

420-C61-IN

PROJET SYNTHÈSE

Fanid, Amine 1370770

Baril, Henrick 2195213

GEEK'S LEGACY

Jeu de plate-forme 2D, inspiré du jeu « Terraria »

Document de conception

Définition technique du projet

Table des matières

420-C61-IN	1
PROJET SYNTHÈSE	1
GEEK'S LEGACY	1
Document de conception	1
Introduction	3
Éléments de conception	3
Infrastructure de développement	3
Interface graphique utilisateur	3
Données persistantes	3
Structures de données	5
Dictionnaires	5
Liste Chainées	5
Matrice	5
Graphe (arbre)	6
Structure à développer	6
Patrons de conception	6
Patron Observateur (Observer)	6
Patron Fabrique (Factory)	7
Patron État (State)	7
Patron à développer	7
Développement d'une bibliothèque	7
Expression régulière	8
Algorithme	9
Mathématique	9
Conception UML	10
Annexe	11

Introduction

« Geek's Legacy » est un jeu de plate-forme 2D, inspiré du jeu « Terraria ». C'est un jeu de genre bac à sable et aventure. Le jeu sera conçu et codé avec la plateforme de développement Unity et en langage C#.

Ce document aura pour but d'éclaircir le côté technique, les besoins et les spécifications nécessaires pour la réalisation de notre jeu. Nous allons fournir avec précision les éléments indispensables à la conception de « Geek's Legacy », et ainsi démontrer les efforts fondamentaux de notre approche technique.

Éléments de conception

Infrastructure de développement

« Geek's Legacy » sera disponible sur la plateforme Windows et sera codé avec le langage C# et développé sur la plateforme de développement Unity. Le choix du langage C# s'est fait assez facilement, étant donné que c'est un langage très compatible avec Unity et qu'il est optimisé pour la programmation orienté objet. Les notions d'encapsulation, d'héritage et de polymorphisme seront absolument indispensable à la programmation de notre jeu. La large documentation concernant C# et Unity disponible sur internet est aussi un avantage et nous donne confiance en notre choix de langage et de plateforme de développement. De plus, en utilisant C# avec Unity, nous bénéficierons de fonctionnalités avancées offertes par le moteur Unity pour le développement de jeux. Unity fournit une suite complète d'outils pour la création de jeux 2D, ainsi que des fonctionnalités pour la gestion des actifs, de la physique, des animations, du son, et plus encore. C'est donc sur Unity que nous allons créer, éditer et tester notre jeu, et c'est Unity qui nous permettra de déployer « Geek's Legacy » sur Windows.








Interface graphique utilisateur

Voir PDF connexe.

Données persistantes

Nous désirons enregistrer nos données persistantes sous format JSON à l'aide de fichier binaires. Nous allons sérialiser les données importantes à l'utilisateur, et utiliser un serveur web pour sauvegarder certaines données des utilisateurs et ainsi pouvoir offrir un service qui affiche un classement de joueurs, selon leur temps de jeu.

Voici la structure de nos tables de données :

player	player_characters	game_materials	game_weapons
id  integer	id  integer	id  integer	id  integer
username varchar	player_id id	name varchar	name varchar
created_at timestamp	name varchar		
password varchar	created_at timestamp		
	time_played long		
		biomes	
		id  integer	
		name varchar	
		description varchar	
game_items	player_inventory		
id  integer	id  integer		
name varchar	player_id integer		
	object_id integer		
	item_name varchar		

La table « player » contient l'id des usagers, leur nom d'utilisateur, la date de création de leur compte et leur mot de passe, que nous allons hacher de notre côté avec une librairie de Unity.

La table « player_characters » contient l'id des personnages créés par les usagers, player_id consiste en l'id de l'utilisateur qui a créé le personnage. La table contient aussi le nom du personnage, sa date de création et le temps de jeu avec celui-ci.

La table « game_materials » contient l'id de tous les matériaux existants dans le jeu. Ainsi que leur nom. (Il y a 3 matériaux différents dans le jeu)

La table « game_weapons » contient l'id de tous les armes/outils que le joueur peut avoir dans sa main, il y en a aussi 3 dans ce jeu. Soit une épée, une hache et une pioche.

La table « game_items » contient l'id et le nom de tous les items dans le jeu que le joueur ne tient pas dans ses mains. Tel qu'un fourneau par exemple.

La table « biomes » contient l'id, le nom et la description des trois biomes présents dans le jeu.

La table « player_inventory » contient l'id du joueur, l'id de l'objet placé dans l'inventaire et le nom de l'objet.

Notice : les varchar seront des strings dans notre base de données.

Voici un exemple de comment une table sera enregistrée en JSON :

```
{
  "player": {
    "id" : 123456789,
    "username" : "Henrick",
    "created_at" : "2024-02-15",
    "password" : "1a53fg6$%af";}
}
```

Structures de données

Dictionnaires

Les dictionnaires sont des structures de données très utiles pour stocker des paires clé-valeur, où chaque valeur est associée à une clé unique. Dans le cadre de notre jeu, les dictionnaires peuvent être utilisés pour stocker diverses informations pertinentes, telles que les caractéristiques des objets ou des ennemis. Il nous a semblé évident d'utiliser ce type de structure de données dans notre projet pour garder les données ou description de toutes les pièces du casse-tête qu'est notre jeu de manière ordonnée.

Pour l'analyse de complexité des dictionnaires :

- Insertion d'une nouvelle paire clé-valeur : $O(1)$.
- Recherche de la valeur associée à une clé donnée : $O(1)$.
- Suppression : $O(1)$.

Liste Chainées

Les listes chaînées sont des structures de données dynamiques qui permettent de stocker une collection d'éléments liés les uns aux autres par des références. Dans le contexte de notre jeu, les listes chaînées peuvent être utilisées pour gérer divers éléments dynamiques, tels que les ennemis ou leurs projectiles.

En ce qui est de l'analyse de complexité de nos listes chaînées :

- Insertion en tête de liste : $O(1)$.
- Insertion en fin de liste : $O(1)$.
- Recherche d'un élément : $O(n)$.
- Suppression d'un élément : $O(n)$.

Matrice

Les matrices peuvent nous servir à gérer et représenter la carte du monde, où chaque case peut contenir des informations sur les blocs, les entités, les obstacles ainsi que les structures. Il sera

alors plus facile de déterminer qu'est ce qui est où et comment gérer chaque bloc, entité, obstacle et structure.

En ce qui est de l'analyse de complexité de nos matrices:

- Accès à un élément : $O(1)$.
- Recherche dans la matrice : $O(n * m)$.
- Insertion ou suppression d'une ligne ou d'une colonne : $O(n * m)$.

Graphe

Les graphes peuvent être utilisés pour structurer les éléments hiérarchiquement, comme les recettes d'artisanat. Par exemple, si un item "X" a besoin de bois dans sa recette, l'item "X" aura le bois comme parents dans le graphe. Nous pouvons alors gérer toutes les options de fabrications de tous les items du jeu avec le graphe.

Les graphes pourraient représenter la carte et les connexions entre les différentes zones du jeu (les biomes). Il serait alors plus facile de lier, de façon lisse, les différents biomes.

En ce qui est de l'analyse de complexité de nos graphes:

- Recherche d'un élément spécifique : $O(n)$.
- Insertion d'un nouvel élément : $O(n)$.
- Suppression d'un élément : $O(n)$.
- Parcours de l'ensemble de l'arbre : $O(n)$.

Structure à développer

La structure de donnée que nous avons décidé de développer de manière approfondie est la **liste chaînée**. Nous pensons qu'il serait très intéressant de voir jusqu'où nous pouvons pousser ce concept pour gérer de manière ordonnée et efficacement les ennemis et leurs projectiles. Ainsi, nous prévoyons maintenir un certain nombre d'ennemi et de projectile autour du joueur et une liste chaînée nous permettrait de faire ceci de façon rapide et efficace.

Patrons de conception

Patron Observateur (Observer)

Étant donné le besoin constant de surveiller les états de plusieurs éléments de notre jeu, tel que les ennemis ou le joueur, nous avons une problématique quand vient le temps de gérer multiples détections à faire dans le jeu. Entre en jeu le patron observateur, celui-ci nous permettra d'établir des relations entre des objets observateurs et observé, permettant d'informer les objets concernés lors d'un changement d'état et que ceux-ci réagissent en conséquence. Par exemple, on pourrait utiliser un observateur pour que les ennemis puissent détecter le joueur à une certaine distance.

Ou encore, on pourrait utiliser un observateur pour détecter les changements d'états (lien avec le patron État), par exemple, lorsque notre joueur court, il consomme plus d'énergie.

Patron Fabrique (Factory)

L'utilisation d'un patron de Fabrique dans notre projet ne peut que faire du sens. En effet, celui-ci nous permettra d'implémenter une délégation de la responsabilité de la création d'objets à des sous-classes ou à des méthodes de fabrique. L'idée serait donc de fournir une interface commune pour créer tous nos objets de jeu (item, ennemi, outil, etc.).

L'utilisation de ce patron nous permettra, dans le futur, d'éviter des heures de refactorisation de notre code lorsque nous voudrions ajouter un objet de jeu. Ce qui, dans le cas d'un projet de grande envergure, est nécessaire pour ne pas perdre de temps. Dans notre cas, cela nous aidera à ne pas perdre notre précieux temps à refactoriser.

Patron État (State)

L'utilisation du patron d'état dans notre projet apporte beaucoup de facilité. En effet, ce patron nous permettra de gérer nous même les états du joueur, ainsi, nous pourrions activer nos transitions, nos animations et nos autres états lorsque nous le souhaiterons et ce, de façon millimétrée.

Ce patron règlera un problème de gestions d'animations et de transitions que Unity peut avoir lorsqu'un joueur est attaqué et attaque en même temps. Bien que Unity gère un peu ce problème à l'aide de son interface graphique, nous souhaitons avoir la gestion totale de nos animations et transitions pour apporter de la fluidité au jeu.

Patron à développer

Nous allons développer le patron **Observateur**, car il nous semble très utile et récurrent. De plus, le fait de le développer nous-même nous permettra de garder un certain contrôle sur les interactions entre nos objets de jeu.

Développement d'une bibliothèque

Après une analyse approfondie du projet, nous avons identifié le module technique de génération de carte comme une composante particulièrement réutilisable. Cette fonctionnalité est cruciale dans le cadre de notre jeu, mais elle peut également être étendue pour être utilisée dans d'autres projets de jeux nécessitant une génération de carte.

La génération de carte est essentielle pour notre projet, car elle permet de créer des environnements de jeu variés et dynamiques, offrant ainsi une expérience unique à chaque joueur. La carte doit être générée de manière procédurale en fonction de différents paramètres tels que la

taille de la carte, la densité des éléments, etc. Cette fonctionnalité est également pertinente dans d'autres contextes de jeu, tels que les jeux de survie, les jeux de rôle, ou les jeux de stratégie, où la création de mondes virtuels est un aspect central du gameplay. D'où l'idée d'en faire une bibliothèque.

Pour rendre notre module de génération de carte générique, nous devrons concevoir une classe flexible et modulaire qui peut être configurée pour s'adapter à différents types de jeux et de paramètres de carte. Cette implémentation générique nécessitera une abstraction soignée des détails spécifiques au projet, tels que les règles de génération spécifiques au jeu actuel, pour permettre une réutilisation maximale dans d'autres contextes. Nous devrons également prendre en compte la performance et l'extensibilité de la classe, en permettant par exemple la génération asynchrone de cartes pour les mondes de jeu de grande taille, et en facilitant l'ajout de nouvelles fonctionnalités de génération de carte via l'héritage ou la composition.

Expression régulière

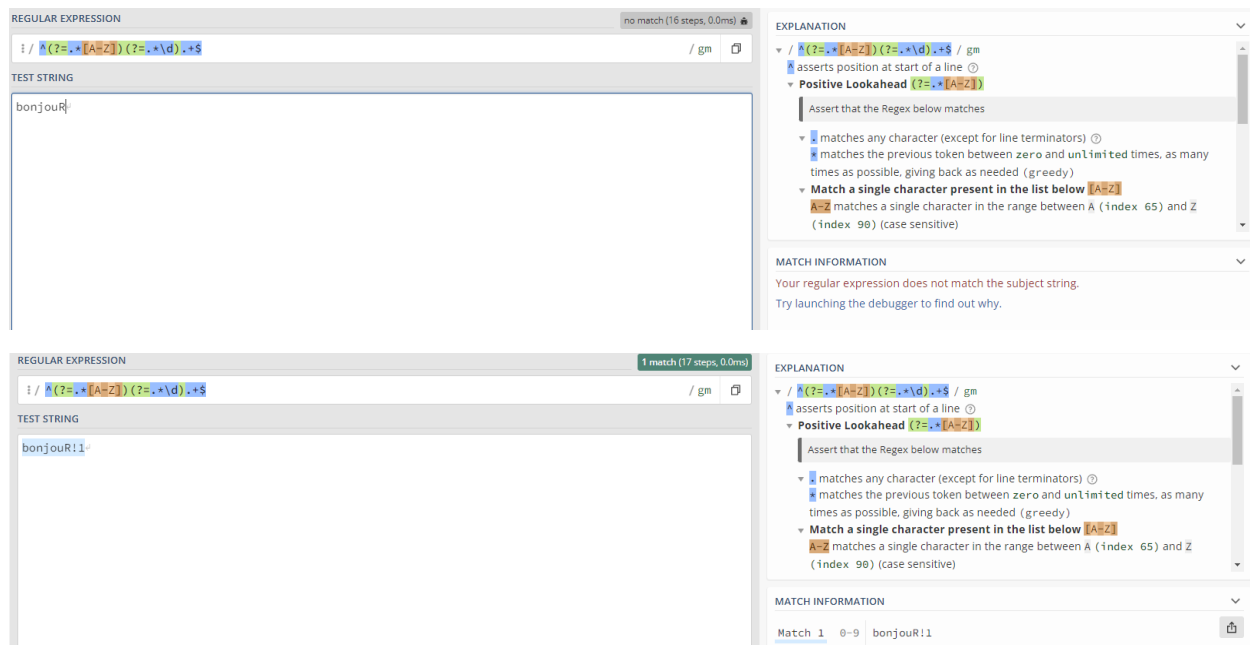
Nous avons décidé de rajouter une couche de sécurité au mot de passe de l'utilisateur. Ainsi, l'utilisateur qui crée un compte sera obligé d'avoir au moins une lettre majuscule et un nombre dans son mot de passe. En C#, la ligne de code qui nous assure que l'utilisateur respecte ces conditions ressemble à ceci « `@"^(?=.*[A-Z])(?=.*\d).+$"` ».

Voici le résultat de nos tests avec cette expression régulière :

The image displays two screenshots of a regular expression testing interface. Both screenshots show the same regular expression: `^(?=.*[A-Z])(?=.*\d).+$`. The first screenshot shows the test string "bonjour" and the result "no match (11 steps, 0.0ms)". The second screenshot shows the test string "bonjour1" and the result "no match (12 steps, 0.0ms)". The explanation pane on the right of each screenshot details the components of the regex:

- `^`: asserts position at start of a line.
- `(?=.*[A-Z])`: Positive Lookahead, asserts that the Regex below matches.
- `(?=.*\d)`: Positive Lookahead, asserts that the Regex below matches.
- `.`: matches any character (except for line terminators).
- `+`: matches the previous token between zero and unlimited times, as many times as possible, giving back as needed (greedy).
- `$`: Match a single character present in the list below [A-Z].

 The match information pane in both screenshots states: "Your regular expression does not match the subject string. Try launching the debugger to find out why."



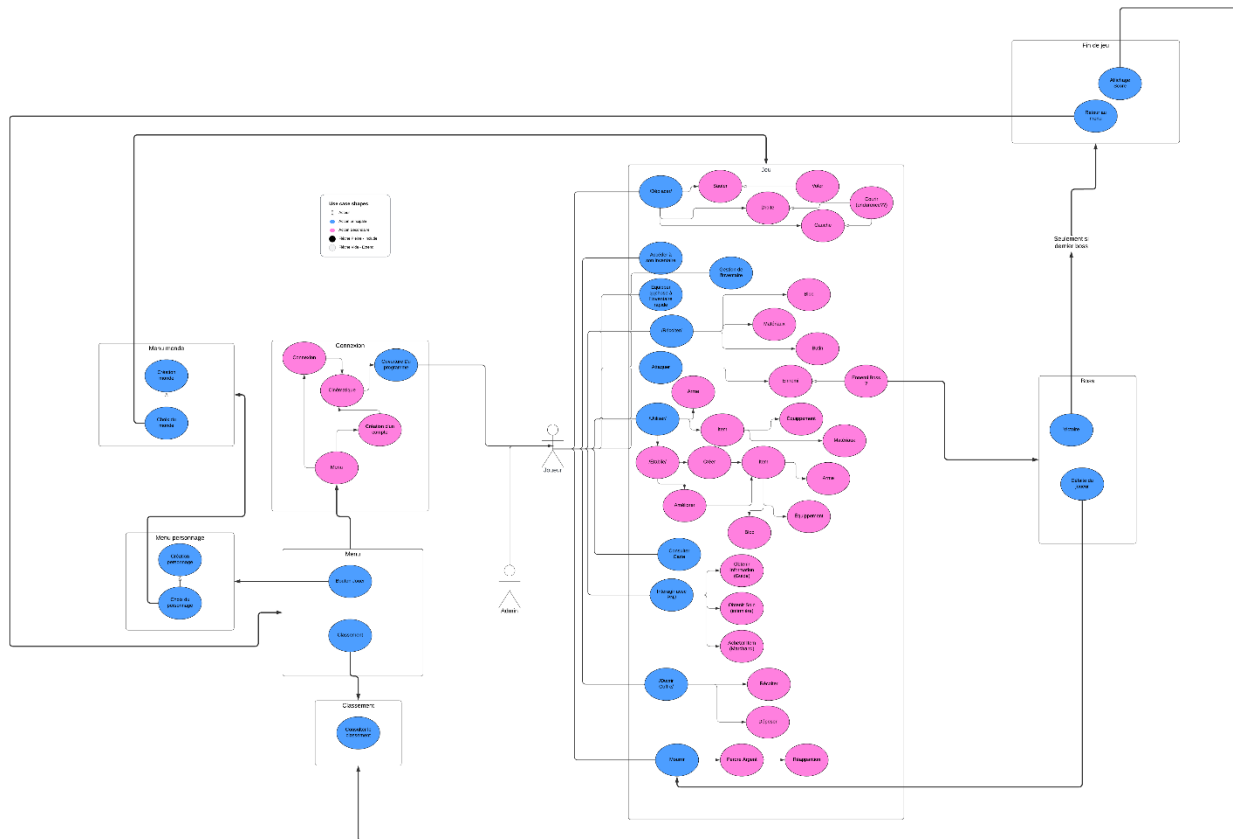
Algorithme

Dans notre projet, nous utiliserons un algorithme de génération de carte aléatoire réaliste afin de régler le problème de redondance dans la carte. De plus, cet algorithme nous permettra de ne pas qu'avoir seulement une carte, mais une multitude sans avoir à toutes les préfabriquer. Pour se faire, nous utiliserons deux algorithmes déjà connu de génération de bruit aléatoire pour nous faciliter un peu la tâche, soit le « Perlin Noise » ainsi que « Cellular Automata ». Ceux-ci nous permettront de générer des « Tilemaps » aléatoirement à travers la carte. Nous aurons des variables qui nous permettront directement dans Unity ou dans le code de régler les détails de la génération, soit la fluidité de génération (smoothness), la hauteur (height) et la largeur (width). À l'aide du concept de « Perlin Noise », nous allons nous assurer que les cartes des mondes soient générées aléatoirement et qu'il y ait une certaine cohérence dans les mondes générés. Ainsi, les reliefs et recoins des mondes générés seront fait d'une façon intelligente et avec un certain réalisme. Quant au « Cellular Automata », celui-ci sera utilisé pour la génération aléatoire des reliefs souterrains, c'est-à-dire les cavernes, et pourrait aussi être utilisé pour une génération aléatoire de veine de minerais. Finalement, cet algorithme sera généralisé afin de permettre son utilisation dans plusieurs autres projets de jeux.

Mathématique

Lorsque le joueur combat un Boss du jeu, nous allons avoir besoin de certaines notions de mathématique pour rendre le jeu plus intéressant et plus réaliste. Pour calculer les dommages que les Boss du jeu font au joueur, nous allons calculer la distance entre le joueur et le Boss avec qui le joueur combat à l'aide de vecteurs, qui seront calculé par la position du joueur moins celle du Boss. Ensuite, nous utiliserons le vecteur pour réduire la vie du joueur selon un calcul qui consiste en le vecteur multiplié par une constante, qui variera par boss.

Diagramme des cas d'usages :



[illegible]

Cellular automata : <https://www.youtube.com/watch?v=slTEz6555Ts&t=30s> (White Box Dev, 19 septembre 2020).