Projet synthèse

Document de conception

Réalisé par :

Matei Pelletier & Christophe Auclair

Technique de l'informatique

Cégep du Vieux Montréal

Présentation générale :

L’objectif de ce projet est de créer un jeu vidéo multijoueur de type 2D dungeon crawler fortement inspiré par les mythic dungeons dans le jeu légendaire « World of Warcraft » dans le contexte du projet synthèse de notre technique en informatique.

Présentation détaillée :

Le projet sera une application Windows utilisant l’engine Unity, codée en C#. Le thème du jeu est un dungeon crawler médiéval fantastique, avec un style d'art pixel. Bien qu'il s'agisse avant tout d'un dungeon crawler, le jeu comporte également plusieurs éléments de RPG, comme des classes de joueurs avec des arbres de talents et du loot qui améliore les capacités des personnages. La boucle de jeu principale consiste à créer des lobbies où d'autres joueurs peuvent se joindre afin de compléter un donjon ensemble. Au départ, les nouveaux joueurs ne pourront créer que des lobbies de donjon de niveau 0, mais ils pourront rejoindre des lobbies de niveau supérieur. Une fois le jeu lancé, un donjon généré aléatoirement (Il y aura des différences dans la génération selon la difficulté du donjon) sera créé pour le lobby. Un chronomètre s'enclenchera alors pour que les joueurs le battent avant la fin du donjon. Mourir enlève 5 secondes au chronomètre. Si un groupe parvient à terminer le donjon à temps, il recevra une clé pour un donjon de niveau supérieur qu'il pourra utiliser pour commencer un nouveau lobby à ce niveau de difficulté. Les difficultés supérieures consistent en une augmentation de la santé et des dégâts des ennemis, ainsi qu'en des mécanismes plus difficiles ou plus nombreux à affronter. À la fin de chaque parcours de donjon, les joueurs recevront des récompenses sous forme d'équipement et de points d'expérience. Plus le niveau de difficulté du donjon terminé est élevé, plus l'équipement sera performant en termes de statistiques. Ainsi, les joueurs devront terminer les donjons de niveau inférieur afin d'obtenir un équipement suffisamment bon pour pouvoir terminer les donjons de niveau supérieur.

Classes de personnages disponibles:

- Berzerker : melee DPS
- Invoker : ranged healer
- Mage : ranged DPS
- Necromancer : ranged DPS qui invoque des minions qui se battent pour eux
- Warden : tank

Talents :

Chaque classe disposera d'un système de talents pour lesquels des points de talent seront dépensés lors de la montée en niveau. Ces talents leur permettent d'acquérir de nouvelles habiletés ou d'améliorer celles qu'ils possèdent déjà.

Ennemies :

Les joueurs devront affronter différents types d'ennemis. Les quatre types généraux d'ennemis, à l'exception des boss, sont les suivants :

- Zugger : un attaquant de mêlée de base qui court vers les joueurs pour les frapper.

-Sorcier : un attaquant à distance qui lance des sorts aux joueurs et dispose d'une attaque spéciale qui inflige de lourds dégâts à ceux qui l'entourent.

-Le meneur de jeu : un attaquant à distance qui lance des sorts sur les joueurs et invoque des serviteurs pour les attaquer.

-Lézard : un attaquant de mêlée qui répand du poison sur les joueurs.

-Gobelin : un attaquant de mêlée qui pose des explosifs et qui explose lorsqu'il meurt.

Systèmes à mettre en place :

- Système de lobby permettant aux joueurs de trouver d'autres groupes et de créer le leur.

- Algorithme de génération de donjons

- IA pour différents types d'ennemis

- Mouvement, contrôles, utilisation des capacités (capacités de visée, ciblage...) pour le joueur.

- Système de gestion de l'inventaire et de l'équipement

- Système d'échelonnement des ennemis et du butin en fonction de la difficulté du donjon

- Infrastructure de serveur et de netcode pour garder les joueurs synchronisés dans le jeu.

- Conception des classes : qui comprend les arbres de talents et les capacités

- Conception des ennemis : capacités, quantité, dégâts, vie...

- Système de clés permettant aux joueurs de progresser en difficulté

- Système de gestion des comptes joueurs (login, personnages, progression)

- Base de données pour stocker les informations sur les joueurs

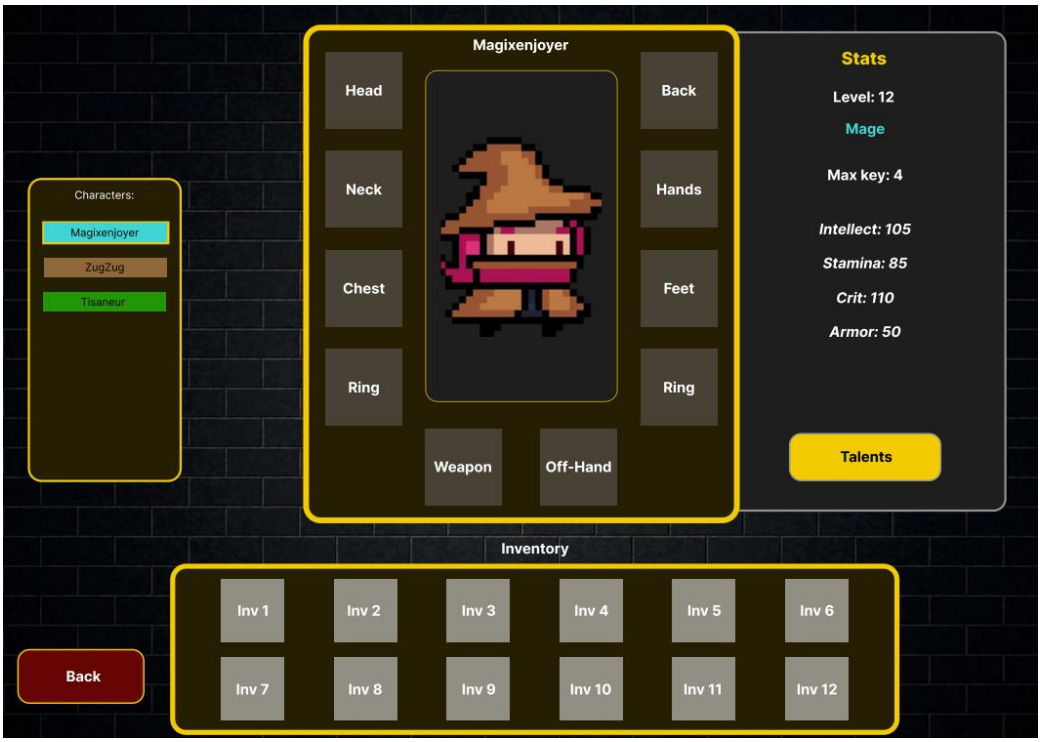
- Options supplémentaires : minimap/map, conception sonore, fidélité graphique chat.

Interface Utilisateur :

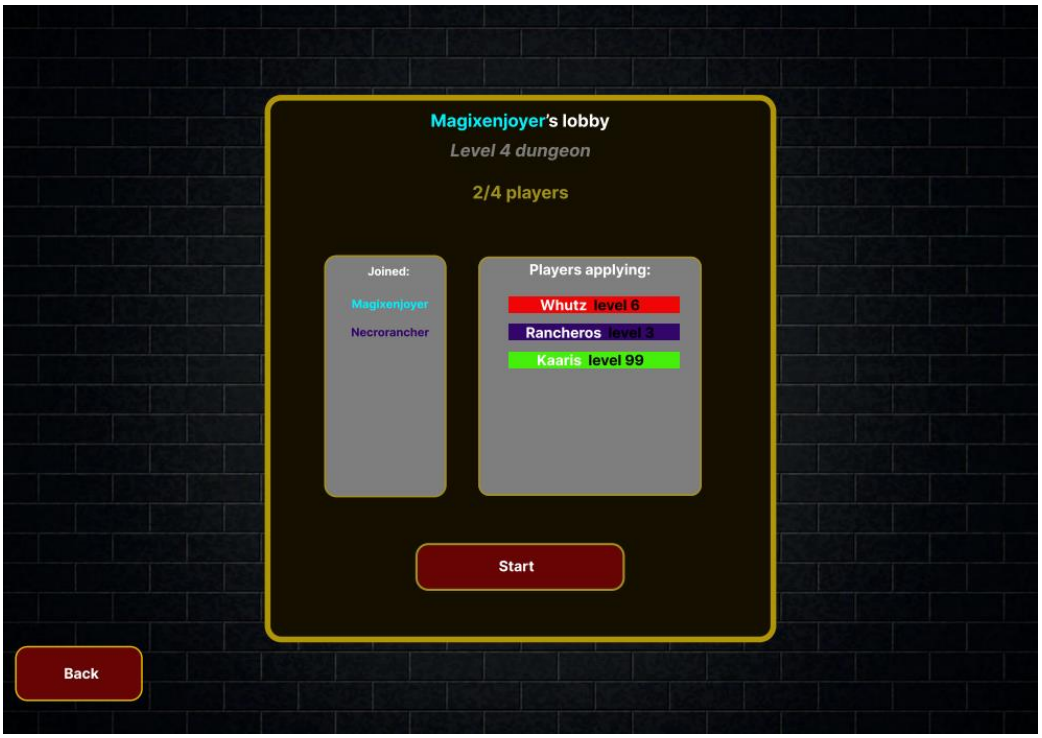
Notre interface utilisateur se composera de plusieurs scènes dans le jeu, telles que la scène du menu, la scène du lobby, la scène de sélection des personnages, la scène du jeu et la scène de fin de jeu.

Maquettes :

Sélection de personnage :

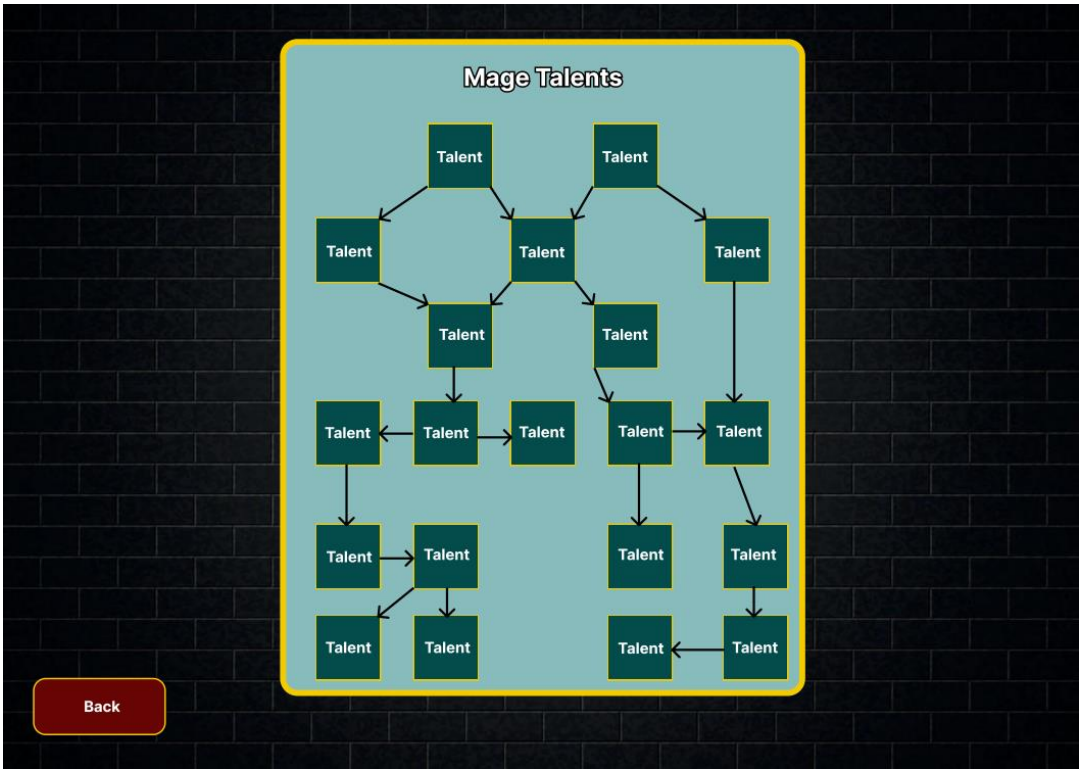


Lobby :



Menu :

Talents :



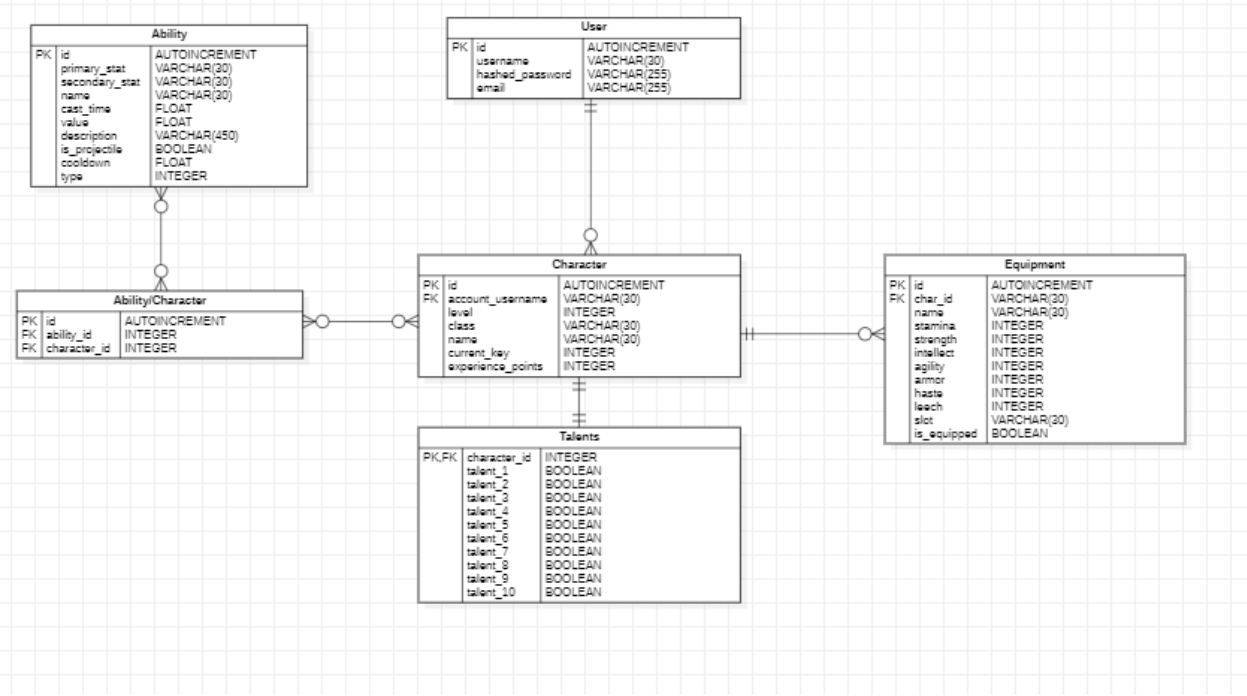
Jeu :

<u>Mécanismes de saisie d'information:</u>

- Champs de texte pour saisir les noms des personnages, les noms des lobbies, le chat.
- Entrées au clavier et à la souris pour contrôler le personnage, utiliser des sorts, ouvrir et fermer des éléments de l'interface utilisateur.
- Timer formaté
- Sprites pour les personnages, les ennemis, les accessoires, les tuiles de la carte.

Données externes :

Nous utiliserons une base de données MySql pour stocker les données relatives aux comptes d'utilisateurs et à leurs personnages.



Patrons de conception:

Décorateur :

Nous utiliserons un décorateur pour construire nos capacités de manière modulaire et personnalisable, ce qui nous permettra d'implémenter facilement plus de capacités pour chaque classe de personnage.

Médiateur :

Nous utiliserons un médiateur pour contrôler toutes les fonctions multijoueur qui nécessitent la transmission de données entre les joueurs et le serveur. Par exemple, lors de l'utilisation de capacités, puisque celles-ci doivent exister pour tous les clients, le client qui souhaite utiliser un sort devra passer par une classe médiatrice pour demander que le sort soit créé. De la même manière, les autres clients devront demander les données au serveur à travers cette classe afin de pouvoir voir le sort de leur côté.

Singleton :

Un grand nombre de nos systèmes dans le jeu nécessiteront l'utilisation d'un Singleton. Il s'agit notamment de tous les gestionnaires qui gèrent les informations de compte du joueur, le gestionnaire de base de données, le gestionnaire de lobby et le gestionnaire de réseau. L'utilisation d'un Singleton est cruciale dans Unity pour la persistance entre les scènes.

Expressions régulières:

Nous utiliserons les expressions régulières dans notre projet afin de valider les noms de compte et les mots de passe des utilisateurs, ainsi que dans le gestionnaire de base de données afin d'effectuer des requêtes.

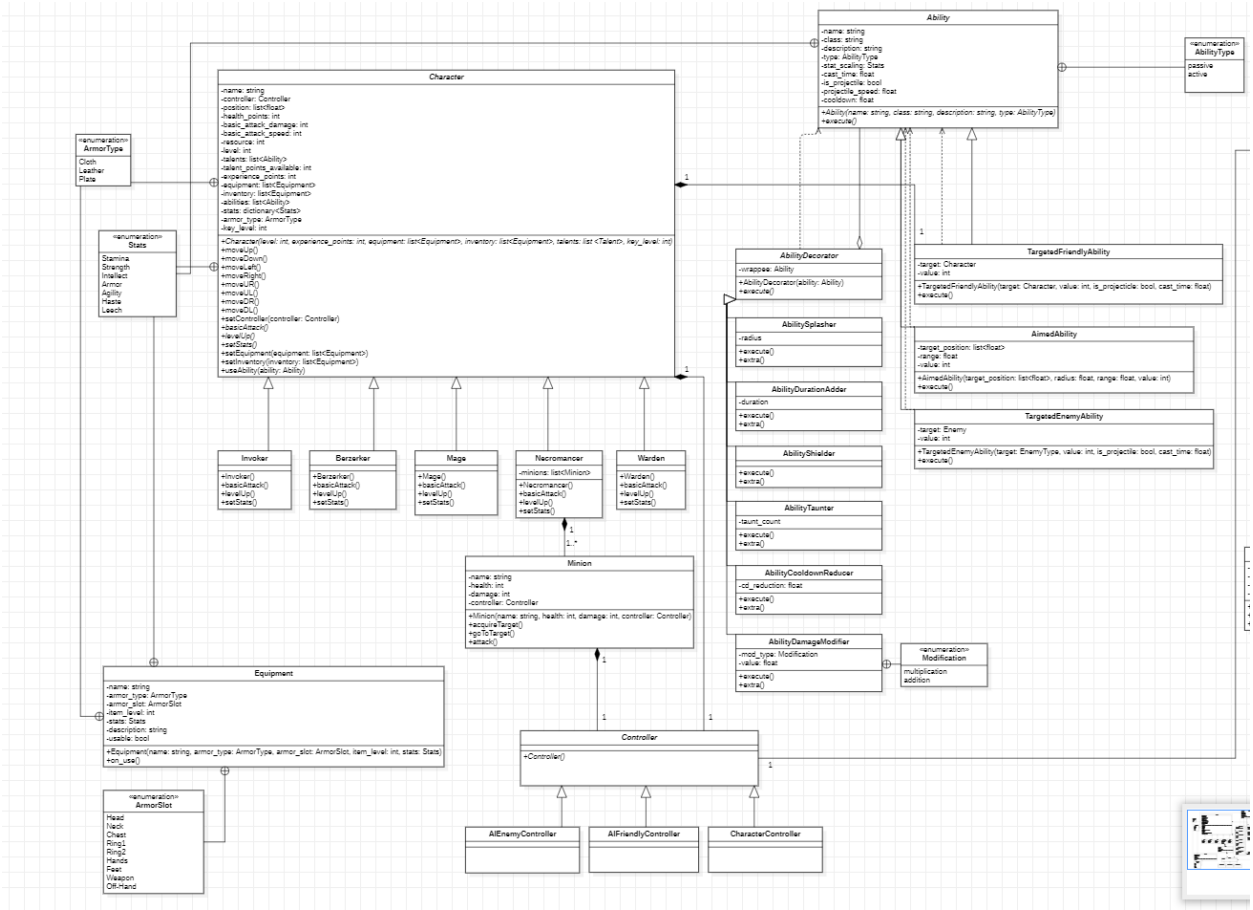
Algorithmes:

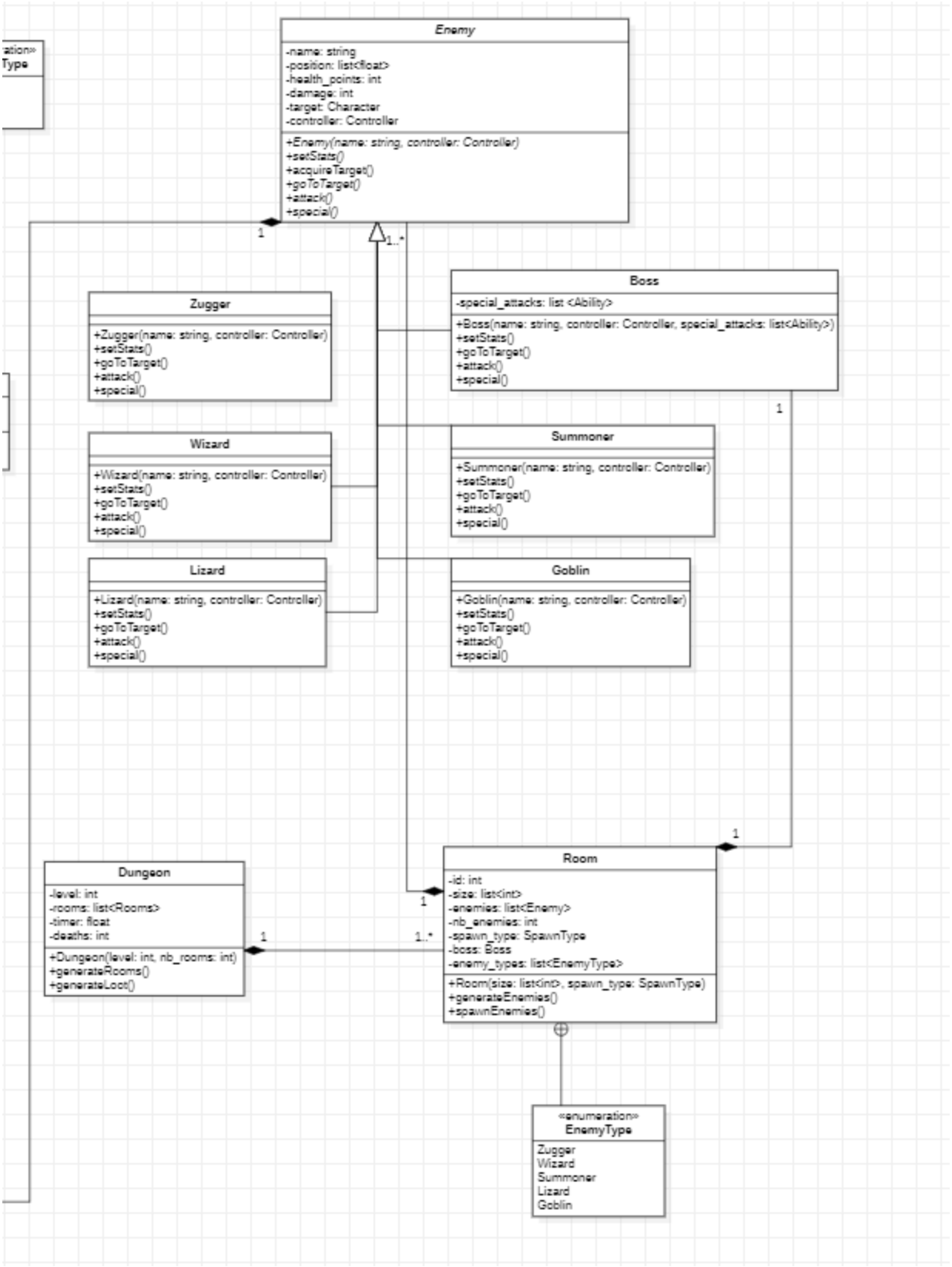
Nous utiliserons l'algorithme de binary space partitioning et l'algorithme de simple random walk afin de générer nos donjons de manière aléatoire.

Mathématiques:

Nous utiliserons les mathématiques dans notre algorithme de partitionnement de l'espace binaire lors de la création de donjons.

Diagramme de classes :


Ability
-name: string
-class: string
-description: string
-type: AbilityType
-stat_scaling: Stats
-cast_time: float
-is_projectile: bool
-projectile_speed: float
-cooldown: float
+Ability(name: string, class: string, description: string, type: AbilityType)
+execute()
«enumration»
AbilityType
passive
active
Character
-name: string
-controller: Controller
-position: listfloat>
-health_points: int
-basic_attack_damage: int
-basic_attack_speed: int
-resource: int
-level: int
-talents: list<Ability>
-talent_points_available: int
-experience_points: int
-equipment: list<Equipment>
-inventory: list<Equipment>
-abilities: list<Ability>
-stats: dictionary<Stats>
-armor_type: ArmorType
-key_level: int
+Character(level: int, experience_points: int, equipment: listEquipment>, inventory: listEquipment>, talents: list <Talent>, key_level: int)
+moveUp()
+moveDown()
+moveLeft()
+moveRight()
+moveUR()
+moveUL()
+moveDR()
+moveDL()
+setController(controller: Controller)
+basicAttack()
+levelUp()
+setStats()
+setEquipment(equipment: listEquipment>)
+setInventory(inventory: listEquipment>)
+useAbility(ability: Ability)
«enumration»
ArmorType
Cloth
Leather
Plate
«enumration»
Stats
Stamina
Strength
Intellect
Armor
Agility
Haste
Leach
AbilityDecorator
-wrappee: Ability
+AbilityDecorator(ability: Ability)
+execute()
TargetedFriendlyAbility
-target: Character
-value: int
+TargetedFriendlyAbility(target: Character, value: int, is_projectile: bool, cast_time: float)
+execute()
AbilitySplasher
-radius
+execute()
+extra()
AimedAbility
-target_position: listfloat>
-range: float
-value: int
+AimedAbility(target_position: listfloat>, radius: float, range: float, value: int)
+execute()
AbilityDurationAdder
-duration
+execute()
+extra()
TargetedEnemyAbility
-target: Enemy
-value: int
+TargetedEnemyAbility(target: EnemyType, value: int, is_projectile: bool, cast_time: float)
+execute()
AbilityShielder
+execute()
+extra()
AbilityTaunter
-taunt_count
+execute()
+extra()
AbilityCooldownReducer
-cd_reduction: float
+execute()
+extra()
AbilityDamageModifier
-mod_type: Modification
-value: float
+execute()
+extra()
«enumration»
Modification
multiplication
addition
Invoker
+Invoker()
+basicAttack()
+levelUp()
+setStats()
Berzerker
+Berzerker()
+basicAttack()
+levelUp()
+setStats()
Mage
+Mage()
+basicAttack()
+levelUp()
+setStats()
Necromancer
-minions: list<Minion>
+Necromancer()
+basicAttack()
+levelUp()
+setStats()
Warden
+Warden()
+basicAttack()
+levelUp()
+setStats()
Minion
-name: string
-health: int
-damage: int
-controller: Controller
+Minion(name: string, health: int, damage: int, controller: Controller)
+acquireTarget()
+goToTarget()
+attack()
Equipment
-name: string
-armor_type: ArmorType
-armor_slot: ArmorSlot
-item_level: int
-stats: Stats
-description: string
-usable: bool
+Equipment(name: string, armor_type: ArmorType, armor_slot: ArmorSlot, item_level: int, stats: Stats)
+on_use()
«enumration»
ArmorSlot
Head
Neck
Chest
Ring1
Ring2
Hands
Feet
Weapon
Off-Hand
Controller
+Controller()
AIEnemyController
AIFriendlyController
CharacterController


Enemy
-name: string
-position: list<float>
-health_points: int
-damage: int
-target: Character
-controller: Controller
+Enemy(name: string, controller: Controller)
+setStats()
+acquireTarget()
+goToTarget()
+attack()
+special()
1
1..*
Zugger
+Zugger(name: string, controller: Controller)
+setStats()
+goToTarget()
+attack()
+special()
Boss
-special_attacks: list <Ability>
+Boss(name: string, controller: Controller, special_attacks: list<Ability>)
+setStats()
+goToTarget()
+attack()
+special()
1
Wizard
+Wizard(name: string, controller: Controller)
+setStats()
+goToTarget()
+attack()
+special()
Summoner
+Summoner(name: string, controller: Controller)
+setStats()
+goToTarget()
+attack()
+special()
Lizard
+Lizard(name: string, controller: Controller)
+setStats()
+goToTarget()
+attack()
+special()
Goblin
+Goblin(name: string, controller: Controller)
+setStats()
+goToTarget()
+attack()
+special()
1
Dungeon
-level: int
-rooms: list<Rooms>
-timer: float
-deaths: int
+Dungeon(level: int, nb_rooms: int)
+generateRooms()
+generateLoot()
1
1..*
Room
-id: int
-size: list<int>
-enemies: list<Enemy>
-nb_enemies: int
-spawn_type: SpawnType
-boss: Boss
-enemy_types: list<EnemyType>
+Room(size: list<int>, spawn_type: SpawnType)
+generateEnemies()
+spawnEnemies()
1
«enumeration»
EnemyType
Zugger
Wizard
Summoner
Lizard
Goblin