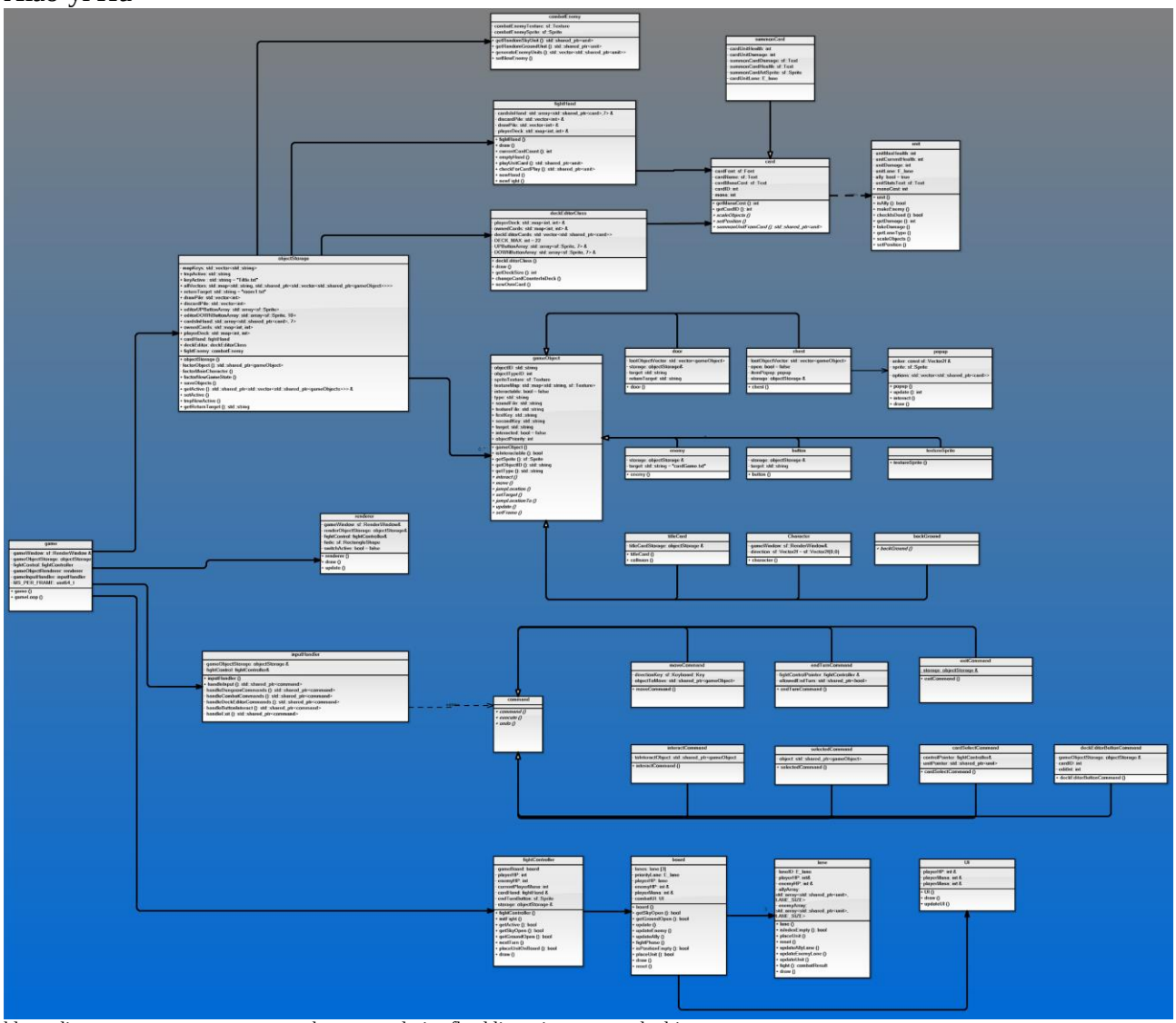
Team Cnubba – 2VC technisch verslag

Gameproject blok 2, jaar 2019-2020

Matthies Brouwer
Pieter Ruijs
Otto de visser
Watze van Steenbergen
Xiao yi Hu



klasse diagram van ons game, voor een hogere resolutie afbeelding, zie onze google drive.

Inleiding

Ons concept voor het spel was een spel waar je een wereld navigeert in een dungeoncrawler, waar men door verschillende kamers heen loopt met chests die loot kaarten bevatten en vijanden tegenkomt.

De combat met de vijanden vind plaats in een card game achtig systeem geinspireerd door onder andere slay the spire en hearthstone, waar men kaarten kan spelen die units van 2 types, flying of ground op het veld in de juiste lane zetten, die richting de tegenstander beweegt en zijn units in de zelfde lane bevecht, in het geval dat de unit de tegenstander bereikt heeft, zal die de tegenstander aanvallen en zichzelf van het veld verwijderen. Wanneer de levens van of de speler of de vijand lager dan 1 wordt, is diegen verslagen.

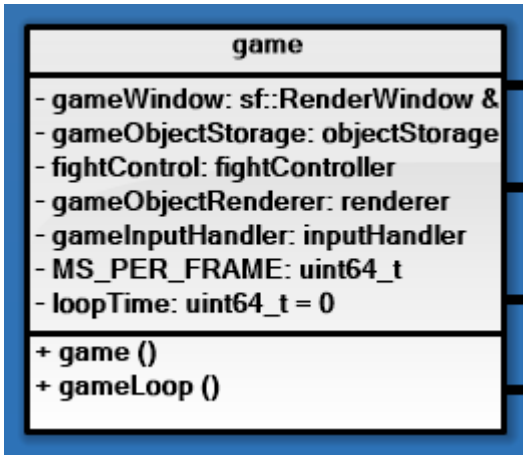
Om deze twee delen goed te kunnen accomoderen moesten we kiezen voor een design waar de twee veschillende delen apart afgehandeld worden door de grote verschil in functionaliteit, en vereiste input. Dit wordt gefaciliteert door de twee delen van de spellen grotendeels apart te houden, met een input listener en command pattern, die ervoor zorgt dat de input van de speler op de juiste wijze wordt behandelt gebaseerd op in welke staat het spel zit.

Tijdens dit project hebben we een sterke focus gelegt op het bouwen van een robuust en sterke basis wat makkelijker maakt om het spel later uit te breiden. Hiervoor kozen wij om gebruik te maken van de factory pattern waar alle drawable gameObjects gemaakt worden. Hierdoor kunnen wij makkelijk levels bouwen in tekst bestanden en die uit lezen om alles te laden.
Verder is er een objectStorage klasse die alle gameObjecten bijhoud en opslaat tijdens het dungeoncrawlen. Overigens slaat deze ook de staat van alle kamers op zelfs wanneer de speler de kamer verlaat, dit zorgt ervoor dat de kamer in dezelfde staat is wanneer de speler weer terugloopt.
Voor het updaten en drawen van alle objecten gebruiken we de klasse renderer. Dit zorgt voor een mooie splitsing tussen model (gedrag van objecten op zichzelf), view (rendering), controller(input) zoals in het MVC model.

Gameloop

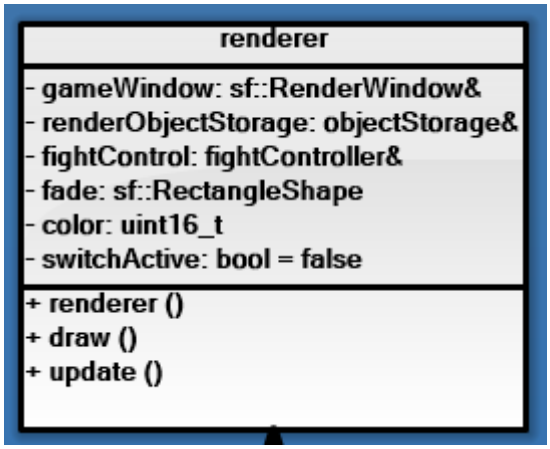
De entry point van ons progamma is de main functie binnen de main.cpp, hier wordt een sf::RenderWindow gemmakt en een game object aangemaakt die een reference bevat, waarop de gameLoop methode aangeroepen wordt.

De game klasse bevat een reference naar de window gemaakt in de main en de alle grote controller klasses die in ons spel zitten, de objectStorage, renderer, fightController en inputHandler.
In de gameLoop wordt er gekeken hoeveeltijd er gepaseert is om de framrate constant te houden, de inputhandler aangeroepen om commands te krijgen gebaseerd op input, de executie van deze commands.
Muziek en booleans worden daarna aangestuurd gebaseerd op de state van de game, zoals in combat, in dungeon, in menu, de staat van het spel wordt bepaalt door de eerste letter van de txt file waaruit de op dit moment actieve locatie van het spel wordt geladen, alle dungeon kamers beginnen met ‘r’, gevechtem met ‘b’, en de menu’s met ‘m’.
Er worden ook constant oproepen gedaan naar de fightController om ervoor te zorgen dat er een goede pauze zit tussen alle stappen die gedaan worden in de nextTurn methode.
Daarna wordt de window gecleared en overheen getekent met de renderer, en als laatste wordt er gekeken r een event is waar de gebruiker de window afsluit, is dat het geval, dan wordt de window gesloten samen met het spel.



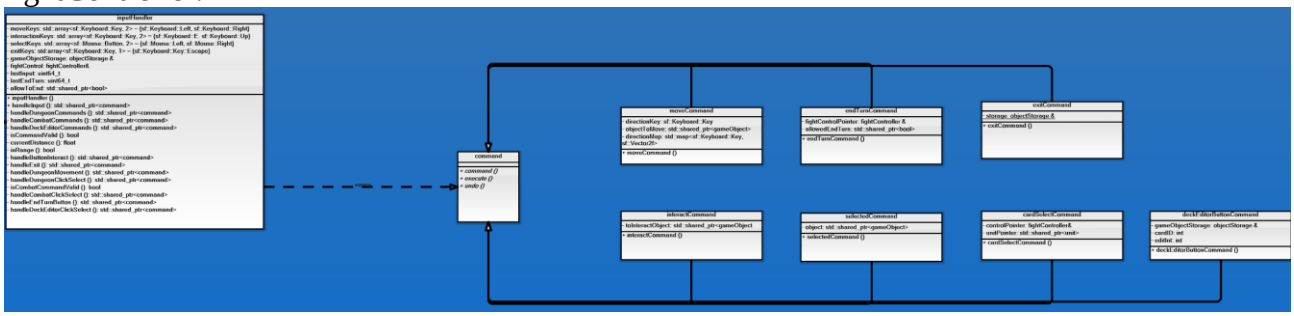
renderer

de renderer bezit references naar de window, objectStorage en de fightController, hij roept de updates van alle objecten aan met een eigen behaviour in de game, en kiest gebaseerd op welke staat de game in zit welke objecten er op het scherm getekent moeten worden.



inputHandler

De inputHandler luistert naar de inputs van de keyboard en muis en maakt gebaseerd op de staat van het spel de juiste commands aan en returnt die naar de gameloop waar ze uitgevoerd worden. De reden dat we gekozen hebben voor deze manier van werken, is omdat het het zeer makkelijk maakt voor ons om nieuwe input functionaliteiten toe te voegen, en ze aan te passen. Voor het correct aanmaken van de commands heeft de inputHandler references naar objectStorage en fightController.

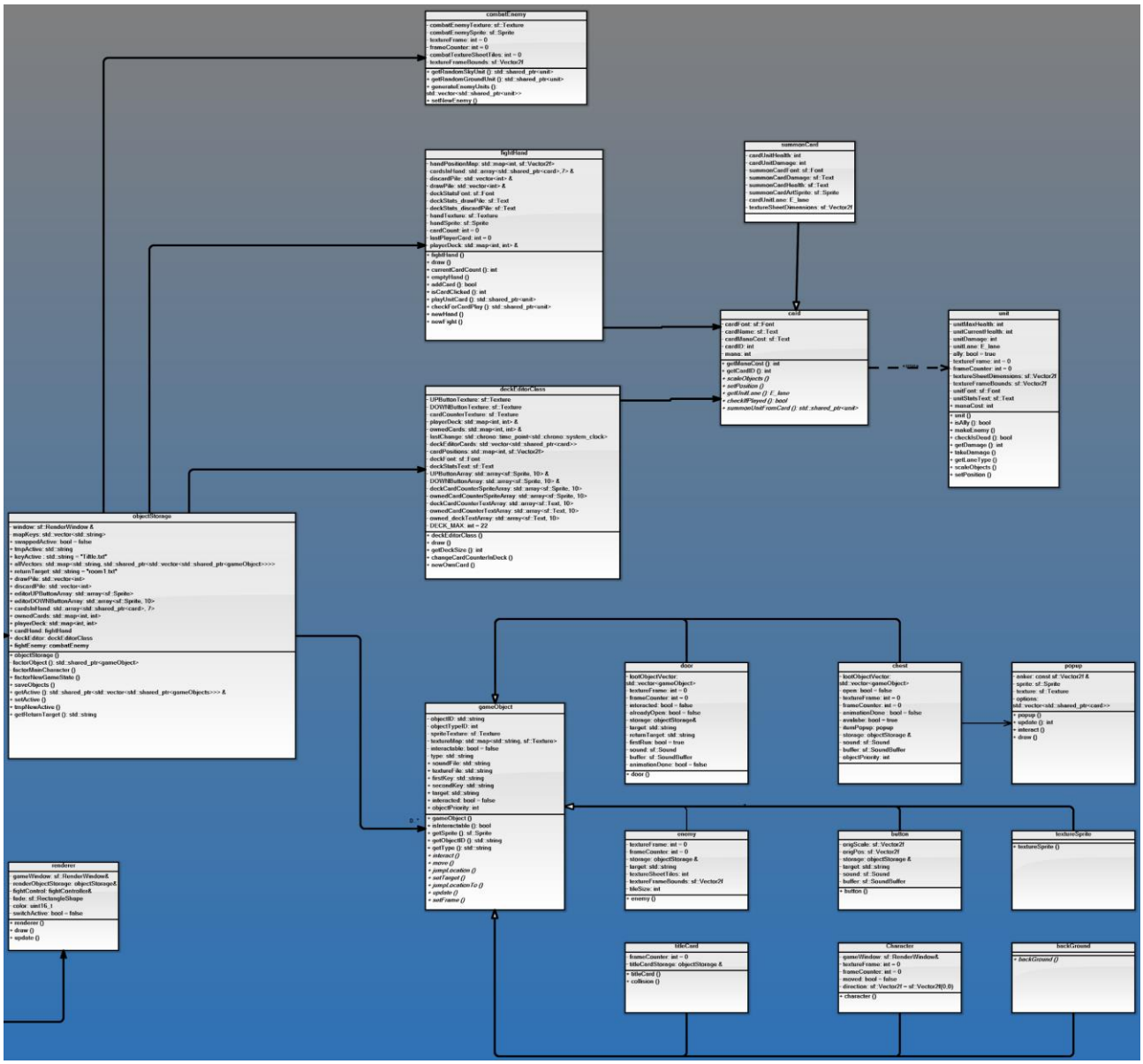


objectStorage

de ObjectStorage klasse is waar alle gameObjecten buiten die in board zitten worden opgeslagen. Deze klasse kan bepalen door te kijken naar de staat van het spel en de kamer waarin de speler zich bevindt , welke gameObjecten actief moeten zijn. hierdoor kunnen wij op een centrale plek alle objecten opvragen en over ze heen itereren, zoals de renderer dit doet moet de update en draw functies.

Verder is deze klasse van groot belang tijdens het opstarten van de game, sinds het ook als de factory van ons programma fungeert. Dit betekent dat hij alle txt files leest en daarop gebaseerd gameObjects produceert, dit zorgt ervoor dat wij makkelijk nieuwe kamers kunnen opslaan en laden. Het was ook de bedoeling om via deze klasse dan ook veranderingen in de gamestate weer op te laten slaan in de eerder genoemde txt files, zodat wij een save state systeem konden bouwen, helaas viel dat buiten de scope van het project.

Wel kan de klasse de staten van alle kamers opslaan binnen een speelsessie, hierdoor kun je als speler zien of je al eerder in een kamer bent geweest omdat de deuren en schatkisten open blijven.



Card Deck

Een van de main mechanics van onze game is het kaartensysteem. In de basis houd het in dat de speler door zijn experience heen een verzameling aan kaarten kan verzamelen die hij kan gebruiken tijdens conflicten met tegenstanders, om betere kaarten te verzamelen.

Kaarten mechanics

Kaarten kunnen worden gebruikt tijdens Combat om units te spawnen. Wanneer een kaart gespeeld wordt zal er op zijn corresponderende lane een unit met alle eigenschappen van de kaart verschijnen. Deze eigenschappen staan allemaal opgeslagen in een kaarten library tekstbestand, dmv de unieke kaartID's. Aan dit kaart ID staan een aantal specifieke eigenschappen gebonden, die in de kaart instantie opgeslagen worden. Wanneer de kaart op het scherm getekend of verplaatst wordt zullen alle losse sf::Text en sf::Sprite variabelen zich aanpassen zodat het altijd op 1 soliede object lijkt. Deze eigenschappen bestaan uit:

ManaCost:

Per beurt krijgt de speler een bepaald aantal mana om kaarten mee te spelen. De manaCost van de kaarten staat zichtbaar in de blauwe indicator van de kaart.

CardName:

Dit is de naam van de kaart. Op deze manier krijgt de kaart iets meer persoonlijkheid

UnitPreview:

Het plaatje in het midden van de kaart geeft het uiterlijk van de unit aan die de kaart zal summonen wanneer deze gespeeld wordt.

UnitDamage:

Dit getal geeft weer hoeveel damage de gespeelde unit zal dealen tegen de tegenstander en andere units.

UnitLaneType:

Tijdens combat kunnen units op 2 verschillende lanes gespeeld worden. De kaarten met een schoen op deze plek gaan naar de ground lane, en de kaarten met een vleugel gaan naar de skylane.

UnitHealth:

Dit getal staat voor hoeveel damage de unit van de kaart kan nemen, voordat deze sterft

Kaartenhand trekken:

De speler heeft tijdens combat de keuze uit verschillende speelbare kaarten. Hiervoor krijgt hij/zij elke beurt een nieuwe hand van 7 kaarten die uit de drawpile worden getrokken. Wanneer de speler zijn kaarten heeft gespeeld en de beurt afgeeft zullen alle overige kaarten in zijn/haar hand naar de discardpile gaan, zodat er aan het begin van de volgende beurt weer 7 nieuwe getrokken kunnen worden. Deze discardpile shuffled zichzelf terug in de drawpile wanneer de drawpile niet genoeg kaarten heeft om een volledige hand te pakken. Op deze manier shuffled de speler constant door alle opties heen, en moet er beter over de opties worden nagedacht.

Kaarten verzameling

Als speler kan je een grote verzameling aan kaarten hebben, waaruit je een deck kan samenstellen die je later in de dungeon kan gebruiken. Hiervoor is een deckviewer aanwezig waarin de speler via pijltoetsen simpel het aantal kaarten in zijn deck kan bijhouden en aanpassen. Deze kaarten worden in de objectStorage klasse van de game opgeslagen in een map met als key en value een Integer, wat staat voor de specifieke kaart ID en het aantal.. Dit geeft een simpele nog geheugen lichte manier om de kaarten op te slaan en simpel aan te passen.

KaartFactory

Om de functionaliteit van het opslaan in integers uit te breiden naar andere klassen bezit onze game over een kaartFactory. Dit is een simpele factory functie die door middel van de opgeslagen kaartID Integer een kaartpointer instantie kan factoreren en returnen. Deze instanties hebben direct een uitgebreide verzameling aan gebruiksvriendelijke functies zoals scalen, verplaatsen en interactie tests, waardoor de kaarten simpel geïntegreerd kunnen worden in verschillende delen van het spel. Hiernaast biedt deze factory ook een simpele wijze voor het toevoegen van kaarten aan het spel, aangezien hier een tekstbestand library aan gelinkt zit waar een developer simpelweg een nieuwe lijn tekst toe hoeft te voegen.

Nieuwe kaarten bemachtigen

Door de dungeon heen kan de speler nieuwe kaarten bemachtigen om aan zijn verzameling toe te voegen. Dit kan op 2 verschillende manieren

Chests:

Door de dungeon heen kan de speler chests tegenkomen. Wanneer deze gebruikt worden krijgt de speler 3 kaarten te zien, waar hij er 1 van aan zijn deck toe kan voegen.

CombatReward:

Wanneer de speler voor het eerst zijn kaarten verzameling bekijkt zal hij direct zien dat 3 kaarten uit een vraagteken bestaan. Dit is omdat deze voor de speler nog niet mogelijk zijn. Wanneer de speler een enemy verslaat in combat krijgt hij de keuze uit 1 van deze drie kaarten. Hierna kan hij deze vanuit zijn deck editor gemakkelijk

toevoegen aan zijn deck. Deze kaarten zijn over het algemeen sterker, maar daarom ook direct moeilijker om te bemachtigen

Combat

De combat wordt afgehandeld door de fightController in samenwerking met de inputHandler en de ObjectStorage klasse. De fightController roept de board klasse aan die tijdens combat alles wat met units te maken heeft beheert, en werkt samen met de het deck systeem en de renderer.

De focus van de fightController is dat het de samenwerking tussen de board klasse en het rest van ons programma goed faciliteert en dat alle fases van het combat systeem goed afgelopen wordt. Dit betekent dat hij aangeroepen wordt in de commands waar het checkt op welke lanes een unit gespeeld kan worden, en hoeveel mana de speler heeft, deze data wordt gegeven aan de deck klasse die dan besluit of de geklikte kaart gebruikt mag worden om uit op het veld te zetten.
In het geval dat alle eisen voldaan worden, returnt de deck klasse een pointer naar de command, die deze doorgeeft aan de fightController, fightController geeft deze dan door aan de board klasse.

De fases in een onze gameplay-loop binnen de spel bevat het volgende:

- friendly unit update en reset mana phase:
  Alle vriendelijke units gaan een positie naar voren als de volgende positie vrij is, als ze op de uiterste positie aan de andere kant bereikt hebben vallen ze de vijand aan voordat ze zichzelf van het veld verwijderen.

- enemy summon phase:
  De vijand kijkt op welke een unit gespeeld kan worden, en zet een willekeurig unit neer.

- enemy unit update phase:
  Alle units van de vijand kijken of ze een stap naar de speler toe kunnen nemen, in het geval van wel, nemen ze die stap.

- fight phase:
  Units die tegenover een vijandig unit zitten initiëren een gevecht, units die sterven worden van het veld verwijdert.

- checks and draw phase:
  Er wordt gecheckt of de speler of de vijand verslagen zijn, in het geval dat de speler wint, wordt een rewardroom geladen waar de speler verder kan dungeon crawlen, in het geval dat de speler verloren heeft komt hij terug in de kamer waar het gevecht geïnitieerd werd. Als het gevecht niet over is worden nieuwe kaarten gepakt.

buiten deze cyclus wordt gestart nadat de speler op de finish knop drukt, waar tussen elke fase een periode gewacht wordt om ervoor te zorgen dat de speler kan zien wat er allemaal tussen elke fase is gebeurd.

Verder heeft fightController de functionaliteit om een gevecht te resetten om ervoor te zorgen dat bord leeg is en de speler en vijand de juiste hp hebben aan begin, en een draw functie, die de draw()

van de board klasse aanroept om ervoor te zorgen dat alle units op het veld op de juiste plek gedrawt wordt gebasser op hun lane, en hun positie binnen de lane.

De acties op het veld worden behandelt door de board object dat de fightController bevat. De board is een klasse die een array van lane objecten bevat, de keuze hiervoor stemt uit het feit dat er in de design van het spel meerdere lanes zouden zijn waar verschillende type units geplaats zouden kunnen worden, die grotendeels hetzelfde gedrag zouden tonen, het was dus simpeler om de lanes als een aparte klasse te implementeren om de code binnen board simpeler en compact te houden.

Board bevat dus een array met 3 lane objecten, deze lanes worden geidentificeerd op basis van een E_lane enum, met de mogelijke waarden E_lane::skyLane, E_lane::groundLane, en E_lane::trapLane. Board bevat ook een E_lane waarde genaamd priorityLane, die bepaalt welke lane eerst geupdate wordt, een UI member variable die de health en mana van de speler in een bar toont, en references naar de playerHP, enemyHP, playerMana en enemyMana variabelen van de fightController.

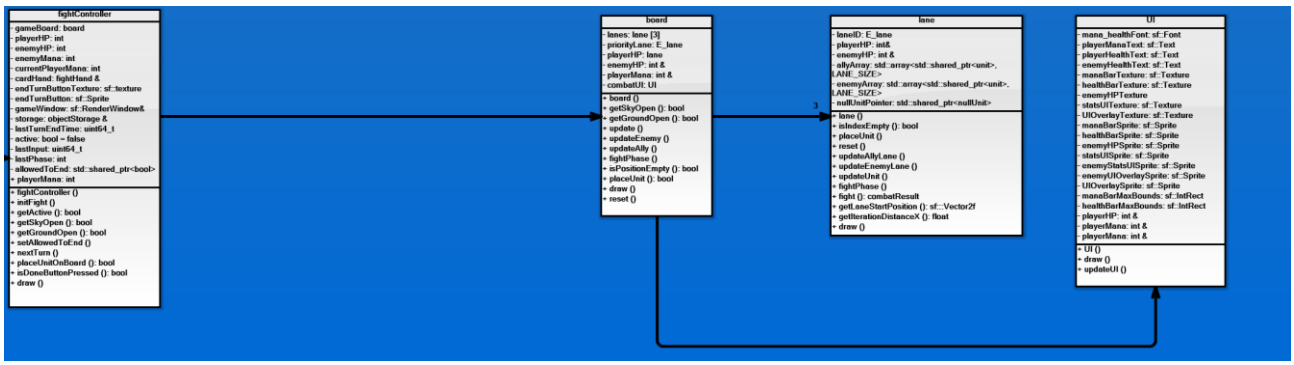
De functionaliteit van de board klasse bestaat uit het checken of verschillende locaties op de lanes vrij staan om units neer te zetten, lanes updaten, vriendelijke units op een lane updaten, of alleen vijandige units up te daten. Ui updaten, units op de juiste lane zetten en draw die de draw van alle lanes aanroept.

De lanes zelf zijn van de klasse lane die een E_lane laneID object bevatten om zich te identificeren, references naar de playerHP en enemyHP, en 2 std::array<std::shared_ptr<unit>, LANE_SIZE> objecten.
We hebben gekozen voor het gebruik van een deze data type, omdat het makkelijker was om over te itereren en duidelijker was dan een default c-style array, ook hadden we alle logica van de lanes geprogrammeerd met de LANE_SIZE macro zodat we op een plek hoeveelheid aan units per lane konden aanpassen, en dat de rest van de code zonder probleem gebruikt konden worden.
De reden waarom er twee van deze objecten zijn, is omdat we graag de vriendelijke wouden scheiden van enemy units, omdat we aan het begin tijdens de update van een unit een gevecht uit voerden, en we mogelijk zouden itereren met units die al vermoord waren in een voorgaande iteratie.

De functionaliteiten van de lane klassen bevat, het checken of bepaalde posities vrij zijn, het updaten van vriendelijke units, het updaten van vijandige units, units laten vechten met elkaar, units plaatsen op de positie binnen de juiste array, het tekenen van de units. En het resetten van de arrays.

In de updates van de units wordt dus eerst gecheckt of de eerstvolgende positie waar de unit heen moet lopen vrij is binnen beiden de allyArray en enemyArray (of de waarde op die index een nullptr is), in het geval dat het waar is wordt de nullptr in de array waar de unit in bevind vervangen met de shared pointer van de unit, en zijn oude positie op nullptr gezet. Er wordt dan gekeken of de unit een ally is of niet daarop gebaseerd checken wij of zijn positie op de index 0 of LANE_SIZE – 1, zodat we kunnen beslissen of we de speler of vijand zijn hp omlaag moeten halen met de waarde van de damage variabele van de unit, als de unit aangevallen heeft wordt de waarde op zijn positie weer op nullptr gezet om de unit van het veld te verwijderen.

In de fightPhase functie wordt er gecheckt of twee units met elkaar moeten vechten, als het zo is worden de pointers meegegeven aan de fight functie de dan het gevecht uitspeelt en de resultaten ervan gebruikt om te beslissen welke units het van het bord moet verwijderen.


fightController
- gameBoard: board
- playerHP: int
- enemyHP: int
- enemyMana: int
- currentPlayerMana: int
- cardHand: fightHand &
- endTurnButtonTexture: sf::texture
- endTurnButton: sf::Sprite
- gameWindow: sf::RenderWindow&
- storage: objectStorage &
- lastTurnEndTime: uint64_t
- active: bool = false
- lastInput: uint64_t
- lastPhase: int
- allowedToEnd: std::shared_ptr<bool>
+ playerMana: int
+ fightController ()
+ initFight ()
+ getActive (): bool
+ getSkyOpen (): bool
+ getGroundOpen (): bool
+ setAllowedToEnd ()
+ nextTurn ()
+ placeUnitOnBoard (): bool
+ isDoneButtonPressed (): bool
+ draw ()
board
- lanes: lane [3]
- priorityLane: E_lane
- playerHP: lane
- enemyHP: int &
- playerMana: int &
- combatUI: UI
+ board ()
+ getSkyOpen (): bool
+ getGroundOpen (): bool
+ update ()
+ updateEnemy ()
+ updateAlly ()
+ fightPhase ()
+ isPositionEmpty (): bool
+ placeUnit (): bool
+ draw ()
+ reset ()
3
lane
- laneID: E_lane
- playerHP: int&
- enemyHP: int &
- allyArray: std::array<std::shared_ptr<unit>,
LANE_SIZE>
- enemyArray: std::array<std::shared_ptr<unit>,
LANE_SIZE>
- nullUnitPointer: std::shared_ptr<nullUnit>
+ lane ()
+ isIndexEmpty (): bool
+ placeUnit ()
+ reset ()
+ updateAllyLane ()
+ updateEnemyLane ()
+ updateUnit ()
+ fightPhase ()
+ fight (): combatResult
+ getLaneStartPosition (): sf::Vector2f
+ getIterationDistanceX (): float
+ draw ()
UI
- mana_healthFont: sf::Font
- playerManaText: sf::Text
- playerHealthText: sf::Text
- enemyHealthText: sf::Text
- manaBarTexture: sf::Texture
- healthBarTexture: sf::Texture
- enemyHPTexture
- statsUITexture: sf::Texture
- UIOverlayTexture: sf::Texture
- manaBarSprite: sf::Sprite
- healthBarSprite: sf::Sprite
- enemyHPSprite: sf::Sprite
- statsUISprite: sf::Sprite
- enemyStatsUISprite: sf::Sprite
- enemyUIOverlaySprite: sf::Sprite
- UIOverlaySprite: sf::Sprite
- manaBarMaxBounds: sf::IntRect
- healthBarMaxBounds: sf::IntRect
- playerHP: int &
- playerMana: int &
- playerMana: int &
+ UI ()
+ draw ()
+ updateUI ()