. Une case (ou tile) est la plus petite division du monde. Elle contient les valeurs pour l’humidité, la fertilité, la fertilité maximum, le propriétaire éventuel.

Le moteur de jeu charge en mémoire les chunks contenant des entités, c’est-à-dire des joueurs, des plantes ou des bâtiments et travaille sur les chunks directement. De façon interne un chunk est juste une classe container avec des getters/setters pour faciliter le travail, mais ne contient pas de code de logique ou de contrôle.

La génération du monde est gérée par le World par une classe dédiée : le WorldGenerator. Cette classe contient la routine de génération pour les chunks. Le monde est généré par chunk à la volée au besoin.

Client/Server Protocol:

Le protocole de communication entre le serveur et le client consiste en une série de messages. Le côté réseau pur étant géré par IGE. Le code des callbacks de traitement de ces messages se trouve dans ClientNetworkEvent.js et ServerNetworkEvent.js.

Cependant, la plupart des événements réseau sont cachés, car intégrés dans IGE. En effet, les entités se synchronisent automatiquement entre les clients et le serveur ce qui permet de ne pas avoir à gérer tous les déplacements de joueurs, les plantes, les bâtiments.

Les seuls événements traités sont la connexion et les actions.

Pour chaque input d’un client, c’est le serveur qui la traite, ce qui permet de sécuriser et évite d’avoir de la désynchronisation entre le client et le serveur. Ces inputs sont : clic droit, clic gauche.

Pour l’envoi de la carte, le serveur stream les chunks du monde quand c’est nécessaire, et cela de façon automatique au travers d’événements sur les déplacements d’un joueur. Le serveur envoie les 9 chunks situés autour du joueur et cela de façon continue.

World Loop / Events Triggered:

La gestion des événements du world est uniquement événementielle, via la boucle d’IGE. Chaque entité du world contient une méthode tick qui est appelée régulièrement. C’est dans cette méthode que se déroulent les événements relatifs à cette entité. Par exemple pour les plantes, la croissance est gérée par cette méthode. Il en est de même pour toutes les entités du monde nécessitant une évolution dans le temps.

Cette méthode permet de garder le monde cohérent et synchronisé entre les clients et le serveur.

World Persistence:

La persistance du monde s’effectue dans la base de données.

Le world est stocké par chunks. Chaque chunk est stocké dans la table chunk avec pour informations ses coordonnées x et y, et une colonne de data qui correspond à une chaine de caractères. Cette chaine de caractères est la représentation JSON de l’object Chunk. Ceci permet de charger/sauvegarder les chunks dans la database de façon simple et efficace. En effet, un stockage de chaque case de façon individuelle aurait été très lourd pour la BDD, du fait du nombre de cases et de la lourdeur des relations qu’il aurait fallu maintenir pour que cela reste cohérent. L’inconvénient du stockage par chunk JSONifié est la mise à jour qui est plus coûteuse.

Les plantes sont stockées de façon simple, chaque plante au moment de sa création est insérée dans la table plants avec ses coordonnées (x, y), son type, et son pourcentage de croissance. À chaque modification de sa croissance (croissance ou régression), la base de données est mise à jour en conséquence. Si la plante est collectée, l’entrée en base de données est supprimée. Au moment du démarrage, toutes les plantes sont automatiquement chargées en mémoire pour redémarrer le monde.

Les données des joueurs sont aussi stockées en base de données, dans la table users.

Le coût de la solution :

L’équipe de développement se compose de 6 personnes. Au vu des profils recherchés et de l’état du marché du travail, le salaire moyen se situe dans la fourchette 38 – 40K euros par ans. Le temps de développement est d’environ 40 jours (2 mois). Soit un coût de la solution de :

Coût des ressources humaines = (39000 / 12) \* 2 \* 6 = 39000 euros.

À cela s’ajoute la licence d’IGE dans notre cas, 1000 euros.

Soit un coût de la solution de 40000 euros.