Project Gradius - zit 2

> project by Luckas Declerck - s0163224 - 2017-2018

Thematiek en werking

Op een bepaald moment vond ik dat de sprites een redesign nodig hadden, dus heb ik het spel een Flappy-bird thema gegeven. De werking blijft echter hetzelfde: de boven- en onderkant zijn obstakels waar je niet tegen mag vliegen *(hier zit echter nog een kleine bug ivm. collision detection)*, zowel de player als de enemies kunnen gele kogels schieten, en de groene buizen zijn sporadische obstakels die de speler moet ontwijken.

De Levels (die als json-bestand kunnen ingelezen worden) geven parameters over de snelheid van de entities, hun levens, het aantal Waves, hoeveel enemies per wave,… Naarmate het level stijgt (levels 1, 2 en 3 zijn meegegeven), kunnen deze parameters moeilijker gemaakt worden. Levels moeten worden ingevuld in de config-file, en worden in die volgorde gespeeld. (je begint bij level 1, en als je wint krijg je je score en kun je naar het volgende level gaan)

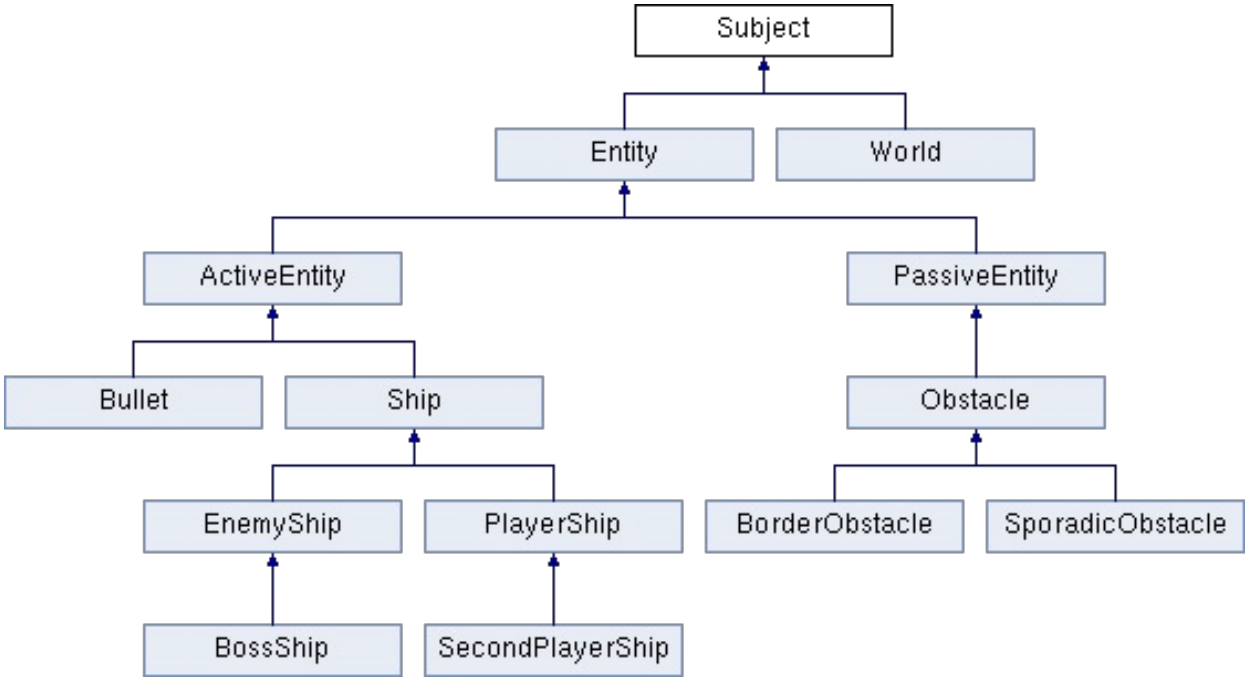
Design keuzes

Voor dit project heb ik me vastgehouden aan de opgegeven structuren en patronen. Een klein overzicht:

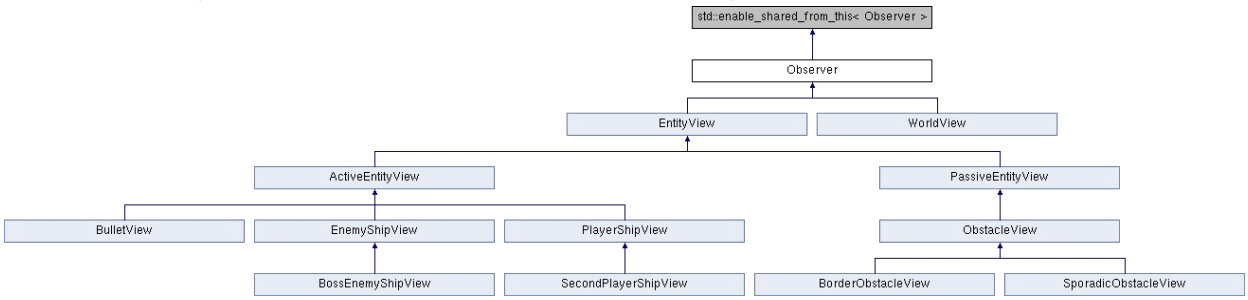
MVC-structuur

Dit project is qua structuur in drie delen gesplitst. De pure logica van het spel bevindt zich in het Model-gedeelte, terwijl de visuele weergave in het View-gedeelte zit. De spelbesturing, de collisioncontrole, de (zeer eenvoudige) AI en het lanceren van obstakels behoort in dit spel tot het Controller-gedeelte.

In het Model-gedeelte erven alle entiteiten af van een Entity-classe. deze zijn nog opgesplitst in een ActiveEntity en een PassiveEntity, het eerste staat voor bewegende objecten die voor interactie zorgen (Ships en Bullets) en PassiveEntity wordt eerder gebruikt door de obstakels. De Entity-classe erft vervolgens af van de **Subject**-klasse, die voor het *Observer-pattern-implementatie* zorgt (zie verder)



In het View-gedeelte erven alle entiteiten over van **EntityView**. Hierin staan de functies om een enteit weer te geven etc. Deze klasse erft af van de **Observer**-klasse, aangezien deze notificaties krijgt van een subject (het model) om vervolgens het object aan te passen.



Al onze modellen worden bijgehouden in een object van de **World**-klasse. Deze houdt lijsten van pointers naar objecten bij. Bij wijzigingen (door de entiteiten zelf of door input van de controllers) worden er notificaties gestuurd naar zijn observer, namelijk een object van de **WorldView**-klasse. Deze heeft dan analoog pointers naar EntityViews. Hierin staan functies om Entityviews aan te maken, te destroyen (als ze dood zijn),…

Observer Pattern

Zoals kort hierboven omschreven zort het observer-patroon voor de communicatie tussen het Model en de Views. Wanneer er bijvoorbeeld een enemyShip wordt toegevoegd (bij bv. het initialiseren van een level) gaat de World klasse aan zijn observer (een WorldView object) laten weten dat er een nieuwe enemy is. WorldView gaat dan een nieuw enemyShipView object genereren en gaat het linken aan het laatst toegevoegde enemyShip model. Dit principe geldt voor alle entiteiten in het spel.

Controllers

In het spel zijn er meerdere controllers die de werking van het spel besturen:

- **GameController**: dit is de *main game-loop* en roept alle controllers op.
- **KeyController**: een simpele klasse die keyboard input inleest op basis daarvan entiteiten laat verplaatsten/schieten
- **AIController**: deze zorgt ervoor dat de enemies zichzelf kunnen verplaatsen volgens 3 extreem eenvoudige statements:

```
if (targetInRange()) {
            fireAtTarget();
        }
        else if( bulletComingTowardsMe()){
            moveAwayFromBullet();
        }
        else {
            wanderAroundAimlessly();
        }
```

- **CollisionController**: deze controller kijkt of er collisions zijn met bullets en playerShip, of bullets/obstacles out of range gaan, of playerShip een obstacle raakt, bestuurt 'Waves' of enemies,…

Gamestates

Bij de uitbreiding van het spel moesten we oa. Scores en extra levels toevoegen. Het leek me interessant om het spel wat uit te breiden, los van de 'gameplay'. Hiervoor heb ik me op een tutorial gebaseerd. Aan de StateController kan men staten toevoegen (pushen) of afhalen (poppen). Als men bv dood gaat in het spel, wordt de spelstaat gepopt, waardoor men in de staat eronder, de game-over staat komt. Een verdere uitbreiding zou het pauzeren vh spel zijn, ipv gehardcode knoppen kan men een knop linken aan het pushen/poppen van de pauze-staat.

Uitbreiding

voor de uitbreiding van het spel zijn er enkele dingen toegevoegd, zoals hierboven beschreven. Volgende zaken zijn in de reeds bestaande code aangepast:

- Het spel is in een gamestate gegoten, dit maakt de menu wat eenvoudiger (de vorige (Game.cpp die het spel bevatte zit nu in GameState))

- namespaces zijn gebruikt voor de custom Exception classes
- File paths en andere magic numbers zijn uit de code gehaald en worden uit de config.json file geparsed

Verder zijn de verplichte implementaties voor 2e zit uitgevoerd.