Anneline Daggelinckx
studentennummer: 20171591
Wiskunde Ba3

Gevorderd programmeren

Verslag Project Space invaders

Observer pattern

Allereerst heb ik een abstracte klasse “Observer” en een klasse “Subject”. Deze representeren samen het “observer pattern”. Observer heeft een virtuele functie “update”, zonder argumenten. Subject heeft een lijst van observers en een methode “notify”, die voor elke observer de functie “update” aanroept. De observers in mijn project zullen “Sprites” zijn, die het visuele aspect representeren van hun subject, namelijk “Entities”. Het voordeel dat ik het observer pattern toepas tussen Entities en Sprites, is dat elke keer dat een entity beweegt, doodgaat, … enkel de bijhorende sprite moet worden aangepast. Op deze manier moet niet alles elke tick geüpdated worden.

Namespace model

Vervolgens is er een klasse “Entity”. Het is een “child class” van “Subject”. Een Entity heeft coordinaten, snelheid, gezondheid (het aantal levens), een hoogte en een breedte. De belangrijkste methodes zijn “move” en “collidesWith”. “Move” zal de coordinaten van het object aanpassen, rekening houdend met de snelheid van de entity en de tijd die is verstreken. Vervolgens zal hij ook de functie “notify” van zijn super class aanroepen, want de coordinaten van zijn observerende sprite moeten natuurlijk ook worden aangepast. “CollidesWith’ krijgt een referentie naar een andere entity mee en checkt of deze botsen (overlappen). Hier wordt handig gebruik gemaakt van polymorfisme, want een instantie van een childclass van Entity kan botsen met een instantie van eender welke andere childclass. De childclasses van Entity zijn “Player”, “Enemy”, “Bullet” en “Shieldblock”. Er kunnen dus makkelijk nieuwe concepten zoals “Power Up” of “Boss” worden toegevoegd, die dan ook meteen collision detection kunnen gebruiken en kunnen bewegen.

“Model” is de klasse die alle entities van de game bijhoudt. Zo heeft hij een player, een lijst van enemies, en twee lijsten van bullets; enemy bullets en player bullets. De reden dat er twee lijsten van bullets zijn, is dat bij de collision detection enkel de nodige collision-checks moeten worden uitgevoerd (er moet bijvoorbeeld niet gecheckt worden ofdat enemy bullets andere enemies raken). Het handige aan deze klasse is dat aan het begin van de game, per level een Model kan worden aangemaakt, en bij het veranderen van level moet gewoon naar een andere model gekeken worden. Zo kan het nooit voorvallen dat tussen twee levels het spel crasht omdat bijvoorbeeld de volgende level file niet leesbaar is. Ook kan het zo nooit voorvallen dat de game in een ongeldige state zit tussen de state van het huidige level en het volgende.

De constructor van Model krijgt de bestandsnaam van een ini file mee waarin specificaties van de entities staan opgelijst. Op deze manier worden level specificaties meteen in het model opgeslagen. Hierover later meer.

Namespace view

In de namespace “view” heb ik eerst een childclass van “Observer” genaamd “Sprite”. Deze observeert een “Entity” en bepaalt hoe deze er uit zal zien en zijn positie op het scherm. De klasse heeft dus als members een sf::Sprite, die zelf een positie heeft, en een sf::Texture. De reden dat de klasse “Sprite” nog een aparte texture opslaat, is dat als je een texture wil toekennen aan een sf::Sprite, je eerst een aparte sf::Texture moet aanmaken. Deze laatste wordt verwijderd van zodra hij out of scope gaat, en de Sprite verliest dan zijn Texture. Een alternatief was de afbeeldingen apart opslaan in “Sprite” in plaats van de texture, maar dan zou elke keer dat alles op het scherm getekend wordt, een nieuwe texture met deze afbeelding moeten worden aangemaakt, om deze vervolgens aan de sf::Sprite toe te kennen.

De “update” methode van “Observer” wordt overschreven zodat elke keer dat er iets verandert aan de entity dat hij observeert, dit ook in de Sprite wordt aangepast (bijvoorbeeld positie). In de constructor wordt een png bestand dat de entity representeert ingelezen en opgeslagen in de sf::Texture (van SFML).

De “View” klasse heeft shared pointers naar al deze specifieke sprites (een sprite die de Player vertegenwoordigd, sprites voor de Enemies enz). Met de methode “render()” tekent hij ze op het scherm. Mijn initiële idee was van deze klasse ook een observer maken zodat er enkel getekend moet worden als er iets is veranderd. Maar aangezien de enemies in mijn versie van Space invaders continu bewegen, en niet in blokjes, zou dit geen voordeel leveren, er verandert namelijk altijd iets. De reden dat elke keer alles wordt getekend en niet enkel de sprites die zijn veranderd, is dat het enige dat het enige dat blijft staan de schildjes zijn (en de player als deze niet beweegt), dus het zou geen voordeel opleveren kwestie performance.

Namespace controller

Ten slotte is er de controller. Hij krijgt een shared pointer naar de view en een naar het model. De belangrijkste methoden zijn handleEvent, die wordt aangeroepen als er een event plaatsvindt (bijvoorbeeld: de gebruiker drukt een toets in) en een functie update(), die zorgt voor de veranderingen die elke tick moeten gebeuren (de enemies die bewegen, de kogels die bewegen en collision detection). Ook het schieten van de enemies gebeurt in update. Enkel de enemies op de frontline kunnen schieten, en elke enemy kan maar om de zoveel seconden schieten. Verder wordt een random waarde gegenereerd om te beslissen of een enemy schiet of niet. Op deze manier is het steeds onvoorspelbaar of er een schiet, en zo ja, welke.

SpaceInvadersGame

De klasse SpaceInvadersGame heeft als datamembers een Model, View en een Controller en kan aan de hand hiervan het spel starten. Eerst wordt de load functie aangeroepen. Deze leest de meegegeven level files in, in verschillende Models, en initialiseert de View en Controller. De methode “loadNextLevel” verandert het model zodat het volgende level gespeeld kan worden.

In een methode start() zit de gameloop. 10 seconden nadat het spel is gestart zal er ook muziek beginnen spelen. De gameloop zal elke loop eerst de functie “update” van de controller aanroepen en view alles laten tekenen. Vervolgens zal hij gedurende overblijvende tijd van 10 miliseconden wachten op events. Dit garandeert een constante tick rate en laat toe dat er meerdere events per tick worden behandeld. Als er een event is, zal hij de functie handleEvent van de Controller aanroepen. Als het scherm niet gesloten wordt, wordt de gameloop weer herhaald.

Files inlezen

Ik heb gekozen voor INI-files omdat ik slechts enkele dingen wil specifieeren, en niet heel de state inlezen. Dit zorgt ervoor dat het model niet kan geserialiseerd worden, maar wel makkelijk aangepast. In deze files kan gespecifieerd worden hoe snel de enemies gaan en hoe veel het er zijn (beiden zowel horizontaal als verticaal), hoeveel levens de player heeft, het aantal shields en de vorm van de shields. Deze laatste kunnen worden ingegeven door #’s of X’en waar je een shieldblokje wil, en underscores voor de ruimte tussen de blokjes. Met dezelfde logica had ik ook de enemies kunnen inlezen, zodat ook de vorm van een “enemyblock” gekozen kon worden.

Bij het inlezen worden de ingegeven waarden gecheckt (of de waarden geldig zijn, of de enemies niet te snel/traag gaan, de shields wel op het scherm passen…) en als er een fout wordt gevonden, zal er een melding worden geschreven naar std::out. Vervolgens zullen er defaultwaarden genomen worden voor de fout ingegeven waarde zodat de game toch gespeeld kan worden.