Exercise 7

Software Development 2018

Department of Computer Science
University of Copenhagen

Emil Møller Hansen <ckb257@alumni.ku.dk>,
Casper Bresdahl <Caspers mail>,
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 Figure 1 ses et screenshot fra vore implementation af Galaga.

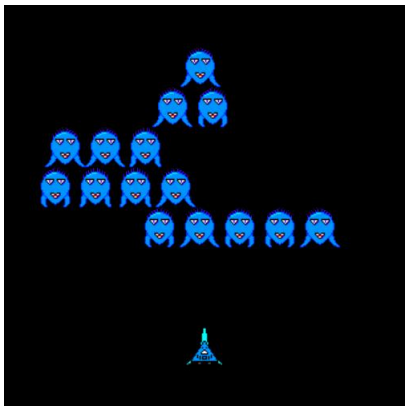

Figure 1: Den sidste implementation af Galaga

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

Dette projekt skal gøre brug af DIKUArcade idet der skal bruges mange grafiske elementer. Udover grafiske elementer skal vi bruge mange event-listeners og collision-detecting. Disse er også en del af DIKUArcade. Et spil skal bestå af følgende interaktioner:

1. Start-menu med 2 valg, start eller afslut
2. Der tjekkes for keyboard input, når der trykkes `enter` vælges den mulighed der nu er aktiv, trykkes der pil op eller ned, skifter den aktive valgmulighed.
3. Spillet starter og der tjekkes igen for input, denne gang for at rykke og skyde, på henholdsvis piletasterne og `spacebar`.
4. Når der er skudt skud af skal disse tjekkes for kollisioner med rumvæsnerne, hvis der rammes skal skuddet og det ramte rumvæsen fjernes.
5. Når man er i spillet skal man have mulighed for at sætte det på pause ved at trykke `ESC`, når man gør det skal der komme en menu hvor man har mulighed for at forsætte eller gå tilbage til start-menuen. Valget af dette fungerer ligesom på start-menu.
6. Når der ikke er flere rumvæsener tilbage skal der stå at man har vundet, og herfra skal man have mulighed for at gå tilbage til start-menuen. Hvis der er rumvæsener der når bunden har man tabt dette vil der ogås gives besked om, og herfra kan man også gå tilbage til start-menuen

Afhensyn til teknologi har vi de krav at koden skal skrives i C# da det er det sprog der bruges på kurset og det er det sprog DIKUArcade er skrevet i. Derudover vil vi bruge IDE'en JetBrains Rider og git til version control.
Vi vil skrive programmet objekt-orienteret da det er oplagt til sådan en opgave. Fordi vi har en masse koncepter fra virkeligheden, eksempelvis et skib der skal styres, som vi kan skrive ned som klasser. Dette indebærer at vi vil overholde forskellige objekt-orienterede design-principper. Det vil sige at vi eksempelvis vil sørge for at dele programmet op i forskellige units som hver er så selvstændige som muligt og hver har deres eget ansvar. Vi vil også sørge for at alle variabler og metoder har mindst muligt scope. Altsammen for at formindske kompleksiteten af den resulterende kode.

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. Da der ikke er blevet implementeret nogen måde for spilleren at dø (altså ved eksempelvis at støde ind i spilleren), så er den condition, at spillet tabes så snart en modstander rammer bunden af skærmen, blevet sat. 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.