Exercise 7

Software Development 2018

Department of Computer Science
University of Copenhagen

Emil Møller Hansen <ckb257@alumni.ku.dk>,
Casper Bresdahl <whs715@alumni.ku.dk>,
Torben Olai Milhøj <vrw704@alumni.ku.dk>

Version 1;
Due: Friday, March 16th

1 Indledning

Spillet Galaga er et 2D spil hvor man styrer et rumskib der kan flyve til højre og venstre. Målet er at skyde alle rumvæsnerne inden de når bunden af skærmen. Implementationen vi har lavet er akkumuleret over 3 afleveringer, hvor vi efter hver aflevering har fået feedback. Implementationen af spillet er lavet med en game engine kaldet `DIKUArcade`. På **??** ses et screenshot fra vore implementation af Galaga.
I løbet af de tre opgaver er implementationen blevet mere og mere omfattende. Efter den første opgave havde vi en implementation der lige nøjagtig var spilbar. Det vil sige man kunne rykke og skyde, men rumvæsnerne rykkede sig endnu ikke. Efter anden opgave fik vi rumvæsnerne til at bevæge sig og de kunne starte ud i flere forskellige formationer, kaldet squadrons. Nu, efter tredje iteration har vi tilføjet en menu, man kan pause spillet og man taber eller vinder når man hhv. ikke når at skyde rumvæsnerne i tide, eller når at skyde dem alle.

2 Baggrund

Som nævnt bruger vi DIKUArcade som game engine til dette projekt. Denne har mange funktioner der hjælper os med implementationen, eksempelvis har denne en collision-detector, som vi bruger til at holde øje med om skudene rammer rumvæsnerne. Et andet element i DIKUArcade vi har brugt meget er eventhandling, som sørger for at forskellige events der sker i spillet, eksempelvis et tryk på piletasterne, bliver håndteret og at der handles korrekt på dette, for eksempelet med piletasterne vil det sige at rumskibet skal rykke sig til en af siderne.
Som implementationen blev mere og mere omfattende igennem de 3 opgaver har der været mere og mere brug for godt design af koden. Dette indebærer eksempelvis at dele programmet op i flere klasser og namespaces i flere forskellige filer og mapper.

3 Analyse

Vi har indelt problemet i forskellige concepter, der hver har deres eget ansvar, resultatet kan ses herunder. Vi har opdelt koncepterne efter koncepttyperne Know og Do, så hvert koncept har netop én type. Vores Do koncepter har hver ansvar for én opgave, eksempelvis file reader der kun skal lave et level ud fra informationen fra en fil. Vi har 6 koncepter hvoraf 4 er Do- og 2 er know-koncepter.
Vores entity creator laver for nu kun firkanter der alle er lige store, og som har et billede, da vi antager vi ikke skal implementere collision detection i denne uge. På længere sigt skal der være forskel på størrelsen og formen på de forskellige objeker.

Responsibility description	Concept name
Læser filen og parser de relevante oplysniger til en level-constructor, derefter returneres det nye level til level loader	File reader
Level class skal kunne lave objekter der kan gemme de forskellige properties for et level	Level class
Singleton der skal indeholde en dictionary af Levels	Levels keeper
Class der kender stier til levels og initialiserer file reader, når den får et level tilbage sender den det op i levels keeper	Level loader
Denne får et level-nummer fra game og læser dette level fra levels keeper, den laver en entity container til alle entities der skal tegnes	Level creater
Denne får et index fra et level array samt billedenavn der hører til dette index, hvorefter den laver en entity med koordinater baseret på index og billede fra billedenavn og returnerer denne	Entity creater

Figure 1: Koncepter og deres ansvar

Concept pair	Association description	Association name
Level Loader og file reader	Level loader kalder file reader med et filnavn og får et level tilbage	Anmoder om data
File reader og level class	File reader laver et object af typen Level gennem Level class	Genererer
Level loader og levels keeper	Level loader giver et Level til Level keeper som Level keeper gemmer	Anmoder om at gemme
Level creator og levels keeper	Level creater anmoder om et level i levels keeper	Giver data
Level creator og Entity creator	Level creator anmoder om en entity med givne parametre	Genererer

Figure 2: Associationer mellem concepter

Table 1: My caption

Concept	Attribute	Attribute description
Level loader	Levels	En liste af filnavnene på level layouts
File reader	Ingen attributer	
Level class	Level array	2D-struktur der indeholder level design
	Name	Navn på level
	Costommer	Information om costommers i dette level
	Platforms	Liste med information om hvilke dele af level layout der svarer til plat
	Decoder	Denne gemmer sammenhængen mellem karakterer og billedenavne
Levels keeper	Levels	Contatiner til alle levels
	Index	Tæller der holder styr på hvor mange levels der er I levels
Level creator	Levels	Reference levels keepers
Entity creator	Ingen attributer	

4 Design

For at gøre et spil underholdende, skal det helst være muligt at tabe (og ligeså at vinde). Af denne årsag har vi lavet forskellige states, der stopper spillet og viser en menu, som der viser "You win" eller "You lost" alt efter omstændighederne. Spillet vindes ved, at alle fjender bliver skudt. Vi gjorde et forsøg på at sørge for at skibet ville springe i luften når en fjende ramte skibet, dette lykkedes dog ikke. Derfor tabes spillet blot når en eller flere fjender går under et vist punkt. Mere præcist benyttes, at man kender til fjendernes position, idet de er Entities med en position, der kan tjekkes om hvorvidt nogen af dem er nået til bunden af skærmen. Ligeså er player en entity med samme egenskaber, foruden at dens bevægelse ikke er forudbestemt af nogen algoritme, men afhænger af hvilke inputs spilleren foretager sig (og dermed hvilke events, der udføres). Designet af playeren og fjenderne er udleveret på forhånd, og de vises visuelt ved at benytte DIKUArcades class Image, som har metoder der formår at benytte de udleverede PNG-filer med sprites og rendere dem. En anden state er, at spillet til et vilkårligt tidspunnkt kan pauses ved at trykke escape og kan fortsættes ved at trykke "continue". Dette har vist sig oplagt at implementere vha. en event, da vi allerede har en event-processor til at tjekke, hvilke keys, som bliver trykket, og i tilfælde af at den pågældende knap er "escape", da bliver der skiftet til en pause-state. For at skabe blot en anelse mere spænding i spillet, er der blevet implementeret bevægelse samt formationer hos modstanderne. Formation er skabt ved at instantiere modstanderne de rigtige steder og så dernæst sørge for, at hvert lag bevæger sig uniformt for at opretholde formationen. I den bevægende formation ZigZagDown, foregår bevægelsen eksempelvist ved at rykke på fjendernes position som en sinusbølge, således at de får en flydende bølge-agtig bevægelse. Det er også med til at sørge for, at ingen af fjenderne overlapper, da de relative positioner mellem fjenderne i hvert lag ikke forandrer sig. Dermed er der ikke årsag til at tage hensyn til, hvordan drab af overlappende fjender ville fungere, da det ikke kan forekomme. For at undgå, at denne bevægelse resulterer i, at fjenderne bevæger sig udenfor skærmen, har vi siden forhenværende aflevering fjernet nogen fjender således, at udstrækningen af bevægelsen ikke er større end hvad skærmen tillader.

5 Implementation

5.1 SquareFormation

I vores arbejde med enemy formationerne har vi fundet det nødvendigt at fjerne de enemies som spawner tættest på kanterne. Dette er for at der ikke er nogen enemies som bevæger sig ud af vores play area når vi giver dem vores ZigZagDown movement strategy. I praksis fungerer vores algoritme ved at vi først tjekker om vi skal begynde at spawne enemies på næste 'lag' af skærmen, eller om der fortsat skal sættes enemies på det nuværende lag. Når vi har fundet ud af dette tjekker vi at vores enemy ikke skal til at spawnes på lagets nulte eller syvene plads. Vi tjekker dette ved at udregne 'i modulus 8' — da der kan

være otte enemies per lag. Hvis vi er på disse pladser, så fortsætter vi med at spawne den næste enemy uden at gøre yderligere. På denne måde får vi kun enemies i midten af vores play area.

5.2 StateMachine

Vores StateMachine bliver kaldt af vores Game objekt når spillet startes op. I vores StateMachine constructor sættes den aktive state til MainMenu således at spillet starter i vores menu. Vi har valgt at StateMachine selv er ansvarlig for at subscribe til de events den skal bruge da det til dels bliver mere overskueligt at det er StateMachine som subscriber til disse events, og til dels fordi vi mener at hver klasse så vidt muligt bør håndtere sig selv. Det kommer desværre til at gå ud over overskueligheden af hvilke events vores program benytter, men vi mener at det ovenstående opvejer dette.
Kommunikationen i vores program fungerer ved, at hver gang en af vores states i vores program ønsker at skifte til en anden state, så får vores StateMachine dette at vide, og ændre dernæst sin ActiveState til den nye state. I vores ProcessEvent metode har vi skrevet noget logik som skelner mellem om vores StateMachine får en InputEvent eller en GameStateEvent. Hvis vores event er en InputEvent, så sender StateMachine denne videre til den aktive state og eventen bliver dernæst overladt til HandleKeyEvent i den aktive state. Hvis StateMachine får en GameStateEvent så starter vi en switch case hvor vi tjekker for hvilken state ActiveState skal ændres til. I tilfældet hvor der ønskes at skiftes til GameRunning staten, gør vi et ekstra tjek for at finde ud af om det er MainMenu som skifter staten, eller om det er GamePaused som skifter staten. I tilfældet hvor det er MainMenu, da vil vi gerne starte spillet som helt nyt, og vi kalder derfor InitializeGameState for at starte helt forfra. Men hvis det er GamePaused som ønsker at kalde GameRunning, så vil vi gerne fortsætte fra det tidligere spil, og derfor kaldes InitializeState ikke. Vi gør dette tjek ved at kigge på om den nuværende ActiveState er MainMenu eller GamePaused inden vi skifter ActiveState.

6 Evaluering

Vi har gennem projektet ikke kunnet lave unit-tests da det vi har skullet teste har været af grafisk karakter og vi har derfor ikke kunnet bruge biblioteker som NUnit. Testsene har derfor bestået i at vi selv har kørt programmet og set om de nyeste ændringer fungerer som vi vil have dem. Eksempelvis har vi testet det at man taber hvis fjenderne rammer bunden. Testene har både bestået i at se om det virker hvis alle fjender er i live, eller hvis der kun er én fjende i live. Vi har også testet det at man vinder når man har dræbt alle fjender. Vi har i forløbet også testet om det korrekte state instansieres på det korrekte tidspunkt, dette har vi testet ved forskellige gennemgange gennem statesene. Vi har på den måde lavet en form for structured path coverage af de forskellige state-changes, udfra diagrammet i opgaveformuleringen side 3. I den nyeste version af koden har programmet besteået alle disse tests, vi vurderer derfor at koden fungerer som den skal.

7 Konklusion

Vi kan konkludere at den objekt-orienterede tilgang til et sådant program har været hensigtsmæssigt, da vi kan lave en intuitiv opdeling i de forskellige klassers ansvar. Det har derfor været naturligt at opfylde de objekt-orienterede design-principper beskrevet i analysen. Med hjælp fra disse principper er det lykkedes os at implementere de forskellige nødvendige klasser og interfaces på en struktureret måde, samt at holde koden læsbar. Denne stuktur har på den måde gjort det simplere at opnå de krav vi har stillet os selv i analysen.