DM575: Design og Implementering af Pac-Man

Celine S. Fussing
cefus23@student.sdu.dk

Sandra K. Johansen
sjoha23@student.sdu.dk

Sofie Løfberg
soloe23@student.sdu.dk

31. maj 2024

Contents

1 Introduktion

Det følgende projekt omhandler udviklingen og designet af spillet, Pac-Man. Projektet udearbejdes i kurset DM575: Objektorienteret programmering med henblik på at kunne anvende objektorienteret principper og designvalg. Projektet implementeres ved brug af Java samt JavaFX. I projektet uddybes der relevante design- og implementeringsvalg.

1.1 Antagelser

Dette projekt implementerer de krav, som er blevet stillet til projektet. Nedenfor vil der dog præciseres ekstra antagelser, som er gjort i vores version af spillet. Først og fremmest har vi valgt, at når spillet er vundet, er der ikke nye levels. Derimod kan brugeren udelukkende vælge, hvorvidt de vil spille igen eller helt afslutte spillet. Herudover har vi valgt, at når Pac-Man støder ind i spøgelser, mens han er i normaltilstand, vil han miste et liv, men også point. Således har brugeren mulighed for at opnå forskellig score ved spillets afslutning. I modsætning til det oprindelige Pac-Man, må vores spøgelser skifte retning til den, de kom fra.

2 Proces

Et af de essentielle værktøjer brugt i forbindelse med projektet, for at sikre et godt samarbejde, har været udvidelsesprogrammet: Live Share fra Microsoft. Denne udvidelse til VS Code har sikret, at alle på lige fod har kunne være en del af projektet. Live Share tilbyder ligesom programmer som Google Docs og Microsoft Word Online mulighed for, at flere kan skrive samtidig. Således har alle haft mulighed for at se koden, mens der blev overvejet eventuelle implementeringer eller problemer. Det førnævnte har også sikret, at ikke kun én har stået for udarbejdelsen af koden til projektet. Til selve kunsten er programmet Clip Studio Paint blevet brugt. Programmet tillader at tegne digitalt og derved at kunne skabe de sprites, vi senere præsenterer og bruger.

Herudover har en central del af vores samarbejdsproces også bestået i at være meget bevidste omkring den enkeltes styrker og svagheder. Eksempelvis omfatter gruppen en, som er rigtig dygtig til det kreative og dermed også har syntes, at animationsdelen har været spændende. Samtidig inkluderer gruppen også en, som er skarp til at se overgangen fra design til implementering. Gruppens sidste medlem har været central i sparringsprocessen og udviklingen af nye ideer og alternative tankegange. Derfor komplementerer vi hinanden godt i gruppen, hvilket har fremmet vores samarbejde.

3 Æstetik og Tema

Dette afsnit omhandler det generelle udseende af spillet, såvel som tematiske valg der blev taget i designfasen. I afsnit 4 præsenteres overvejelser vedrørende design, bl.a. hvilke klasser vi ville have, samt hvordan de skulle hænge sammen.

3.1 Tema

Projektbeskrivelsen satte generelle regler for, hvad vores spil skulle indeholde. Disse regler har vi umiddelbart holdt os til, dog er vi gået ud over, hvad der blev beskrevet som "minimale krav" æstetisk. Herunder har vi valgt et tema - nemlig gamle England - mere specifikt Camelot. Derved havde vi et basis for vores æstetik.

3.1.1 DragonMan

Med vores tema kunne vi ikke have den klassiske, elskede, gule kugle. Han var for moderne til vores Camelot. I mangel på en helt til at være hovedfigur i vores spil, fandt vi i stedet på antihelten, dragen. Han blev navngivet DragonMan til ære for Pac-Man; så skulle han blot have en livsmission. Dette blev at samle guld ind til sin skattebunke, da drager gør to ting; Samle skat til sin bunke, og brænde ting ned. Det førte ret naturligt til en titel til vores nu vagt Pac-Man-lignende spil: "Camelot's Burning!"

3.1.2 Knights

Endnu engang stødte vi ind i, at spøgelser - dog nok til stede i magiske lande - ikke passede særlig godt ind med Camelot som tema. Vores kære DragonMan skulle have nogle fjender. Dette ledte os direkte til riddere; en drages naturlige fjende. Meget som i Pac-Man skulle vi bruge fire riddere for at symbolisere Clyde, Blinky, Pinky og Inky. Herunder blev vores riddere skabt: Blue, Purple, Pink og Orange. Vi indser, der kunne have været mere kreativitet ved navnene, men riddere, her skurken, fortjener ikke nødvendigvis at kunne få sympati ved navngivning.

3.1.3 Camelot

Et hvert godt spil, herunder Pac-Man, har et bræt, hvor selve handlingen foregår. Den kliniske verden, Pac-Man lever i, kunne heller ikke passe ind i det nu næsten magiske Camelot. Der manglede liv. Vi udsmykkede verdenen med murstensvægge, udkigstårne, smalle gader og blomster. Den vigtigste ændring bestod i at lave guldmønter, som DragonMan kunne samle ind. Vi valgte ligeså at lave de fire power-pellets til ildkugler, som igen passede bedre til vores tema. Så manglede vi kun, at få alle ideerne repræsenteret visuelt på en måde, som hang sammen.

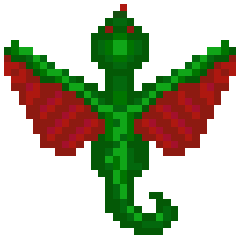
3.2 Æstetik

Til ære for gamle arkade spil blev pixel-kunst en nødvendighed. Alle illustrationer der skulle bruges fik en konsekvent størrelse, her 32x32 pixels, så det let kunne sammensættes. Dette gjorde det også lettere at skabe selve kunsten, der hørte til - altså sprites. Vi valgte også, at prioritere animationen af sprites.

Selve metoden til animationen af sprites gøres endvidere klart i afsnit 4. De brugte sprites dertil præsenteres herunder.

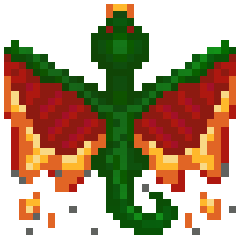
3.2.1 DragonMan

Fra projektbeskrivelsen kan vi se, at vi må have to drager; en normal, og en til når spillet er i powermode. Herunder gav det æstetisk mening, at vores power-pellet-erstatning, ildkugler, gav vores DragonMan evnen til at spy ild. Dette skulle tydeliggøres ved hans sprite.
Nedenfor har vi DragonMan i hans to versioner.



Figur 1: Oprindelige DragonMansprite, pegende opad, i normal-tilstand.

Figur 2: Alle DragonMan sprites i normal-tilstand, brugt til animation.



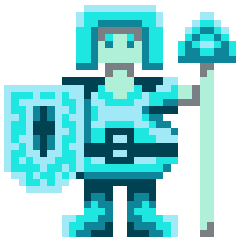
Figur 3: Oprindelige DragonMan sprite, pegende opad, i power-tilstand.

Figur 4: Alle DragonMan sprites i power-tilstand.

3.2.2 Knights

Herunder skulle der flere sprites til de fire forskellige riddere, såvel som en repræsentation af dem i power tilstand. De fire riddere fik blot forskellige farver, og det samme gyldne udseende i power tilstand, for at gøre det lettere at identificere, når tilstanden foregår. De omtalte sprites ses herunder.

Først har vi ridderne i normal tilstand. Eksempelvis viser den blå ridder standardformen på alle farver af riddere - efterfulgt af alle sprites for farverne på række. Derefter ses powerstilstand:



Figur 5: Oprindelige blå ridder sprite, pegende opad, i normal-tilstand.

Figur 6: Blå ridder sprites.

Figur 7: Lilla ridder sprites.

Figur 8: Pink ridder sprites.

Figur 9: Orange ridder sprites.

Figur 10: Power-tilstand ridder sprites.

3.2.3 Camelot

Til vores mytiske land, Camelot, skulle der bruges de førnævnte vægge, tårne, gader osv. De skulle også repræsentere både normal og power tilstand. Herunder en mere dunkel farve og ild i tårnene for at repræsentere dragens handling: At brænde byen ned og dræbe dens riddere. De felter - dvs. tiles - der bruges vises herunder i rækkefølgen: Jord, tårn, væg, ildkugle, mønt, ridderborgens jord. Dette gøres af to omgange, nemlig normal tilstand og derefter power tilstand:

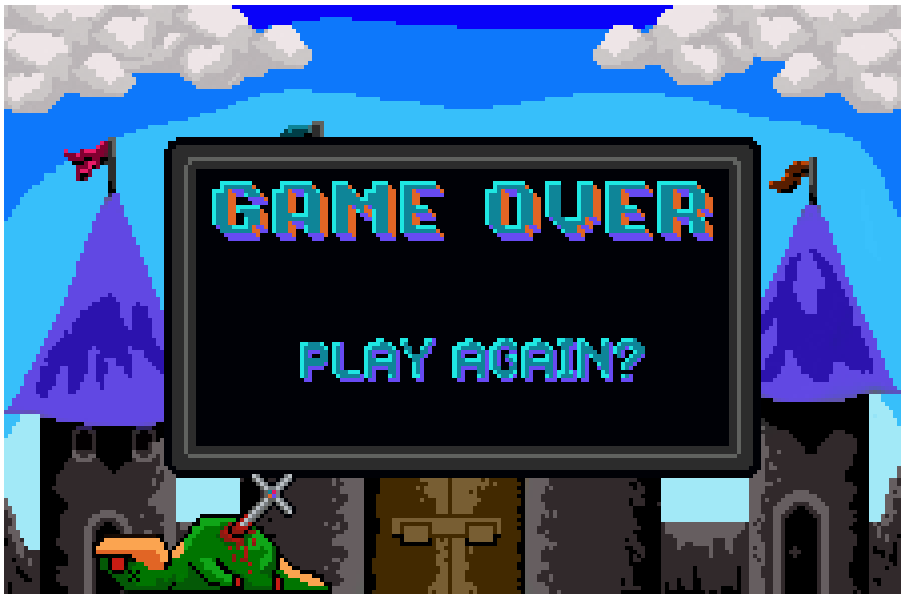
Figur 11: Tiles i normal tilstand.

Figur 12: Tiles i power tilstand.

Der kan her ses, at felterne i power-tilstand er markant mere dunkle og brændende. Mere specifikt er tårnene i brand, blomsterne er visnede, og mønterne bærer nu dragens ansigt som hån mod de bange riddere.

3.2.4 GameOver

Under planlægningen af vores tema var slutskærmen ikke umiddelbart en del af vores overvejelser. Dog indså vi, at man kunne være mere kreativ med sluttilstanden end projektbeskrivelsen lagde op til. Derfor lavede vi to forskellige skærme til, om man henholdsvis vinder eller taber. Begge viser scoren og holder sig desuden til temaet. De ses herunder:



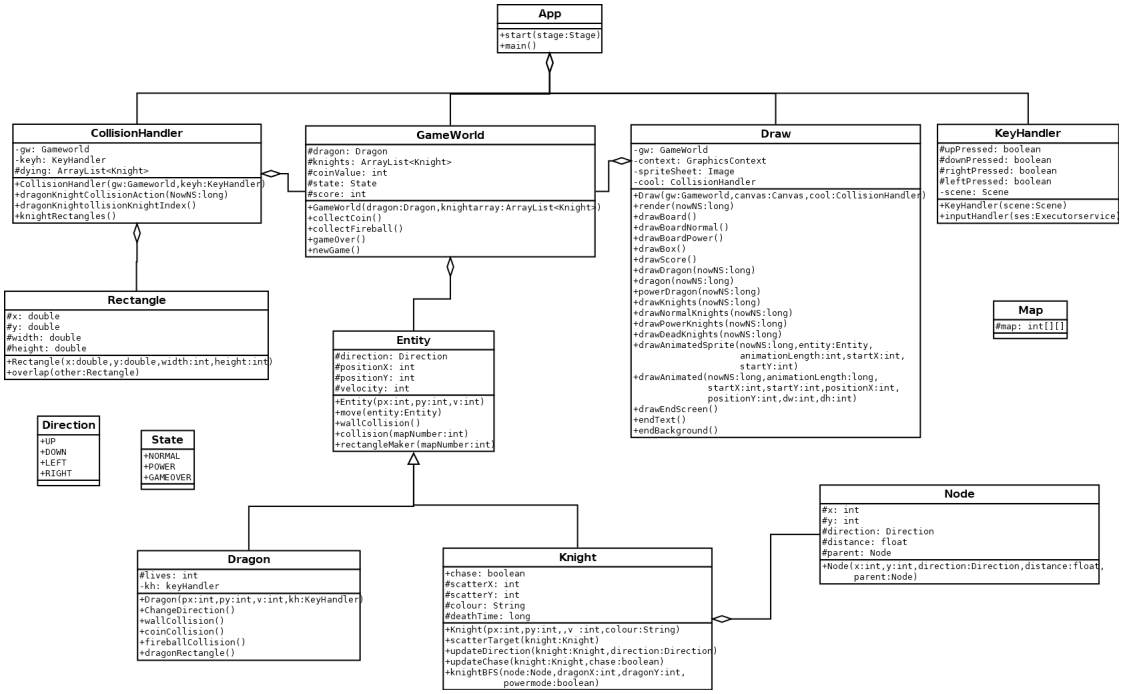
Figur 13: Skærmen vist, hvis man mister alle sine liv.

Figur 14: Skærmen vist, hvis man samler alle mønter.

Som en sidste kommentar vil vi gerne pointere, at alt kunst relateret til spillet er håndtegnet via Clip Studio Paint. Derved har vi rettighederne til dets brug i vores programmeringsprojekt. Alt det for at sige, vi har intet taget fra internettet udover inspiration og skal derved ikke bruge referencer for vores kunst.

4 Design og Implementering

4.1 Klasser og objekter



Figur 15: UML diagram

Afleveret separat er vores UML diagram i større version - navngivet "G15 UML.pdf".

4.1.1 App

App fungerer som vores forbindelse mellem andre klasser. Den indeholder metoden 'start', hvor hele spillet køres fra. I start initialiseres alle de relevante objekter såsom gameworld, draw, collisionhandler, keyhandler og entities. Her passeres objekter til de relevante klasser. Kommunikationen mellem klasser foregår derfor i App. App indeholder også vores gameloop, som for hver frame kalder de metoder, som bruges til at køre spillet. Det er for eksempel metoderne i Draw, som tegner et opdateret bræt. Ligeså metoderne fra Gameworld, CollisionHandler og Entity, som styrer spillogikken.

4.1.2 Entity

Entity er en klasse, som både dragon og knight nedarver fra. Den definerer derfor opførsel og attributter, som de to objekter har tilfælles. Det er for eksempel, at begge skal bevæge sig kontinuerligt og håndtere kollisioner med vægge.

4.1.3 Dragon

Pac-Man er i vores version blevet erstattet af vores Dragon-objekt. Dragon får det meste af sin opførsel fra Entity. Størstedelen af klassen er derfor, hvordan dragen opdaterer retning, da denne adfærd er unik. Dragon skal opdatere retning baseret på input fra tastaturet, hvilket er implementeret i KeyHandler-klassen. Dragon har derfor et keyhandler-objekt som en attribut. Vi har således simpelt kunne lave en metode til at opdatere dragens retning ved hjælp af de boolske variabler defineret i Keyhandler.

4.1.4 Knight

I vores version af Pac-Man har vi ændret spøgelserne til riddere for at passe ind med vores tema. Knight nedarver ligesom Dragon fra Entity og får delvist attributter og opførsel derfra. Udover det har objektet knight nogle unikke attributter såsom en boolean 'chase', som fortæller om knight er i chase- eller scatter-mode. I forbindelse med det, har knight også 'targetX' og 'targetY', som bruges i scatter-mode.

Den store forskel mellem knight og dragon er, at knight selv skal finde ud af, hvor den skal hen. De skal gerne forsøge at fange dragon uden at være alt for svære at undgå. Det er også i den forbindelse 'chase' og 'scatter' er relevant. knight benytter breadth first search til at finde den rute på kortet, som leder dem til et specifikt punkt. Når attributten 'chase' er sand, vil det punkt være dragon's position. Et tilfældigt punkt på kortet bruges i stedet, når 'chase' er falsk. I power tilstand skal knight flygte fra dragon, og til det formål bruges søgealgoritmen ligeså. Der søges dog i stedet for et punkt, der er en vis afstand væk fra dragon.

4.1.5 Gameworld

Gameworld indeholder store dele af spillogikken samt alle dele af spillet, som skal passeres til eksempelvis Draw-klassen. Gameworld indeholder også metoderne 'gameOver' og 'newGame', som henholdsvist tjekker, om spillet er færdigt og sætter spillet tilbage til startindstillingerne.

4.1.6 Draw

Draw har ansvaret for at tegne vores bræt, så spilleren kan se spillet. Dette gøres ved at passere et gameworld-objekt til Draw via App. Draw kan så bruge dataen i gameworld samt klassen Map til at skabe den visuelle repræsentation. I Draw bruges JavaFX. Dette er nødvendigt til at tegne de ønskede sprites, da JavaFX har mange metoder til at udskære og tegne billeder. Vi bruger disse i vores egne metoder, som minimerer antallet af input og henter de korrekte billeder i korrekt rækkefølge. Således kan vi ved brug af vores 'gameloop' animere vores sprites som ønsket.

4.1.7 KeyHandler

KeyHandler har ansvaret for at modtage input fra tastaturet, så spilleren kan styre dragon. KeyHandler har derfor fire boolske variabler, som repræsentere de fire retninger. Vi laver en EventHandler, som håndterer KeyEvents. Eksempelvis bliver den boolske variabel, som repræsenterer retningen op, sat til sand, når der bliver trykket på pil op. dragon-objektet benytter disse variabler. I vores implementering af spillet har vi valgt at bruge 'enter' for at starte et nyt spil, og bogstavet 'q' for at lukke spillet ned. Det håndteres ligeså i KeyHandler.

4.1.8 CollisionHandler

CollisionHandler-klassen har udelukkende ansvaret for kollisioner mellem dragon og knight. Klassen har derfor en attribut med gameworld-objektet, så den har adgang til de aktuelle positioner. Kollisionerne håndteres ved at bruge klassen Rectangle. Rectangle har en størrelse og en position som attributter og indeholder også metoden 'overlap'. I CollisionHandler kan der så laves rektangler for listen af knights, og metoden 'overlap' bruges til at tjekke, om de rektangler kolliderer med dragon's rektangel.

CollisionHandler indeholder også metoden 'collisionAction', som sørger for effekterne af en kollision mellem dragon og knight. Til det formål har et keyhandler-objekt også været nødvendigt i CollisionHandler. De boolske variabler i KeyHandler bruges til at sikre, at spillet først starter, når brugeren trykker på en piletast. Dette skal også gælde, når dragon støder ind i en knight og mister et liv. Derfor skal 'collisionAction' i CollisionHandler ændre på variablerne, som bruges til at skabe denne forsinkelse.

4.2 Objektorienteret Design

I udviklingen af vores design har vi gjort flere overvejelser for at holde os til de objektorienterede principper. I forhold til OO-principperne har vi kigget på for eksempel princip 1: Indkapsling, som har været i fokus i forbindelse med dannelsen af vores objekter. Når vores objekter har attributter, som udelukkende bruges i egen klasse, er de lavet private. Attributter, som bruges af andre klasser, såsom de boolske variabler i KeyHandler og vores map, er beskyttede, så de kun kan bruges af filer i samme package.

I forbindelse med de tre klasser Entity, Dragon og Knight har vi også haft fat i OO-princip 3: Nedarvning. Dragon og Knight nedarver fra Entity, da de har opførsel tilfælles. Ligeså har vi haft Liskov Substitution Principle (SOLID-princip) i fokus her. Dette overholder vi siden, at vi kan bruge metoderne i Entity til både knight-objekter og dragon-objekter. Det kan ses i App, som bruger 'move' på begge typer, og at dette virker gnidningsfrit. Dette kan også knyttes til subtypepolymorfi.

4.3 Designmønstre

Vi benytter App som controller, Draw som view og Gameworld som model. Dette er opførselsmønstret MVC eller Model-View-Controller. I vores implementering fungerer det således, at Model/gameworld indeholder vores data og metoder til databehandling. Dette inkluderer vores entities, som også ligger i gameworld-objektet og tilgås via det. View/Draw er en visuel repræsentation af dataen i model og tilgås via aggregering. App er vores controller, men lidt mere nuanceret er den controlleren i sammenhæng med KeyHandler, siden det er KeyHandler, som modtager input fra brugeren. App virker dog som ledet mellem Gameworld og Draw. Den transporterer aktuel data fra Gameworld til Draw, så Draw kan vise det til brugeren.

5 Diskussion

I det følgende afsnit beskrives overvejelser angående mulige implementeringer samt eventuelle forbedringer.

5.1 Hastighed på entities

Idet man kører vores spil, kan man opleve tre scenarier: Enten at DragonMan og ridderne bevæger sig langsomt, at de bevæger sig tilpas hurtigt eller at de bevæger sig meget hurtigt. Dette hænger sammen med, hvordan vi har valgt at implementere 'hastighed' i projektet. Som beskrevet i afsnittet 'Design og Implementering' har hver af vores entities en attribut, velocity. Den definerer i Dragon- og Knight-klassen, hvor mange pixels de skal bevæge sig per frame. Dette medfører dog, at den ikke tager højde for den egentlige spilletid for den enkelte computer, der er passeret siden den sidste frame. Derfor vil man - alt efter hvilken computer der spilles på - opleve en forskel på DragonMan og riddernes hastighed.

En løsning på dette problem kunne være at bruge 'delta time'. Delta time bruges nemlig til at gå fra pixels per frame til pixels per sekund. Dette ville sikre en uniform brugeroplevelse på tværs af computere. Der er flere måder at bruge og anvende delta time - også alt efter hvilken præcisionsgrad man vil have. Vi vil dog nok have gået med en af de mere simple anvendelser, som stadig løser problemet. For implementering af dette, ville vi skulle udregne tiden for gennemløb af hver udførsel af gameloop. Desværre nåede vi ikke videre end idéfasen, hvorfor ovenstående ikke blev indført.

5.2 AI

Nedenfor uddybes de tanker og overvejelser vi gjorde angående vores ridders AI. Først og fremmest inkluderer det vores oprindelig AI samt udfordringen

med den. Vi diskuterer også eventuelle forbedringer, man kunne lave til den nuværende AI.

5.2.1 Oprindelig AI

Oprindeligt implementerede vi riddernes AI ultra simpelt. Dette blev gjort ved, at ridderne vurderede den korteste afstand til DragonMan og den dertilhørende retning. Vi bestemte deres fremtidige positioner, hvis de fulgte en hypotetisk retning. Disse teoretiske positioner blev brugt til sammenligne distancer til DragonMan ved brug af Manhattan-distance. Afslutningsvist tilføjede vi et element af random for også at tilføje lidt forskellig karakter til ridderne, så de ikke klumpede sammen.

Implementeringen virkede, men der viste sig at være nogle ret store udfordringer med den. Først og fremmest bestemte vi Manhattan-distancen i fugleflugt, som selvfølgelig ikke tog højde for vægge. Det betød at for hver frame valgte ridderne den retning med den korteste distance til dragen, ligegyldigt om en væg var i vejen eller ej. Dette betød, at ridderne ville blive fanget bag vægge. Selv med det random element inkluderet, ville ridderne ved næste frame vælge den samme retning igen. Det resulterede ofte i, at ridderne ville gå frem og tilbage samme sted. Konklusionen på denne implementering blev dog hurtigt, at de stadig forblev for uintelligente til, at man kunne få et interessant spil. Dette førte os til at tænke i Depth-First Search, A* Search og Breadth-First Search.

Vi overvejede først Depth-First Search, men den ville udvide en hel gren uden at vide, om det ville give den bedste løsning. Vi overvejede derfor en informeret søgealgoritme såsom A* Search. A* ville have været en fornuftig løsning, dog undersøgte vi også brugen af Breadth-First Search og konkluderede, at vi nemmere kunne implementere den.

5.2.2 Forbedringer til nuværende AI

I vores AI ville vi også gerne have nået at give ridderne forskellige personligheder, så de havde mere karakteristisk opførsel. Dette kunne vi have gjort på forskellige måder, hvis vi havde haft tid til det. Eksempelvis kunne det været fedt, hvis de brugte forskellige søgealgoritmer til at bestemme deres næste retning til DragonMan. Omvendt kunne man også have tilføjet en begrænsning til den eksisterende søgealgoritme, som betød, at en vis ridder kun måtte nærme sig dragen fra en bestemt retning. En anden løsning kunne være at tildele hvert spøgelse en kvadrant, som de søger tilbage til i scatter-mode. Dette tænker vi kunne have gjort spillet både sværere, men også mere interessant.

5.3 State - designmønster

Designmønsteret, state, kunne have været relevant i vores implementering af projektet. Flere steder tjekker vi hvilket state, vi er i, ved brug af if-statements. Dette kunne have været mindsket ved brug af dette designmønster. Vi havde dog svært ved den reelle implementering både på grund af evner og tid og endte med en mere simple metode.

5.4 Glidende retningskift for DragonMan

En ulempe ved vores nuværende spil er, at brugeren skal være ret præcis, når de styrer DragonMan for at få drejet. Dette skyldes, at vi definerer alle vores entities, herunder DragonMan, som rektangler, og runder alle entity-positioner til at være centreret på en tile. Derfor hvis selv enkelte pixels i rektanglet har passeret stien, hopper dragen i stedet hen på næste eller forrige felt. Det kunne have været dejligt, hvis vi kunne have udbedret det, så den ikke var nær så sensitiv og mere glidende i sine retningsskift.

5.5 DRY

Vores projekt har desværre i visse klasser en tendens til ikke at følge DRY-princippet helt. Dette betyder således, at der er gange, hvor vi får gentaget stykker af kode. Vi ville gerne have nået at ændre det, så vi i større grad kunne følge DRY-princippet og gøre koden nemmere at vedligeholde. Det havde vi desværre ikke mulighed for på grund af tidsmangel. Dét, vi kunne have gjort, var at tilføje flere overordnede metoder eller hjælpemetoder til at få mindsket antallet af gentagelser.

6 Konklusion

I løbet af projektet har vi implementeret de simpleste Pac-Man regler. Således har vi et spil, som kører og samtidig opfylder opgavekravene. I projektet har vi særligt lagt vægt på tema og æstetik samt objektorienteret principper såsom indkapsling og Liskov Substitution Principle. Dog er der også svagheder ved vores implementering. Vi diskuterer i rapporten også eventuelle forbedringer, vi kunne have lavet, havde vi haft mere tid. Dette markerer afslutningen på vores projekt.