Space Invaders: Report

Jakob Struye
UA 2BINF

1 Datamodel design: de Entity-klasse

Alle mogelijke entiteiten op het speelveld van Space Invaders zijn afgeleid van de abstract base class Entity. Entity is een rechthoekige entiteit met een locatie, een horizontale en verticale grootte en health points. De basisklasse bevat ook de nodige getters en methods om de HP te decrementeren, botsingen met andere Entities te detecteren en te controleren of de Entity zich binnen een bepaald venster bevindt. De concrete entiteit Shield wordt rechtstreeks van de Entity afgeleid. Daarnaast is ook MobileEntity, met move-methode, van Entity afgeleid. Van MobileEntity wordt Bullet afgeleid, en daarvan AlienBullet en PlayerBullet. Bullets hebben een vaste richting waarin ze bewegen, namelijk naar beneden voor AlienBullets en naar boven voor PlayerBullets. Van MobileEntity is ook ShootingEntity, met een puur virtuele shoot(), afgeleid. Ik ga ervan uit dat alle entiteiten die kunnen schieten, ook kunnen bewegen. Moest ShootingEntity direct van Entity zijn afgeleid, was er multiple inheritance nodig voor entiteiten die zowel schieten als bewegen, wat hiermee vermeden is. Van ShootingEntity wordt PlayerShip (met implementatie van shoot()) en Alien (met puur virtuele method getScore(), die teruggeeft hoeveel punten de Alien waard is) afgeleid. Van Alien wordt verder BonusAlien (met implementatie van getScore() en lege implementatie van shoot()) en RegularAlien (met implementatie van shoot) afgeleid. Ten slotte worden van RegularAlien nog drie soorten afgeleid, namelijk RegularAlienBot, RegularAlienMid en RegularAlienTop. De drie soorten leveren elk een ander aantal punten op, wat wil zeggen dat ze alle drie getScore() implementeren. Merk ook op dat RegularAlien bijhoudt of de entiteit botste met een andere entiteit die geen Bullet was. Dit is nodig omdat het spel eindigt wanneer een RegularAlien botst met een Shield of PlayerShip. Zo'n botsing wordt dan buiten de entiteit verder afgehandeld. Zie bijgevoegde diagram Design.pdf voor een overzicht van de structuur.

2 Model-View-Controller

Ik heb gekozen voor een implementatie van MVC met 1 model, 1 view en 1 controller. Meerdere controllers/models leek me voor een project van deze grootte onnodig.

2.1 Model

De model omvat alle dataopslag en functionaliteit (logica) van het spel. De model bevat een aantal data members die al in de lijst met data members geïnitialiseerd worden. Deze kunnen aangepast worden om het spelverloop aan te passen (bv sneller bewegende RegularAliens, meer/minder levens, ...). Verder zijn er heel wat andere data members die nodig zijn voor de logica van het spel. Deze worden nog niet meteen in de lijst met data members geïnitialiseerd. Ten slotte worden pointers naar alle Entities per soort opgeslagen (in vectoren als er meerdere van de soort zijn). Ik heb lang getwijfeld of ik de Entities in één grote vector of in verschillende vectoren ging opslaan en ben een aantal keer van plan veranderd. Uiteindelijk heb ik geopteerd

voor verschillende vectoren. Zie onderaan dit document voor motivatie.
De step() method van de Model wordt om een bepaald tijdsinterval aangeroepen door de Controller. Eerst gaat step() de collision detection laten gebeuren. Deze checkt enerzijds alle mogelijke collisions en gaat anderzijds de Entities door voorgaande collisions vernietigd afhandelen (precieze afhandeling hangt af van het soort Entity). Daarna wordt gecontroleerd of het spel mag blijven doorgaan en worden de steps() voor de verschillende soorten MobileEntities aangeroepen (concreet zorgen deze voor het bewegen en eventueel schieten). Via het observer pattern worden Controller en View op de hoogte gesteld dat een stap gebeurd is. Daarbij wordt uitdrukkelijk vermeld of het spel nog bezig is en of er net een nieuw level begon (aangezien acties door observers ondernomen rechtstreeks daarvan afhankelijk zijn).

2.1.1 Controller

De taken van de Controller zijn enigszins beperkt. Om een spel te initialiseren, wordt een Controller aangemaakt, die dan zorgt voor het aanmaken van Model en View. De Controller verwerkt ook de input en gaat om een bepaald tijdsinterval (afhankelijk van de FPS die kan worden ingesteld in de lijst met data members van de Controller) de step() van zijn Model aanroepen. Verder gaat de Controller bij het starten van een nieuwe game na een game over een nieuw Model aanmaken.

2.1.2 View

De View genereert via SFML een grafische weergave van de staat van de Model. Bij initialisatie worden alle Textures alsook een Font voor tekst ingeladen en opgeslagen in een struct. Eventuele ontbrekende files worden via exceptions afgehandeld. In geval van een ontbrekende Font wordt gewoon geen tekst weergegeven en in geval van ontbrekende Textures worden (alle) Entities afgebeeld als witte rechthoeken. In de .draw() method (aangeroepen wanneer de View via Observer pattern van een verandering in de Model op de hoogte gesteld wordt) worden eerst alle Sprites gegenereerd en getekend, en daarna nog een HUD met huidige score, levens en level. Er zijn aparte draw() methods voorzien voor gepauzeerde games of game over. Merk op dat de keuze om Entities per soort op te slaan hier voor een iets langere code zorgt, maar wel hopen dynamic_casts en een ingewikkelder ogende berekening van de scale (of herberekening voor elke Entity) vermijdt. Ook is de keuze van Textures niet bij alle soorten Entities van hetzelfde afhankelijk (bij RegularAliens wordt er tussen Textures geswitcht, bij Shields is de keuze afhankelijk van de HP). Keuze van Texture naar de Entity zelf verschuiven zou een violation van MVC zijn.
Deze view is geoptimaliseerd voor 800*600, maar is ook speelbaar in andere resoluties. De Entities zullen correct scalen, alleen zal de HUD niet meer op de goeie plaats staan. Om de HUD uit te schakelen kun je data/arial.ttf verwijderen. De Controller bepaalt de grootte van het speelveld, wat ook als resolutie van het venster gebruikt wordt.

3 Factories

Er is een abstract factory die Entities aanmaakt, met daarvan afgeleid factories voor elke concrete soort Entity. De factories worden overal waar Entities gegenereerd worden gebruikt (zowel in de Model als voor Bullets binnen de ShootingEntities). De ShootingEntity heeft als data member een pointer naar de correcte concrete factory.

4 Namespaces en filehiërarchie

Model, View en Controller krijgen elk hun eigen namespace en map. De losse Entities bevinden zich in een submap van het Model. Factories zitten in de namespace Factory en bevinden zich in een submap van het Model (dit omdat enkel het Model de Factories gebruikt). Verder is er nog een submap van de source root genaamd Other. Hier bevindt zich alles door meer dan één deel van de MVC gebruikt en niet direct gerelateerd aan MVC (concreet: een enum voor richtingen en de abstract base class voor observers)

5 Eén vs meerdere Entityvectoren

Op de collision detection na, is de functionaliteit van een Entity sterk afhankelijk van het soort Entity. Een step() toevoegen aan de Entity, om zo iedere stap in het spel voor elke Entity binnen de Entity zelf te laten verlopen was geen optie, omdat verschillende Entities te veel verschillende informatie vanuit de Model of vanuit andere Entities nodig hebben. Zo gebeurt het bewegen van een Bullet elke frame onconditioneel, terwijl een PlayerShip inputs nodig heeft en de grootte van het speelveld moet weten, en RegularAliens op de hoogte moeten zijn van de posities van de andere RegularAliens. Ook het afhandelen van collisions (moet het spel beëindigd worden, moet de score aangepast worden, moet een countdown tot respawn gestart worden, ...) is sterk afhankelijk van het soort Entity. Dit zou bij één grote Entityvector voor heel veel dynamic_casts zorgen. Daarnaast is soms slechts één soort Entity nodig, bv. enkel de RegularAliens wanneer moet bepaald worden of ze nog verder in een bepaalde richting kunnen bewegen of moeten omkeren. Ook is het onnodig om alle RegularAliens te overlopen wanneer ze tijdens een stap niet moeten bewegen of schieten. Bij één grote Entityvector zou dit dus voor heel wat traverses zorgen terwijl de meeste Entities niet gebruikt worden. Het enige geval waarin één vector handiger zou zijn, is de collision detection, aangezien deze enkel een method van de Entity aanroept, en verder enkel iets hoeft te weten over de andere Entity waarvoor op collision gecheckt wordt. Om hier toch polymorfisme toe te kunnen passen, heb ik gekozen om enkel in de collision detection alsnog een tijdelijke vector met alle Entities te maken en die te traversen. Een vector van een zeventigtal pointers extra zal geen groot verschil maken in het geheugengebruik van het programma. Verder is een eventueel kortere execution time van één step() bij één Entityvector geen reden om het te gaan invoeren. Op mijn systeem gebruikt het berekenen van één stap aan 30FPS ongeveer 2% van de beschikbare tijd (en dat is inclusief genereren en tekenen van grafische output), dus is het niet erg zinvol om te gaan proberen die execution time verder in te korten. Ik heb dus gekozen voor de optie die voor de gemakkelijkst te lezen code zorgt. Meerdere vectoren vereist enkel wat extra lijntjes code om één grote vector aan te maken bij de collision detection, terwijl één vector op heel wat plaatsen hopen dynamic_casts zou vereisen. Ook valt het nut van zo'n polymorfe vector grotendeels weg als er zoveel dynamic_casts nodig zijn.