Project Gevorderd Programmeren: Gradius

*Door Benjamin Vandersmissen, studentennummer: S0164228*

Mijn implementatie van Gradius bestaat uit 2 grote delen, er is een deel dat de algemene data van de game bijhoudt en een deel dat te maken heeft met entities, alle mogelijke objecten in een level. Beide delen zijn geimplementeerd in Model-View-Controller structuur, bij het deel met de algemene data zijn deze classes models::Model, views::View en controllers::Controller. Entities zijn van de vorm models::Entity, views::Entity en controllers::Entity.   
De MVC structuur is als volgend geimplementeerd :

- Een model heeft een pointer naar een view en een controller

- Een controller heeft een pointer naar het model

- Een view heeft een pointer naar het model

Elke game tick wordt de controller geupdate, met onder andere keyboard input en andere events, de controller beslist dan of hij het model moet op de hoogte stellen van de events door **notify()** te callen of niet en of hij event moet doorgeven aan de controllers van de entities. Het model kan dan eventueel ook de view op de hoogte stellen als er iets aangepast is in zijn state, ook door **notify()** te gebruiken. Entities zitten op dezelfde manier in elkaar, alleen krijgen hun controllers de events van de game controller.

Een grote focus van mij was om het spel makkelijk uitbreidbaar te maken, dit heb ik gedaan door onder andere de entiteiten volledig af te splitsen van het spel zelf, zodat de entiteiten geen pointers nodig hebben naar het model voor hun collisions bijvoorbeeld. Ook wordt er in elke namespace (models::, views:: en controllers::) een lijst (**models::list**, **views::list** en **controllers::list**) bijgehouden van alle actieve entity models, resp. views, resp. controllers. Zo is die informatie onafhankelijk van de state van de game zelf.

Een resource is een verzameling van eigenschappen voor een specifieke klasse. Resources worden ingeladen uit ini files en er kunnen dus makkelijk extra resources aangemaakt worden van ondersteunde klasses, wat dus ertoe leidt dat het aanpassen en uitbreiden van het spel wordt terug gebracht tot het aanpassen van wat waardes in een bestand. Om een nieuwe entity aan te maken uit een resource werk ik met een abstract factory method, het enige wat er moet gebeuren is resource->**create**(positie).

Levels zijn een ander onderdeel waarbij uitbreidbaarheid en gebruiksgemak centraal stond. Een level is eigenlijk gewoon een opsomming van resources en hun posities. Er is de mogelijkheid voorzien om extra resources in te laden die niet aanwezig zijn in het level zelf, maar die op andere manieren aangemaakt kunnen worden. Zo moeten er ook bijvoorbeeld resources aanwezig zijn voor kogels, ook al worden kogels niet expliciet ingeladen in het level zelf.

Bij het inlezen van levels heb ik ervoor gekozen om de performance te verhogen en de resources maar 1 keer in te laden om ze dan op te slaan in een entry in **resources::map**, elke resource krijgt dan een naam, deze is dezelfde als diegene die gebruikt wordt in het level file en deze naam verwijst naar een filename in het mapje resources. Dit is ook handig bij het schieten van een schip, want het schip heeft dan gewoon de naam van de kogel resource zodat hij die kan opzoeken en hij geen resource tussen zijn datamembers moet hebben staan.

Ook bij het starten van een level wou ik de performance verhogen, mijn oorspronkelijke plan was om alle entities gewoon direct in te laden, maar dit zou dan leiden tot veel onnodige updates van entities die nog eigenlijk niet in het scherm zijn. Niet alleen dat, maar bij complexe AI, of lange levels, zou het kunnen dat er kogels van de vijanden het scherm opkomen, terwijl er nog geen vijanden te zien zijn.. Om dit op te lossen dacht ik om een soort van dynamisch inladen van het level toe te passen. Dat wil zeggen, bij het opstarten van het level wordt alles ingeladen wat zichtbaar moet zijn plus nog een kleine offset aan de rechterkant. Deze offset is er, omdat het kan gebeuren dat een object bij het inladen al direct een klein stuk op het scherm staat, waardoor het lijkt dat het object er opeens is, dus voor een vloeiendere beweging is die offset er. Een ander probleem waar ik dan eerst op gestoten was, was dat we dan elke tick over een hele lijst moesten loopen om de x-waarden te controleren, dit heb ik opgelost door een multiset te gebruiken voor mijn entiteiten bij te houden, waarbij er door middel van een functor gesorteerd wordt op de x-waarde, dan kon ik de loop afbreken als de x-waarde te hoog werd.

Een laatste onderdeel, wat ik toch nog belangrijk vond om te vermelden, was mijn menu’s. Menu’s zijn geimplementeerd als een vector van entries. Elke entry heeft een tekts, een positie, een menuState, de menuState van de volgende entry (met [↓]bereikt men de volgende entry) en ook de menuState van de vorige entry (idem, maar met [↑]). Het voordeel van deze implementatie is dat ik redelijk gemakkelijk extra entries in een menu toevoegen, het enige wat ik dan nog moet doen is acties definieren die gebeuren als [ENTER] ingedrukt wordt, afhankelijk van welke menuState geselecteerd was.