Verslag: project Advanced Programming

Als project kregen we als opdracht om een interactieve game geïnspireerd door Doodle Jump te ontwerpen en implementeren. We doen dit dan aan de hand van C++ en een graphics libray genaamd SFML. Omdat we ervoor willen zorgen dat de code ordelijk, flexibel in toevoegingen en zowel met als zonder graphics library zou moeten werken, moeten we een goed design ontwerpen en implementeren.

Het design

Om te beginnen hebben we een onderscheid gemaakt tussen de logic en de representation van de game. Hiervan is de bedoeling dat we het spel al kunnen simuleren zonder dat een visuele weergaven nodig is. We kunnen door dit onderscheid ook gebruik maken van eender welke graphics library.

Logic

Het logic gedeelte bevat bepaalde klassen met nodige design patterns.

Het eerste design pattern waar ik aan begonnen was, was het **observer** pattern. Deze wordt gebruikt om te observeren over een subject. Het subject is een “voorwerp” in de game die bepaalde acties zal moeten uitvoeren die met het spel te maken hebben. Het subject wordt dan geobserveerd door een observer. Beide klasse (subject en observer) zijn “Abstract” classes, deze hebben alle bij ook nog een concrete klasse ofwel de subclass genoemd. Deze subclass gaat meer in detail over de abstract klasse en zal de informatie bevatten die nodig zal zijn in het spel. Het subject zal zelf een lijst bijhouden van observers. We kunnen deze observers dan raadplegen met een Notify() functie. Deze zal dan telkens een Update() functie op de observer aanroepen, deze update functie zal de observer op de hoogte brengen wat er veranderd is aan het subject. Het is wel enkel de bedoeling dat de Notify() functie wordt aangeroepen wanneer er een verandering is gebeurd aan het subject. We roepen dus enkel de observer functie aan in de move functies van de concrete class van het subject.

De observer zelf zal ook nog een score als subclass hebben, de score zal ook een observer van de player zijn. Het is de bedoeling dat hij de player obseveerd en vanaf deze een nieuwe maximale hoogte bereikt de score zal aanpassen.

Nadien hebben we de **world** geïmplementeerd. Deze staat in om het spel te laten functioneren en entities aan te maken, verwijderen, checken op collisions. Wanneer de World zal worden aangemaakt, zullen er als eerst een speler, platformen, een score en achtergrond tiles gemaakt moeten worden. Deze worden dan gemaakt door het AbstractFactory design pattern (hier later meer over). Nadat deze worden aangemaakt worden deze allemaal opgeslagen in een lijst van shared_ptrs, het gebruik van smartpointers is veel handiger zodat je zelf de pointers niet moet gaan verwijderen wanneer deze niet nodig zijn, je moet enkel wel opletten als smartpointers in een container zitten, dat je wel eerst deze container leeg maakt voor je het gehele object verwijderd anders worden deze niet vrijgegeven en treden er memory leaks op. Hiervoor heb ik dan ook een releaseObservers() functie gemaakt die aangeroepen wordt, net voor de world verwijderd wordt. Ik heb dan ook voor shared_pointers gekozen omdat er op 2 plaatsen naar deze entities wordt verwezen, zowel in de World als in de Observer.

Voor de collision detection heb ik er mee rekening gehouden dat deze altijd zal werken, zelf ook als de game niet voldoende fps zou halen. Dit heb ik gedaan door het bewegingsspoor van mijn player te gebruiken voor de detectie. Het spoor bestaat tussen de huidige en vorige frame van het spel. Nadien gaan we zien of dat eender welk object (bonus of platform) dit spoor kruist als dat zo is, is er

een collision. Het was eerst de bedoeling om een functie te maken voor de collision detection die gebruikt kon worden voor zowel de bonussen als platformen. Maar toen merkte ik op dat de breedte van een platform groter was als dat van de player en dat van de bonussen kleiner als de player. Hiervoor heb ik dan 2 functies gemaakt zodat er bij de platformen wordt gekeken of de player tussen de uiteindes van het platform zit en bij de bonussen wordt gekeken of de bonus tussen de uiteindes van de player zit.

Hierna kwam het **Abstract Factory** design pattern aan de pas. Dit zorgt ervoor dat de entities worden aangemaakt. Omdat deze klasse zelf een volledige virtual class is, is hier verder niet veel over te zeggen. Het bevat enkel pure virtuele functions die worden overgeschreven in de concrete factory, maar hier meer over omdat dit tot het representation gedeelte hoort.

Als laatste gedeelte van het logic gedeelte hebben we nog 3 **singelton** implementaties:

1. De **Stopwatch** class**,** deze zal ervoor zorgen dat de game logic altijd op de zelfde snelheid zal verlopen, de fps zelf zal hier geen inspraak op mogen hebben. We hebben deze klasse dan ook als singelton pattern geïmplementeerd omdat het volledig spel gebruikt maakt van eenzelfde stopwatch, die ervoor zorgt dat alles op de juiste manier en tijd wordt uitgevoerd.
2. De **Random** class, Dit zorgt ervoor dat alles random wordt genereerd. Het is ook de bedoeling dat dit voor elke game anders is, dus we moet altijd de seed veranderen. Voor meer uitleg over hoe de kansen zijn voor iets/wat te genereren, staat dit uitgelegd in de comments in mijn cpp file van de Random class. Dit is ook een singleton implementatie omdat er hiervoor ook maar 1 instance nodig is en het zo overal makkelijk te bereiken is.
3. De **Camera** class, deze zorgt voor het omzetten van mijn logic points naar geprojecteerde pixels. We werken hier dan ook met een offset voor het projecteren zodat altijd mijn player in beeld blijft. Want het is de bedoeling dat enkel mijn player naar boven beweegt en de rest van mijn entities blijven staan. Waardoor we de camera bewegen (met de offset) i.p.v. Alle entities. De implementatie reden is net hetzelfde als de 2 vorige.

Als toevoeging in mijn logic heb ik nog een extra klasse gemaakt genaamd ControllingPointers. Hiermee kunnen we wanneer we functies/argumenten van het type pointer willen gebruiken eerst controleren of de pointer daadwerkelijk wel naar iets verwijst en geen nullpointer is. Moest het zo zijn wordt het programma gestopt en wordt een message weergeven dat zegt dat het een nullpointer is, in welke class en zijn functie dit voorkomt, zodat het makkelijk te “onderzoeken” valt.

Representation

Dit gedeelte zal vooral de weergeve zijn van het logic gedeelte.

Als eerst was ik hier begonnen met het implementeren van de subclass **entityview** deze heeft als base class Observer. De entityview zal zorgen voor de weergeven van alle entities. Deze classe bevat ook een entitymodel zodat we aan de geprojecteerde pixels kunnen komen van het entitymodel. We

Daarna zijn we begonnen aan de **Window** class deze klasse is ook een **singleton implementatie**, ik heb hiervoor gekozen omdat er maar 1 instance nodig is van de window en deze op verschillende plaatsen nodig is. Zo is deze dan ook makkelijk de bereiken. Het doel van de window klasse is om alles weer te geven op een window. In deze classe controleer ik ook op event handeling zodat er interactie kan zijn met het keyboard voor de speler te bewegen en voor het sluiten van de window zelf.

Om alle entities aan te maken hadden we een AbstractFactory aangemaakt in het logic gedeelte. Ik heb toen vermeldt dat we hier nog een subclass van gingen maken genaamd **concreteFactory**, daar komen we nu aan.

Het doel van de concreteFactory is voor een specifieke entity een functie aan te maken waar deze wordt gecreëerd. Aan de functie wordt dan een entitymodel meegeven zodat de gemaakte entityview hier toegevoegd kan worden aan de lijst van observers van het entitymodel. In de concreteFactory ga ik zelf verwijzingen bijhouden naar elk gemaakt entityview, dit voor de reden dat we wel altijd een view kunnen tekenen, maar niet per se genotified moet worden. Wat ik hiermee bedoel, ik had eerst in mijn update() functie van de entitymodel klasse een draw(sprite) staan waardoor ik altijd de notify functie moest aanroepen om mijn entityview weer te geven. Dit is natuurlijk niet bedoeling, want dat is minder efficiënt + het observer pattern is dan van minder belang. Ik heb dan dus telkens een extra verwijzing naar een entityview gemaakt waardoor dit allemaal shared_ptrs worden (ik had eerst hier unique_ptr) voor, het gebruik van de shared_ptrs komt dan ook later nog van pas om de views uit de concreteFactory te verwijderen. Het ding is dat we zowel in mijn subject als in mijn concreteFactory een verwijzing hebben naar het entityview. We hebben dus een use_count gelijk 2. Wanneer we een entitymodel verwijderen omdat deze niet meer gebruikt moet worden, verwijderen we dus de lijst van observers die deze bij houdt ook. Waardoor de use_count naar 1 gaat. Aan de hand hiervan kunnen we dan ook wanneer we de views gaan tekenen, ineens controleren of de use_count gelijk is aan 1. Wat ervoor zorgt dat deze view niet meer getekend moet worden en dus ook niet meer moet bijgehouden worden in de concreteFactory.

Als laatste deeltje hebben we de **Game** class zelf, hierin wordt geswitched tussen menu en de game. Dit wordt gedaan aan de hand van een data member in de world class die zegt of de game actief is of niet. In de Game class is er ook een loop die een update functie van de world zal aanroepen en een drawViews() functie die zelf in de game class bevindt. Hierin worden alle draw() functies van de views aangeroepen. We kunnen hieraan door een shared_pointer naar de concreteFactory bij te houden, we maken deze shared omdat we een extra referentie hier naar willen hebben want deze wordt ook nog gebruikt in world maar als een AbstractFacotry verwijzing.

Als klein extratje heb ik een saveScore() toegevoegd wat de highscore en vorige score opslaagt in een bestand. Telkens als het spel terug wordt opgestart wordt dit bestand uitgelezen zodat we de data terug hebben die we hadden toen we het spel afsloten.