Gevorderd Programmeren: Projectverslag

Kasper Engelen

January 14, 2018

1 Implementatie van Model-View-Controller en Observer pattern

Het project is georganiseerd rond het MVC-pattern. De gebruikersinvoer wordt verwerkt door de GameController klasse en wordt doorgegeven aan de GameModel klasse. Bijvoorbeeld: wanneer de GameController ziet dat de "omhoog" toets wordt ingedrukt, vraagt de controller aan de GameModel klasse om de richting van de speler aan te passen. De GameModel klasse verwerkt deze input door zijn toestand aan te passen. De aanpassing aan de toestand van het model wordt doorgegeven aan de GameView klasse d.m.v. het Observer pattern. De GameView geeft deze toestand dan grafisch weer door sprites te tekenen op een venster.

De GameController is verantwoordelijk voor het afhandelen van de gebruikersinvoer. Dit gebeurt d.m.v. de `handle_event()` methode. Deze methode ontvangt een IOEvent object en gebruikt deze om de Keyboard klasse te updaten. Het doorgeven van deze input gebeurt in de `handle_tick()` methode. Hierin wordt er o.b.v. de toestand van de Keyboard klasse aan de GameModel klasse gevraagd om bijvoorbeeld de speler naar boven te doen bewegen. Verder wordt de Game Tick geforwarded naar de GameModel waardoor de speltoestand wordt vernieuwt. Daarnaast is de GameController ook nog verantwoordelijk voor het bijhouden van levels en het inladen van het volgende level als het vorige uitgespeeld is.

De toestand van de GameModel klasse bestaat voornamelijk uit `std::vector`'s met pointers naar Entity objecten. Deze entity objecten houden informatie bij over de speler, de vijanden, de obstakels, etc. Deze informatie is bijvoorbeeld de positie en richting van het object, het aantal levens van de speler, etc. De GameModel klasse is afgeleid van de abstracte Observable klasse, en is op die manier gelinkt aan de GameView klasse d.m.v. het Observer-pattern. Dit wordt bijvoorbeeld gebruikt bij het aanmaken van een Bullet entity, of bij het doodgaan van een Entity object.

De GameView klasse houdt een `std::set` bij van EntityRepresentation objecten. Deze geven de toestand van hun Entity tegenhangers weer. De individuele EntityRepresentation objecten zijn gelinkt aan hun Entity tegenhangers d.m.v. het Observer-pattern. Telkens dat de positie van de Entity wijzigt, wordt de EntityRepresentation ervan op de hoogte gesteld. Daarnaast houdt de EntityRepresentation een `std::weak_ptr` bij naar de bijhorende Entity. Daardoor kan de EntityRepresentation ten alle tijden weten of de Entity nog in leven is. Elke keer dat de `GameView::render()` methode wordt opgeroepen, worden alle "dode" EntityRepresentation objecten verwijderd. Vervolgens worden alle EntityRepresentations die overblijven getekent op het scherm.

2 Entity-hiërarchie

De Entity-hiërarchie voorziet klassen die entiteiten voorstellen binnen het spel, zo is er een abstracte basisklasse (EntityBase), een abstracte basisklasse voor entities die kunnen bewegen (DynamicEntity) en voor entities die kunnen schieten (ShootingEntity). Daarnaast zijn er ook nog concrete klassen voor kogels (Bullet), vijanden (Enemy), de speler (Player), obstakels (Obstacle) en het eindpunt van het level (FinishLine).

3 SFML en IO wrappers.

Daar ik reeds eerder met de IO-libaries SDL en GLFW heb gewerkt, en dus reeds wist dat het handig is om de interface van een library af te schermen van de rest van het project, besloot ik wrappers te schrijven voor de meeste SFML-faciliteiten. Dit zijn onderandere:

- IOEvent voor `sf::Event`,
- Keyboard voor `sf::Keyboard`,
- Sprite voor `sf::Sprite` en `sf::Texture`,
- Text voor `sf::Text` en `sf::Font`,
- Window voor `sf::RenderWindow`,
- Vec2D voor `sf::Vector2f`.

Dit bleek net zoals bij mijn eerdere projecten zeer handig te zijn, ik kon op deze manier d.m.v. constructoren, getters en setters precies bepalen hoe de SFML-faciliteiten benut en gemanipuleerd zouden worden. De meeste van deze IO-wrappers zitten vervat in de `game::IOhandlers` namespace. Vec2D zit echter in `game::utils`.

4 Levelformaat en -parsing

Het levelformaat bestaat uit JSON files die informatie meedelen over de speler, de verschillende types van vijanden, obstakels, en het eindpunt van het spel. Daarnaast zit er in de JSON file ook nog een matrix waarin verduidelijkt wordt hoe een level eruitziet. Het inladen van een level gebreurd door het oproepen van de `level::parse_level()` functie voorzien van de filename van de level-file. Deze functie returnt een Level object met daarin alle informatie. De informatie over de entities aanwezig in het level zit vervat in LevelEntity objecten (LevelEntity, LevelPlayer, LevelEnemy, LevelObstacle, LevelFinishLine).

5 Exceptionstructuur

De exceptionstructuur van het project heb ik overgenomen uit een eerder project. Het concept is dat er een basis klasse is die alle logica en datamembers voorziet. Indien men een nieuwe exception wilt voorzien moet men enkel in de constructor van de nieuwe exception de constructor van de basis klasse oproepen, zonder dat men `what()` of een gelijkaardige methode moet overloaden. De basisklasse (BaseException) maakt daarnaast ook nog het onderscheid tussen de reden en de identificatie van de exception. Een reden is waarom de exception heeft plaatsgevonden, bijvoorbeeld “Cannot load OpenGL”, “Cannot Initialze Window.”, etc. De identificatie van de exception specifieerd waar de exception thuishoort binnen de hiërarchie. Bijvoorbeeld: “`EXCEPT::GAME::WINDOW::INIT`”.

6 Extra klassen.

Daarnaast zijn er ook een aantal andere klassen die nog aanwezig zijn in het project, dit zijn voornamelijk “utility”-klassen.

De Singleton klasse is een getemplatiseerde klasse die het Singleton-pattern voorziet voor afgeleide klassen. De StopWatch, Transformation, Keyboard, CoordTransform en Settings klassen zijn hiervan afgeleid.

De projectopgave stelt dat we de Transformation en Stopwatch klassen moeten ontwerpen en implementeren. Ik heb de “canonieke” versies van deze klassen ontworpen en geïmplementeerd zoals het in de opgave stond, dit resultuurde in de Transformation en de StopWatch klasses. Daarnaast wou ik nog mijn eigen versie van deze klassen maken wat resulteerde in de CoordTransform en LoopTimer klassen.

De CoordTransform klasse is wat geavanceerder dan de Transformation klasse. Kortgezet projecteert het alle coördinaten binnen een bepaalde horizontale en verticale range gecentreerd rond de speler op het scherm. Indien de speler te dicht bij de rand van het level komt, blijft het scherm stilstaan. Zowel de schermresolutie, de positie van de speler, als de ranges kunnen ingesteld worden. Ik heb hierin wel afgeweken van het [-4, 4] x [-3, 3] coördinatensysteem, maar ik vind dat het resultaat zo veel beter is. Indien men toch het [-4, 4] x [-3, 3] coördinatensysteem wenst zoals in de opgave, verwijs ik naar de Transformation klasse. De CoordTransform klasse laat toe dat de speltoestand en de controller volledig gescheiden zijn van de spelweergave. Zo kan de GameView klasse aan de CoordTransform klasse vragen om een spelcoördinaat afkomstig van de GameModel klasse om te zetten naar een schermcoördinaat.

De LoopTimer klasse is verantwoordelijk voor het begeleiden van de game-loop. Dit laat toe om dynamische FPS en statisch TPS (ticks per second) te bekomen. Hierdoor wordt er optimaal gebruik gemaakt van de grafische capaciteiten van het platform. Daarnaast voorziet de LoopTimer ook performance informatie zoals FPS en TPS-readouts.

De Settings klasse voorziet informatie over spelinstellingen. Dit zijn onderandere de keycodes voor de spelbesturing, de schermresolutie, de filenames voor de level JSON files, etc.