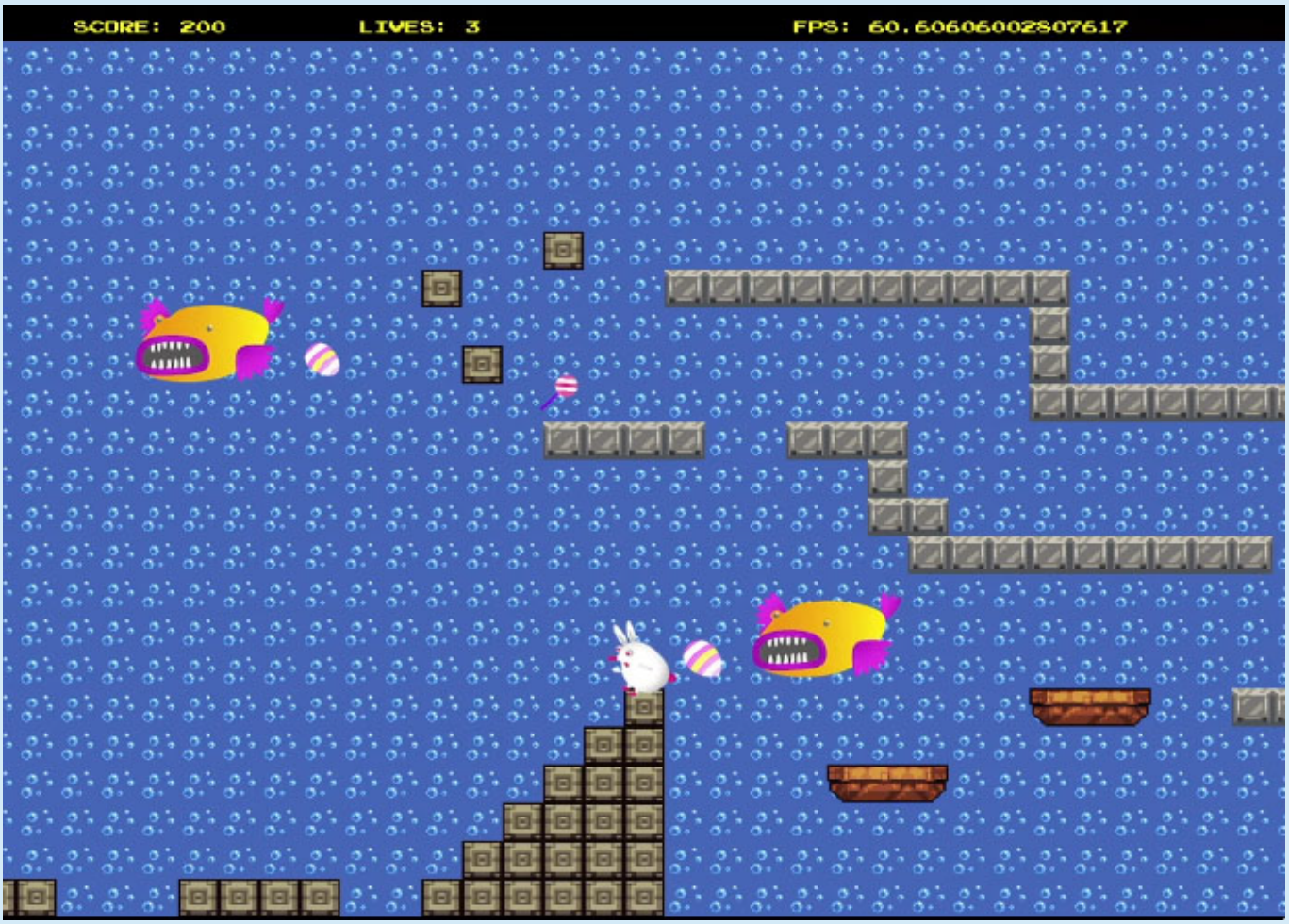
Spiludvikling i Python med Pygame

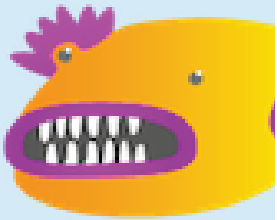
2D Platformspil til Raspberry Pi 3 drevet arkademaskine



Skrevet af: Alberte Jeberg Kjær (58096), Leonora Bryndum (57944), Kevin Martin Lindemark Holm (57939), Jakob Scheunemann Hastrup (57318)

Vejleder: Ebbe Vang
Fag: Datalogi
RUC, 2017, 4. Semester

Tak til:
FabLab
Ebbe Vang
Felicia Österlin



Indholdsfortegnelse

Abstract

The purpose of this study, is to explore how to develop a 2D-arcade platform game for a Raspberry Pi 3 computer. This will be done with the programming language Python 3 and the game development framework Pygame. To do this, we investigate and learn about the Python 3 language and how this is structured. We investigate what an arcade game usually contains in the terms of programmatic content, and choose the focus points: Game Loop, Collisions, Physics, Animation, Level Design, Camera, Heads Up Display and Screens, to explore further. We follow tutorials, use Pygame documentation and experiment to develop the game: *White Rabbit*. We conduct tests to find out how the game performs on different platforms and under different circumstances. We will further discuss the experience we get from the tests, from working with Python and Pygame, and the challenge of making a scalable game.
We discuss further development that could improve the performance of the game.
We find that it is important to consider what platform a game is intended for early in the process, and the necessity to do frequent tests on this platform while developing. We also conclude on the game development process, Python as a language, and on which features are important, when programming a game.

Indledning

Problemfelt

Dette projekt omhandler læringsprocessen omkring det at lære at programmere et 2D-spil. Projektet udspringer af gruppens personlige ønske, om at lave vores egen arkademaskine. Dette kræver både at bygge maskinen, og programmere spillet, der skal køre på den. Da dette er et semesterprojekt i faget Datalogi, beskæftiger projektet sig hovedsageligt med det at lære programmeringssproget Python, som vi har valgt at skrive spillet i, samt at lære hvilke elementer, der er nødvendige for et spil. Til dette bruges frameworket Pygame, og det er derfor også vigtigt at lære at forstå at bruge dette.

Vi har begrænset os ved, at vi gerne ville lave et 2D platformspil til en arkademaskine. Vores krav til denne maskine har været, at den skal kunne køre vores eget spil, på en autentisk måde, inspireret af gamle bartop arkademaskiner (figur 1).

Figur 1: Bartop Arcade Machine

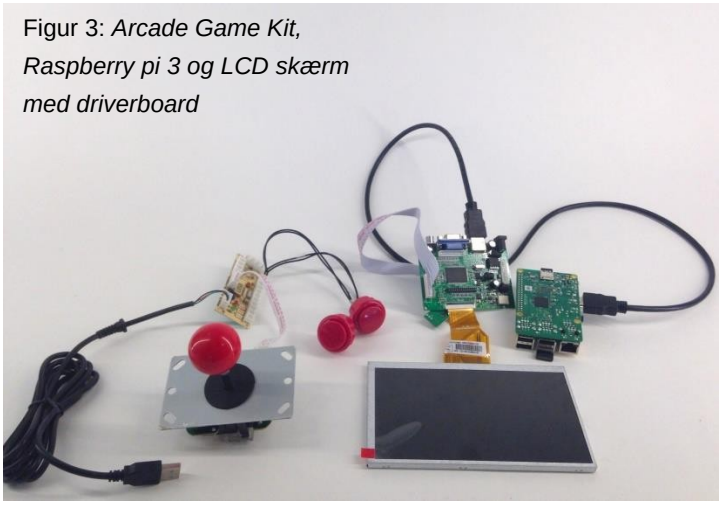
White Rabbit

Dette afsnit vil kort præsentere *White Rabbit*, som er det arkadespil, vi har udviklet. Dette er for at give læseren en grundlæggende forståelse og indsigt i det færdige produkt i forhold til opgavens indhold.
White Rabbit er et arkadespil, som er udviklet til at kunne køre på en Raspberry Pi 3. Hovedkarakteren i platformspillet er en hvid kanin (White Rabbit), som skal gennemføre hvert level ved at nå hen og spise en svamp. I hvert level kan *White Rabbit* møde de flyvende fisk, som er farlige, og kaninen kan derfor dræbe disse, ved at skyde påskeæg ud af numsen. I hvert level vil White Rabbit ligeledes kunne finde karameller, som kan spises, og som giver point, der tilføjes en score (figur 2). I bilag 1 findes et link til en YouTube-video hvor spillet afvikles. I bilag 2 kan man se hele den færdige programkode.

Figur 2: *White Rabbit* spillet

Arkademaskinen

For at gøre oplevelsen autentisk, ville vi gerne have joystick og knapper, der skulle minde så meget som muligt, om dem på de gamle arkademaskiner. Efter at have undersøgt, hvordan man kunne gøre dette, fandt vi ud af, at man kunne købe et billigt Arcade Game kit (figur 3).



Figur 3: *Arcade Game Kit, Raspberry pi 3 og LCD skærm med driverboard*

Dette kit bestod af knapper, joystik og en USB-encoder, som ville gøre det muligt, at sørge for at inputtet fra knapperne og joystick ville blive encoded, så de svarede til normale keyboard-knapper. På den måde ville knapperne kunne gøres mere kompatible med normale computere.
Vi fandt ud af at single-board computeren, Raspberry Pi, ville være en oplagt platform til dette formål. En Raspberry Pi er ikke så dyr og derved ville vi kunne holde os inden for projektets budget. Platformen matcher ikke en normal computers processorkraft og grafikkort, men i konteksten af at vi ville udvikle et 2D arkadespil, ville denne single-board computers hardware specifikationer formentlig være tilstrækkelig. Desuden, er en Raspberry Pi lille, og kræver ikke så meget strøm. Begge dele er fordele, hvis den skal integreres i en arkademaskine, som eventuelt skal være batteridrevet eller stå tændt i længere perioder (se Bilag 3 for specifikationer vedr. Raspberry Pi 3).

2D-Platformspil

Efter at vi havde begrænset os omkring platformspil, begyndte vi at undersøge, hvilket framework (Library til spiludvikling), vi ville arbejde med for at udvikle *White Rabbit*. Det skulle være kompatibelt med en Raspberry Pi, men det behøvede kun at understøtte 2D-spil. Eftersom vi havde erfaring med at programmere i Java, virkede dette som den logiske vej at gå. Efter lidt søgen på nettet, fandt vi frem til et Java framework kaldet Libgdx, som var godt dokumenteret. Det viste sig dog efterfølgende, at dette library som standard ikke var understøttet af Raspberry Pi. Vi fandt derefter frameworket Monogame, som er skrevet til C#, som havde den fordel at sproget minder om Java. Dette viste sig dog heller ikke at være understøttet af Raspberry Pi. Derfor endte vi med at vælge frameworket Pygame, der blev anbefalet til spiludvikling på Raspberry Pi.

Dette framework var ikke baseret på Java, men derimod Python. Vi undersøgte lidt om Python, som skulle være et forholdsvist let sprog at lære. Vi besluttede os for at undersøge sprogets syntaks igennem nogle tutorials. Umiddelbart virkede sproget og dets syntaks forståeligt for os, og efterfølgende kunne vi teste Pygame på en Raspberry Pi, som et *proof of concept*. Da vi forsøgte at køre et spil, der havde samme kompleksitet som vi ville opnå, og så at det fungerede, besluttede vi os for at arbejde i Python med Pygame som framework.

En læringsproces

Fordi vi førhen har arbejdet objektorienteret i Java, ønskede vi at strukturere vores programkode på en overskuelig og genkendelig måde. Ingen af gruppens medlemmer havde dybdegående erfaring inden for programmering af spil, og vi skulle derfor finde ud af, hvilken type elementer, og hvilken programstruktur et spilprogram typisk skulle indeholde. Parallelt med dette, arbejdede vi alle med et nyt sprog, som således først krævede, at vi fik en grundlæggende forståelse og færdigheder indenfor. Dette projekt kan således betragtes, som en undersøgelse og en læringsproces i konteksten af at udvikle et spil.

Problemformulering

Da vores projekt kan anskues som en læringsproces omkring at lære programmeringssproget Python, samt at lære at udvikle et simpelt platformspil med frameworket Pygame, der kan køre på en Raspberry Pi, lyder problemformuleringen således:

Hvordan kan man udvikle et 2D-platformspil til Raspberry Pi i Python med frameworket Pygame?

Problemstillinger

For at besvare ovenstående problemformulering bedst muligt, har vi opstillet følgende problemstillinger:

Hvad er Python, og hvordan er det at benytte og lære som programmeringssprog?
Hvad er Pygame, og hvordan bidrager det til udviklingen af et spil?
Hvordan laves objektorienteret programmering i Python, og hvilke fordele har det i konteksten af spiludvikling?
Hvilke elementer er vigtige i 2D-platformspil, og hvordan kan disse implementeres med Python og Pygame?
Hvordan kan man teste performance i et spilprogram, og hvorfor er dette relevant?
Hvordan kan man tænke skalérbarhed og videreudvikling ind i spillet *White Rabbit*?

Fokuspunkter

Meget af vores viden omkring hvad et spil skal indeholde af programkode, objekter og metoder, er kommet ved at følge spil-programmerings-tutorials. Flere forskellige tutorials (fra fx KidsCanCode.org eller Pygame (Python Game Development) - Youtube), introducerer til, hvordan man kan spiludvikle med frameworket Pygame.
På baggrund af undersøgelserne nævnt i problemfeltet og vha. tutorials, kom vi frem til nogle fokuspunkter vi anså som vigtige i forhold til spiludvikling, som vi vil undersøge nærmere igennem dette projekt.

For at håndtere den kompleksitet, der kan opstå, når man programmerer spil, har vi fundet det vigtigt at undersøge, hvordan man kan strukturere sit program på en smart

måde, så man ikke taber overblikket. Ligeledes fandt vi frem til nogle elementer, der var særligt vigtige i et spil. Vi fandt ud af, at næsten alle spil skal have et gameloop, for at fungere optimalt. Vi fandt at physics særligt er vigtigt i arkadespil, når man skal lave et underholdende gameplay. Vi anså kollisioner, som et vigtigt element, da gameplay i klassiske 2D-arkadespil nærmest ikke ville kunne fungere uden spilobjekter, der kolliderer med hinanden. Vi anså animation, som vigtigt i spil, da det er med til at gøre spiloplevelsen mere levende. Vi anså HUD (Head Up Display), som relevant da det er med til at give, den der spiller, et bedre overblik over bl.a. score, liv og lignende udviklinger, der måtte ske i løbet af et spil.
Således vil opgaven kronologisk præsentere nedenstående fokuspunkter.

Sprog og struktur
I dette afsnit vil vi beskrive programmeringssproget, Python ud fra Pythons egne principper omkring design. Ligeledes vil afsnittet forsøge at reflektere over væsentlige forskelle mellem Python og Java. Til sidst vil det præsentere frameworket, Pygame og objektorienteret programmering i Python.

Game Loop
I dette afsnit belyses, hvad et game loop er, samt hvordan det kan implementeres, og hvad vi har lært.

Physics
I denne sektion vil vi komme ind på, hvad physics er, og hvilken rolle det har i spiludvikling, og de erfaringer vi har tilegnet os.

Collisions
I dette afsnit vil vi komme ind på, hvad kollisioner er, og hvilken betydning, de har når man udvikler spil.

Animation
Her vil vi komme ind på hvordan animationer kan anvendes til at skabe et mere interessant gameplay, og hvad vi har lært ved implementeringen.

HUD og screens
I dette afsnit vil vi belyse hvad HUD og screens er, og hvordan det kan implementeres i et spil.

Sprog og struktur

Python bygger på programmeringsparadigmet struktureret programmering. Struktureret programmering prøver at forbedre tydeligheden, kvaliteten og udviklingstiden i computerprogrammer. Dette afsnit handler om Pythons filosofi og forskelle mellem Python og Java. Derudover vil vi forklare, hvad Pygame er for et framework og til sidst vil vi beskrive objekt-orienteret programmering, som udspringer af struktureret programmering og er den tilgang vi primært har fulgt.

Python

Følgende afsnit præsenterer programmeringssproget Python. Vi vil her kort redegøre for, hvordan Python opstod, hvordan det fungerer, og hvordan det adskiller sig fra andre programmeringssprog. Vi er i gruppen allerede bekendte med Java, og derfor vil følgende afsnit hovedsageligt redegøre for de væsentlige forskelle mellem Python og Java, og vores erfaringer omkring at lære et nyt programmeringssprog.

Programmeringssproget, Python, er lavet af Guido Van Rossum, en hollandsk programmør. Det blev udtænkt i løbet af 80'erne, og implementeringen blev startet i 1989 (Python (Programming Language) - Wikipedia, 2017). Det blev desuden videreudviklet under hans senere ansættelse hos Google (Guido van Rossum - Wikipedia, 2017). Python blev udviklet til at være læsbart, og skulle give mulighed for, at bruge færre linjer kode til en metode end fx i C og Java. Sproget er Open Source og kan benyttes af alle, så længe man overholder Pythons copyright-betingelser. Det er desuden designet til nemt at kunne udvides og køres på langt de fleste platforme (General Python FAQ — Python 3.6.1 Documentation, 2017).

Pythons designprincipper

Guido Van Rossum har skrevet et blogindlæg —Python's Design Philosophy‖ (2009) om hans grundtanker bag Python. Han startede Python som et fritidsprojekt uden et budget, og da han gerne ville have resultater hurtigt opstillede han en række guidelines for projektet. Et udpluk af reglerne lød:

- *Borrow ideas from elsewhere whenever it makes sense.*
- *Don't fret too much about performance--plan to optimize later when needed.*
- *Don't try for perfection because "good enough" is often just that.*
- (Rossum, 2009)

Man kan tydeligt se disse regler i sproget, da langt de fleste funktionaliteter findes i andre programmeringssprog. Således ville han hellere skabe et sprog, der var godt nok på alle punkter, end et der var perfekt på få punkter. I de første udgaver af Python var der mange performance problemer, men med tiden er langt de fleste blevet optimeret. Guido Van Rossum fremhæver den sidste af ovenstående regler, som værende grunden til Pythons succes (ibid.).

En anden vigtig grundsten i Python-sproget fremgår i udvalgte linjer af Python-udvikleren Tim Peters tekst —Zen of Python‖:

- *Beautiful is better than ugly.*
- *Explicit is better than implicit.*

- *Simple is better than complex.*
- *Complex is better than complicated.*
- *Flat is better than nested.*
- *Readability counts.*
- *Errors should never pass silently.*
- *There should be one-- and preferably only one --obvious way to do it.*
- *If the implementation is hard to explain, it's a bad idea.*
- *If the implementation is easy to explain, it may be a good idea.*

- (Peters, 2004)

Python sproget prøver altså, at være smukkere end andre sprog. Sproget er bygget på at være visuelt forståeligt for udviklere, frem for udelukkende at være logisk. Derudover forsøges det at gøre sproget så simpelt som muligt. Der er flere eksempler på, hvordan Python forsøger at efterleve disse principper, og det vil vi komme ind på i de næste underafsnit.

Forskelle i Python og Java

For at øge læsbarheden, benytter Python indrykninger til at afgrænse kode-blokke, i stedet for fx tuborgklammer, i Java. Dette mener Van Rossum er den mest elegante løsning, der skal gøre koden let at forstå (Design And History FAQ — Python 3.6.1 Documentation, 2017). Desuden gør Python semikolon efter enden af et statement til en valgfri mulighed.

Dette betyder, at det er nemmere at undgå simple fejl, såsom at glemme en tuborgklamme eller semikolon. Men samtidig skal indrykninger have den samme størrelse, og kan, ved lange kodeblokke, godt gøre koden mere forvirrende at læse.

Python indeholder i grove træk de samme statements og udtryk som Java, fx *if* -, *while* -og *for* statements, dog ser nogle af disse lidt anderledes ud. Dette stammer fra Pythons filosofi om at være mere simpel og læsbart. Et for-loop i Java skrives typisk: `for (int i=1; i<4; i++){ doStuff(); }`Mens det tilsvarende i Python skrives: `for x in range(0, 4): doStuff()`Det er umiddelbart tydeligere i Python, at der skal ske noget 4 gange, end i Java. Java for-loopet består egentlig af 3 dele, nemlig at `i` skal defineres, en værdi for loopet sættes (altså den værdi `i` skal være lavere end), og derefter skal `i` forøges for hvert loop. I Python kræver for-loopet bare den 'range' den skal køre indenfor og et variabelnavn.

Typer i Python

Python bruger *dynamic typing*, dvs. at i modsætning til Java, har variabler i Python ikke en fast type, og kan derfor referere til en hvilken som helst værdi. Dette kan betyde flere fejl under run-time end Java, da Python-programmet ikke kompileres fra start, men i stedet fortolkes undervejs, som programmet køres. Det betyder også, at selvom man bruger et smart IDE (Integrated Development Environment), kan det ikke tilbyde hjælp til Python-kode, på samme måde som med Java-kode. Dette er fordi, man ikke skriver de samme informationer. Dvs. at hvis IDE'et ikke ved, at en variabel er en integer, kan det heller ikke informere om, at man fx ikke må give en integer til en metode, der beder om en String (Radcliffe, 2016).
Dynamic typing, betyder til gengæld også, at man ikke på forhånd skal vide præcis hvilke typer, der skal bruges, og kan på den måde, være lettere og hurtigere at skrive. Selvom Python bruger *dynamic typing*, er sproget stadig *strongly typed*. Dvs. at der er restriktioner på, hvilke værdier af datatyper, der kan blandes. Metoder i Python nægter fx, at modtage en string, hvis der er brug for en integer, da disse altså ikke er kompatible. Dog tjekkes dette igen først under run-time og ikke på forhånd af en compiler. Python programmer har dermed nemmere ved at få

fejlmeddelelser under run-time, end Java programmer. Blandingen af *dynamic* og *strong* typing kaldes også *duck typing* (ibid.).
I Python kan man definere egne typer, ved at bruge *classes*, som i Java. Dette gør Python egnet til Objektorienteret Programmering. Metoder er tilknyttet et objekts klasse, og kræver, at man bruger self (self.metodekald) foran alle metodekald og metodedefinitioner. Dette gøres for, at sikre, at der arbejdes med en metode eller instans og ikke en lokal variabel (Design And History FAQ — Python 3.6.1 Documentation, 2017). Dette er implicit i fx Java, hvor det ikke skal skrives.

Fortolker og compiler

Python bruger en fortolker til at oversætte kildekoden til maskinkode, hvorefter den udfører koden, linje for linje (se figur 4). Derfor kan der opstå flere fejl i run-time, da programmet pludselig kan støde på en linje, der ikke virker, får forkert input eller lignende, og programmet kan derfor ikke længere køre. Python kører således al den kode, den kan, før der kommer en fejl, hvor Java tjekker hele programmet og kompilere til et nyt, kompileret program (Sestoft, 2009). Her kan et program, der er kompileret, kun få fejl, hvis programmet får forkert eller ubrugeligt input.

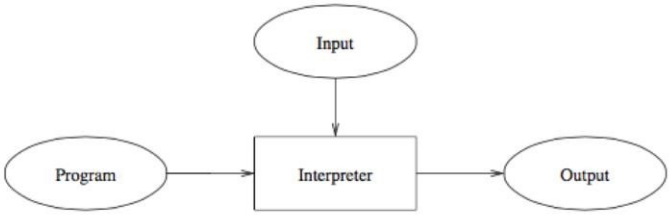


Figure 2.1: Interpretation in one stage

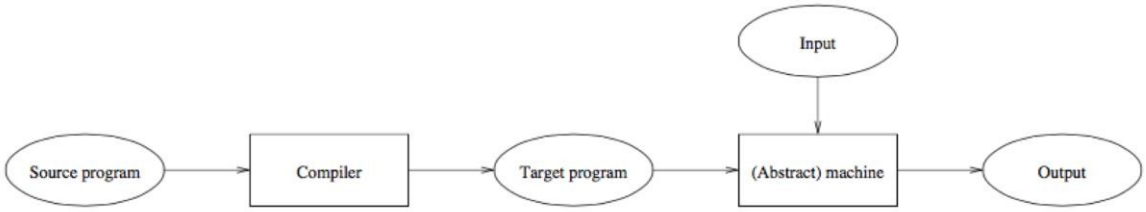


Figure 2.2: Compilation and execution in two stages

Figur 4 :Fortolker og Compiler (Sestoft, 2009)

Dette kan gøre at et Python-program kan være langsommere end Java, da det ikke bliver kompileret, og derfor skal fortolkes løbende. Dog er det generelt meget hurtigere at programmere i Python, grundet at man ikke skal deklarere typer, og derfor fylder selve den indtastede kode ligeledes mindre end i Java (Comparing Python To Other Languages, 2017). Python anvendes derfor ofte som et 'scripting language' – et sprog der ikke kompileres før det køres. Disse bruges til at integrere og kommunikere med andre programmeringssprog og bruges fx rigtig ofte til HTML. Dog findes gode compilere også til Python, der gør det muligt at skrive programmer i Python, og kompilere dem, så de tilsvarer et C eller Java-program. Derfor

beskrives Python både som et scripting– og et traditionelt programmeringssprog, og kan derfor ikke endegyldigt defineres som det ene eller det andet (Python (Programming Language) - Wikipedia, 2017).

Erfaringer med Python

Da mange aspekter af Python minder om Java, var det ikke en fuldstændig uoverkommelig opgave, at lære sproget at kende. Vi gjorde brug af en Python programmeringsbog (—Learn Python the Hard Wayll - Shaw, 2013), men fandt hurtigt, at det gik for langsomt til vores niveau. I stedet fik vi mere ud af, at følge tutorials på nettet, for rent faktisk at få skrevet noget programkode til et spil. Herved fik vi hurtigt syntaksen ind under huden. Selve programmeringslogikken i Python er ikke anderledes fra Java, så her var vi hurtigere med. På denne måde sparede vi tid på at skulle lære dette, og kunne lægge mere fokus på de områder, hvor de to sprog adskiller sig fra hinanden.

Python er intuitivt og nemt at lære, da man ikke i samme grad skal have styr på variabelnavne og andre allerede fastsatte måder at skrive bestemt kode. Dog gør fortolkeren det lidt sværere at finde eventuelt opståede fejl, end i Java. Der kan desuden kun findes én fejl ad gangen, som så skal fixes, før programmet kan køre videre, og flere eventuelle fejl kan findes. Sprogets enkelthed gør dog, at der ikke opstår så mange fejl. Desuden opstår der heller ikke fejl, som en manglende tuborgklamme eller semikolon, da det visuelt er nemmere at få øje på en manglende indrykning.
Det har taget tid at blive bekendt med Python, men det har hovedsageligt været dét at lære at programmere et spil, der har taget længst tid. Det indebar at lære at bruge Pygames library og finde ud af, hvilke metoder og struktur et godt spil kræver. Denne læring ville nok være den samme, om vi skulle skrive i Java, og kunne om muligt være endnu mere besværlig, da Java ofte kræver meget mere skrevet kode end Python.

Pygame

Dette afsnit er en kort beskrivelse af Pygame; et library til Python, der gør det nemmere at programmere multimedie-applikationer. Afsnittet kommer ind på, hvorfor det er en fordel at benytte et sådant library, og hvordan og hvorfor det indgår i vores programmeringsproces.

Wrapper-library

Pygame *wrapper* i virkeligheden, det allerede etablerede SDL-library til Python. SDL (Simple DirectMedia Layer) er et multimedie-library skrevet i C, som bliver brugt til mange spilprogrammer (og generelt multimedie applikationer) (Simple DirectMedia Layer - Wikipedia, 2017).
Pygame blev skrevet af Pete Shinners, der oprindeligt var C-programmør. Han opdagede Python og SDL-library samtidigt og på den måde opstod ideen om, at lave en SDL-binding til Python (Shinners, 2017).
SDL og Pygame er ‗wrapper'-librarys. Et wrapper-library er et program, der oversætter et library's interface til et mere kompatibelt interface. SDL wrapper nogle system-specifikke operationer, som herefter kan tilgås via dets library (Wrapper Library - Wikipedia, 2017).
Pygame, er derfor et SDL-library, wrappet til Python, med nogle ting bygget ovenpå det originale SDL-library. Således benytter Pygame C-kode og assembly kode til mange hovedfunktioner, hvilket er en del hurtigere end Python-kode. Det kører samtidig på de fleste platforme og operativ-systemer (About - Pygame wiki, 2017).

Moduler

Pygame startede med omkring 14 moduler, der gjorde det lettere at snakke med input udefra, og lave en multimedie-applikation. Fx er *image* et modul, som gør det muligt at loade billeder fra computeren til koden, og *key* et modul, som gør det muligt at snakke med computerens keyboard. I øjeblikket er der kommet over dobbelt så mange moduler til, som indeholder metoder, til at snakke med operativ-systemet og andet input _udefra'. Det indeholder desuden metoder, der gør det nemmere at skabe kommunikation mellem objekterne internt i koden - dette kunne fx være *Sprite* eller Rect modulet. Pygames Sprite vil i opgaven blive omtalt som spilobjekt. Vi vil ikke gennemgå alle moduler her, men man kan læse om dem, som vi benytter, i Bilag 4.

Erfaringer med Pygame

Følgende afsnit vil beskrive de erfaringer, vi har gjort os i arbejdet med frameworket, Pygame.
I forhold til vores brug af Pygame i konteksten af at udvikle et spil, kan man argumentere for, at fordi mange metoder på forhånd er skrevet i frameworket, giver det ikke en reel, dybdegående læring af, hvad der skal til for at programmere et spil. På den anden side, er det en del af programmering at benytte allerede eksisterende metoder, således at man slipper for, at skrive mere kode end nødvendigt. Vi koder jo ikke længere i binær, selvom det ville give en reel forståelse for, hvad computeren gør i et program. Desuden ville det være en sværere opgave, at lave et spil, som indeholder alle vores fokuspunkter, på kun et par måneder. Det kræver stadig en masse kodning og logisk tankevirksomhed, at programmere et spil med Pygame, men man kan så og sige bruge sin energi på de _spændende' metoder, der virkelig driver spillet og få foræret nogle af de mere _kedelige'. Det kunne ligeledes være en mulighed at benytte en _GameEngine' (som fx Unity). Dette er software designet til nemt, og hurtigt at kunne lave et spil. I en sådan _GameEngine' er meget givet på forhånd, og det ville nærmest ikke længere minde om _normal' programmering. Selvom dette sikkert hurtigere kunne have givet et flot, sjovt spil, var det aldrig rigtig en overvejelse, da det var vigtigt for os, selv at programmere hoved-spil-logikken.
Der har været en læring i, at forstå hvordan Pygames indlagte metoder fungerer, og lære hvordan, man benytter et library som dette. Det har ikke altid været lige nemt, da man ikke kan vide præcis, hvad metoden indeholder, når man ikke selv har skrevet den. Det har krævet en kombination af, at kigge i Pygames dokumentation, samt prøve at se, hvordan andre benytter metoder og så at prøve sig frem i koden.

Objektorienteret Programmering

Dette afsnit, vil beskrive hvad Objektorienteret Programmering er, hvordan dette fungerer i Python, samt hvordan og hvorfor vi har benyttet det i spil-koden.

White Rabbit bygger på konceptet omkring Objektorienteret Programmering (OOP). OOP er programmering, hvor koden er struktureret i klasser og instanser af klasserne, i form af objekter. Et programmeringssprog, som Java, er allerede objektorienteret, mens man i Python kan skrive både objektorienteret og lade være. Vi har valgt at bruge OOP, da vi allerede kender konceptet fra Java, og fordi OOP bidrager til programmets skalérbarhed og fleksibilitet.

Da *White Rabbit* gør brug af OOP i form af klasser, objekter og nedarvning, vil vi give et eksempel på, hvordan man opretter en klasse i Python og initialisere et objekt baseret på denne

klasse. Derefter vil vi forklare hvordan Pythons nedarvningssyntaks fungerer, og hvordan vores spilobjekters nedarvningshierarki ser ud.

Klasser bruges til at generalisere en type af objekter, som derefter kan initialiseres med unikke attributter (Klein, 2011-2017) . I *White Rabbit* er der en Enemy class og et enemy objekt. En klasse er et generelt billede på en type af et objekt. Enemy class defineres med forskellige attributter, som en enemy altid har. Herunder ville en class Enemy fx kunne indeholde: Enemy's type, Enemy's position og Enemy's scoreValue. Derudover vil en klasse typisk have metoder, som er de aktiviteter den kan udføre, eksempelvis kan en Enemy angribe. I *White Rabbit* er der mange forskellige klasser, som er generelle billeder af de objekter, der er med i spillet. Derfor vil vi gerne vise hvordan man skriver en klasse med Python syntaks. Det gøres således:

```
class Enemy:
      def __init__(self, type, x, y, scoreValue):
            self.type = type
            self.x = x
            self.y = y
            self.scoreValue = scoreValue
      def attack(self):
            print("WRAAR")
```

Når man har skrevet class Enemy, har man ikke oprettet en instans af Enemy endnu, men udelukkende skabt muligheden for initialisere et Enemy-objekt. I dette eksempel kræver klassens __init__ metode (klassens initialiseringsmetode), at Enemy-objektet får tildelt sine attributter; type, position og scoreValue. I Python vil class Enemy dermed skulle initialiseres således:

```
enemy = Enemy("Flying-Fish-Monster", 300, 450, 50)
```

Når class Enemy initialiseres bliver der skabt et enemy objekt, med egne specifikke attributter. Denne enemy er fx er Flying-Fish-Monster, på positionen x=300 y=450 og giver 50 score points når man slår den ihjel. Referencenavnet til denne Enemy, er enemy. Det er muligt at lave mange forskellige instanser af class Enemy og derfor er OOP en god programmeringstilgang, hvis man skal oprette mange objekter der ligner hinanden, men har forskellige variabler. Som eksempel på initialisering af to objekter fra samme klasse:

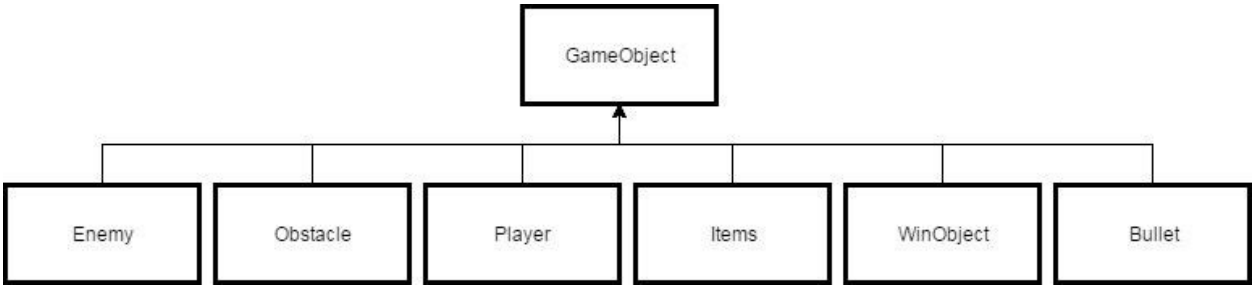
```
enemy1 = Enemy("Skeleton", 100, 100, 300 )
enemy2 = Enemy("Ghost", 20, 20, 15)
```

Begge objekter er af typen Enemy, men har unikke referencer til klassen og kan derfor blive tildelt unikke værdier til klassens attributter.

Anvendt Objektorienteret Programmering

White Rabbit består af forskellige typer af spilobjekter. Spilobjekterne er alle de objekter spillet udgøres af, herunder bl.a.: Player, Enemies og Bullets. Mange af disse objekter har ting til fælles; de skal eksempelvis initialiseres i programmet og opdateres efter input fra programmets bruger. Derudover har nogle af objekterne behov for de samme slags funktioner. Fx skal både Player, Enemies og Bullets kunne bevæge sig. Derfor vil vi kunne drage nytte af at skabe et overordnet nedarvningshierarki.

Her vil et overordnet GameObject, som har alle de mest gængse variable og funktioner, kunne nedarves til spillets forskellige objektklasser. Således kunne man skabe et hierarki, hvor de forskellige spilobjekters *egne* klasser, kun behøver de linjer kode, der afviger fra det øverste GameObject. Et hierarki kunne se sådan ud (figur 5):



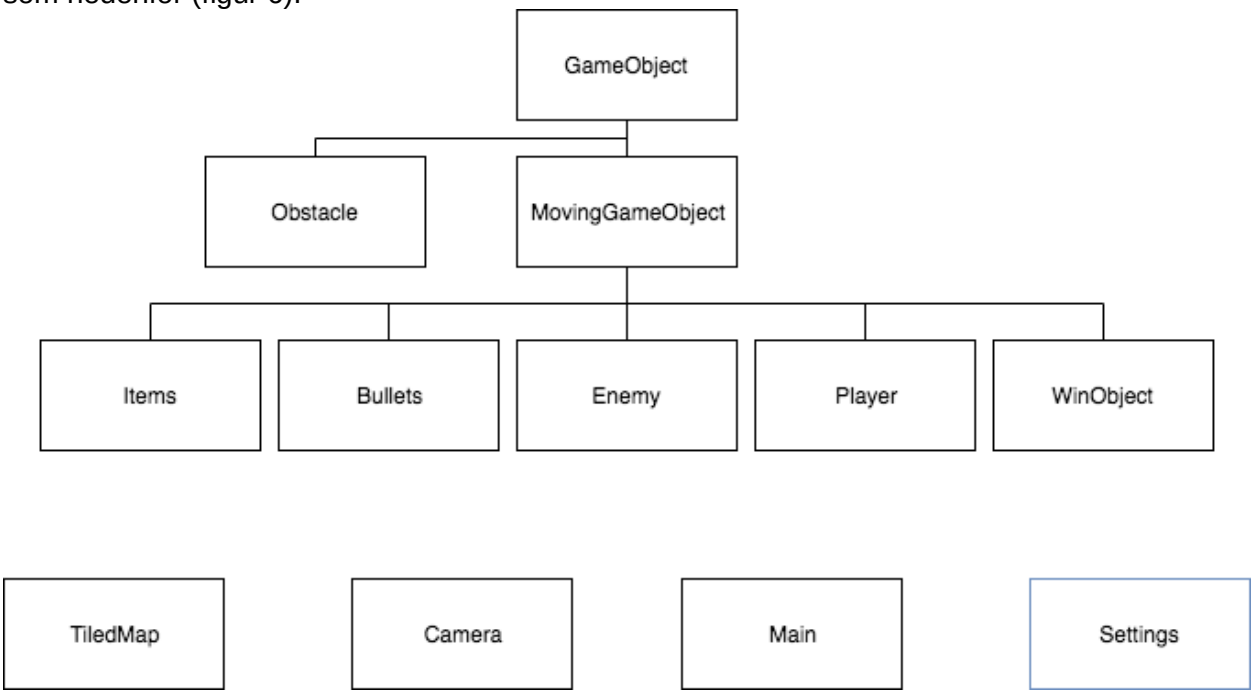
Figur 5: Eksempel på nedarvningshierarki

Det handler således om at gennemskue, hvornår flere objekter minder om hinanden, og der derfor eventuelt er overflødigt programkode, og mulighed for nedarvning. I *White Rabbit* skal alle objekter have attributter, som giver adgang til, at der eksempelvis kan tilføjes billeder til et objekt, og at det kan have en position. Derudover er der, som tidligere nævnt, en række objekter, som alle skal kunne bevæge sig. Det er således generelle attributter, som flere af objekterne i spillet har til fælles. Nedenstående eksempel præsenterer vores GameObject class og vores MovingGameObject class. GameObject class har nogle basale attributter, som nedarves til MovingGameObejct class - og denne klasse, har nogle attributter, som senere kan nedarves til bevægende objekter.

```
class GameObject(pg.sprite.Sprite):
    def __init__(self, game):
        self.game = game
        pg.sprite.Sprite.__init__(self, self.groups)

    def setupGameObject(self, game, image, x, y, hit_rect, layer,
type):
        self.game = game
        self.image = image
        self.rect = self.image.get_rect()
        self.rect.center = (x, y)
        self.hit_rect = hit_rect
        self.hit_rect.center = self.rect.center
        self.type = type
        self._layer = layer


class MovingGameObject(GameObject):
    def __init__(self, game):
        self.game = game
        GameObject.__init__(self, self.groups)

    def update(self):
        self.mask = pg.mask.from_surface(self.image)
```

Ovenstående class GameObject tilføjer her metoden, setupGameObject, som opretter nogle basale attributter. Disse arver MovingGameObject class nedenunder, ved at tilføre GameObject som parametre, når klassen oprettes. Ligeledes tilføjer MovingGameObject class metoden update, som indeholder en metode, der giver adgang til at objekter kan maske-kollidere med andre objekter. Denne metode, kan på samme måde, arves af alle de bevægelige objekter i spillet. Herunder eksempelvis class Player, Bullet og Enemy.

Således er det simple nedarvningshierarki fra tidligere allerede forældet. Vi implementerede derfor et multipel nedarvninghierarki. Her kaldes den første klasse superklassen, og klasser der nedarver herfra kaldes subklasser (Klein, 2011-2017). Således ser vores nedarvningshirarki ud som nedenfor (figur 6).



Figur 6: *White Rabbit's* nedarvningshierarki

Ovenstående figur præsenterer alle klasser og filer, som tager del i *White Rabbit*. Øverste del illustrerer vores nedarvningshierarki med de klasser, som deler fælles attributter og metoder, der derfor nedarves. Således ses det, hvordan GameObject nedarver til Obstacle og MovingGameObject, mens der sker en multiple nedarvning, når MovingGameObject igen nedarver til alle spilobjekter, som bl.a. skal kunne bevæge sig. Således arver disse ‗nederste' spilobjekter både fra den anden subklasse, MovingGameObject, og fra superklassen, GameObject.

Nederste bokse repræsenterer de klasser og filer, som ikke tager del i nedarvningen. Her fungerer TiledMap og Camera class, som to parallelle klasser, der ikke har stærke fællestræk med klasserne fra hierarkiet, og derfor ikke deler deres attributter eller metoder. Mens Game class er den klasse, som indeholder vores gameloop, læs mere i afsnittet *Gameloop*. Til sidst viser figuren den blå boks, Settings, som ikke repræsenterer en klasse, men i stedet en fil. I denne fil findes programmets konstanter. Konstanterne i programmet udgør her fx farver, egenskaber, lyde og billeder, som Settingsfilen sørger for er lette at ændre og give et overblik over.

Erfaring med nedarvning i White Rabbit

OOP kan være svært at implementere midt i en proces, og fungerer derfor nemmest, hvis det implementeres fra starten. Vi måtte dog implementere dette midt i processen, og selvom det tog et par dage at få styr på, har det gjort den videre udvikling af programmet og modificering en del nemmere. I et lille projekt vil OOP muligvis ikke være givende, da man ikke ville have særlig mange klasser af den samme type.
Disse nedarvningsstrukturer gør ikke blot, at vi slipper for at skrive flere linjer kode. De gør det meget lettere at implementere et nyt spilobjekt, da der allerede findes en 'skitse' til, hvad sådant et objekt, som minimum skal indeholde. De gør også, at programmet i højere grad kan bruges som _skabelon' til at skabe andre spil, og gør i højere grad spillet skalérbart, således at det fx er nemt, at tilføje nye Enemy-objekter.

Elementer i spil

Gameloop

I dette afsnit vil vi uddybe, hvad et gameloop er, hvordan det kan implementeres, og hvordan vi valgte at implementere det i *White Rabbit.* Et spilprogram adskiller sig fra de fleste andre programtyper ved hele tiden at køre i et loop. Dette loop kan med fordel konstrueres, som et *Gameloop*. Et gameloop er et design pattern, der sørger for, at programkoden gentages igen og igen, uden at blive stoppet af brugerens inputs og computerens processerkraft (Nystrom, 2014). Vores viden om forskellige gameloops stammer fra Robert Nystrom, som har skrevet bogen —Game Programming Patternsǁ og har arbejdet for Electronic Arts (EA Games) som spilprogrammør i 8 år.

De fleste programmer standser, når der ikke er noget input fra en bruger. Dette gør sig derimod ikke gældende i de fleste spil, hvor der netop findes et gameloop. Her fortsætter programmet med at udføre handlinger, også selvom der intet input er fra brugeren. I *White Rabbit* kan dette fx ses ved, at Enemies bevæger sig selvom Player står stille. Den mest simple konstruktion af et gameloop i Python vil være:

```
while True:
      handle_events()
      update()
      render()
```

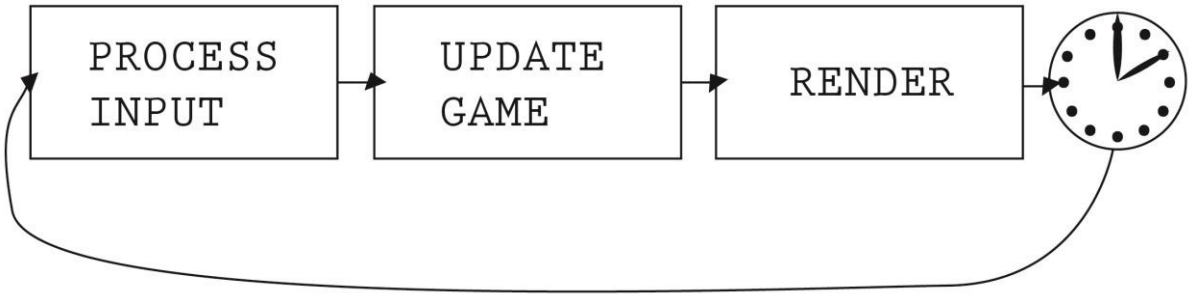
Loopet består af tre metoder: *handle_events(), update()* og *render()*. I handle_events() bliver der tjekket for input fra brugeren. Typisk vil brugerens input påvirke spilobjekternes attributter eller igangsætte deres metoder. Derfor skal disse indvirkninger på spillet opdateres i metoden update(). Til sidst køres metoden render(), der tegner spillets nye tilstand på skærmen. Et simpelt loop, som er konstrueret på denne måde, vil altid køre så hurtigt som muligt. Hastigheden ville afhænge af hvilken hardware spillet bliver kørt på, og hvor mange beregninger programmet skal lave per frame (Nystrom, 2014). På den måde ville et loop køre hurtigt på en nyere computer med en stærk processor og langsomt på en gammel computer uden så meget processorkraft. Når man snakker om tid i forhold til spilprogramme,r er det nyttigt at kende til begreberne *real time* og *game time*. Real time er den reelle tid, der går i den virkelige verden. Det er derfor den brugeren af et program opfatter. Gametime er den tid der går inde i spillet. Spillets tid øges efter hvert frame og derfor vil tiden gå hurtigere i et spil, hvor loopet afvikles hurtigt, end når loopet afvikles langsomt (Ibid.).

På denne måde vil game time, altså tiden som den opfattes i spillet, afhænge af hvilken platform den spilles på. Den reelle tid, også kaldet —real timeǁ, måles i frames per second (FPS) og er således, hvor mange gange gameloopet bliver kørt igennem per sekund. Hvis der er avanceret AI og physics i et spil, vil det ofte tage længere tid at køre igennem et frame, da processoren skal udføre flere beregninger. Når der er mange frames i sekundet, vil bevægelser i spillet opleves som glidende og naturlige, hvorimod de ved få frames i sekundet, vil opleves i hakker (Ibid.).

Derfor er det gameloopets opgave at sørge for at et spil får en nogenlunde konstant hastighed, så spilprogrammet opleves så ensartet som muligt. *White Rabbit* skal køres på en Raspberry Pi, og da der kan være forskellige processer i baggrunde,n der kører samtidig med spilprogrammet, har vi brug for et værktøj til at skabe et ensartet gameloop.

Fixed time step

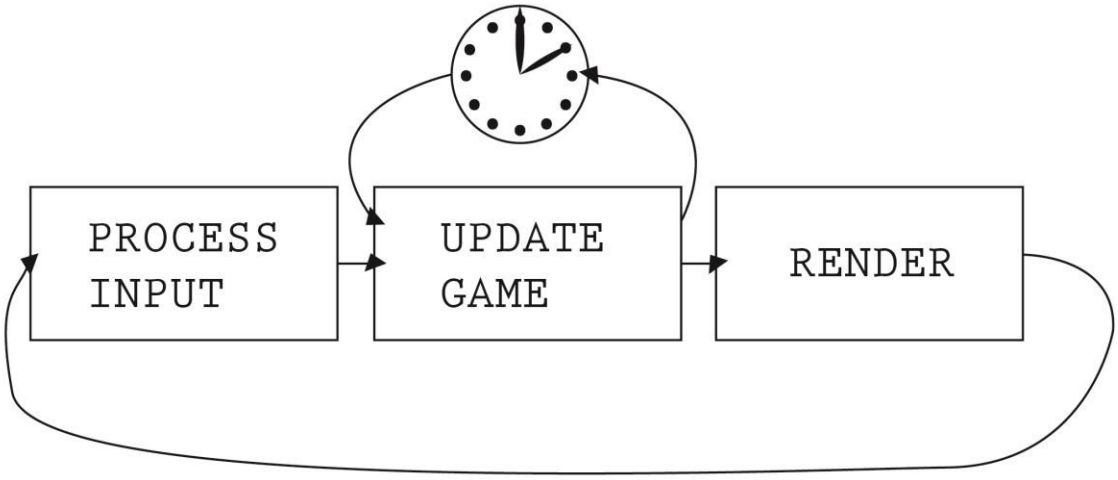
For at sørge for at kontrollere, hvor mange frames, der skal være per sekund, så findes der en metode som Nystrom (2014) kalder —Fixed time step with synchronization‖ (figur 7). Her lader man loopet vente et lille stykke tid, efter hvert frame, således at der højest kan være et fast antal frames i sekundet (FPS), ofte 30 FPS eller 60 FPS. Årsagen til at man vil ramme 30 FPS eller 60 FPS er, at de fleste standardskærme har en frekvens på 60 Hz, og dermed ser disse framerates mest naturlige ud. Denne metode løser problemet, der kan opstå ved, at den computer der kører spillet er for hurtig, men hvis den er for langsom, vil der stadig opstå —lag‖, hvor der bliver vist mindre, end de ønskede antal frames i sekundet.



Figur 7: Eksempel på fixed time step

Variable time step

På langsomme platforme, der ikke har en stærk nok processor til at kunne køre 60 FPS, findes der en måde man kan håndtere loopet således, at spillet alligevel bliver kørt i real time. Dette er, hvad Nystrom (2014) kalder —Variable Time Step‖ (se figur 8). Dette fungerer ved at elementer i spillet bevæger sig i forhold til real time, som bruges til at skalere hastighed op så spilobjekter ikke bevæger sig i game time, men kun i real time.



Figur 8: Eksempel på Variable time step

Variable Time Step er smart da det kan køre spil i samme hastighed på langsomme og hurtige platforme, men til gengæld er ulempen, at metoden giver mulighed for at spilobjekternes bevægelse hakker. Lag sker når objekternes position har rykket sig langt, uden at billedet er blevet tegnet. Dette kan skabe en dårlig spiloplevelse, og derfor har vi valgt at implementere fixed time step.

Implementering af gameloop i Pygame

Til implementering af fixed time step bruger vi Pygames time.Clock() til at sætte en framerate i *White Rabbit*. Det gør vi sådan her:

```
self.playing = True
while self.playing:
	self.dt = self.clock.tick(FPS) / 1000.0
	self.events()
	self.update()
	self.draw()
```

Når time.Clock.tick() modtager et argument skaber den et delay i spillet, sådan at det ikke kan køre med en hurtigere framerate end argumentet. Vores FPS argument er sat til 60 (i Settings-filen) og på den måde, prøver vi at styre frameraten til højest at være 60 FPS. Vi har testet metoden på en PC uden argumentet, og loopet blev dermed mere simpelt og derfor steg vores framerate til 180. Med argumentet i metoden kommer FPS, dog stadig nogen gange under 60, hvis der er for mange beregninger i gang. Dette kan man læse mere om i vores *test-afsnit*.

Opsamling
Vi har brugt Fixed Time Step loopet, for at holde Game Loop simpelt, og undgå lag. Det kunne dog, muligvis betale sig at teste begge metoder, i forhold til at opnå den bedste FPS, især når spillet køres på Raspberry Pi. Fixed Time Step fungerer dog udmærket på de fleste computere, i *White Rabbit*.

Collisions

Nedenstående afsnit beskriver kollisioner, og vil beskrive hvorfor kollisioner er essentielle i spil. Desuden vil vi beskrive Pygames forskellige kollisions-muligheder, nemlig `collide_mask`, `collide_circle` eller `collide_rect`. Desuden vil vi beskrive `spritecollide()` og `groupcollide()`, der er metoder til at holde styr på, hvem der kolliderer, og hvad der skal ske med dem.

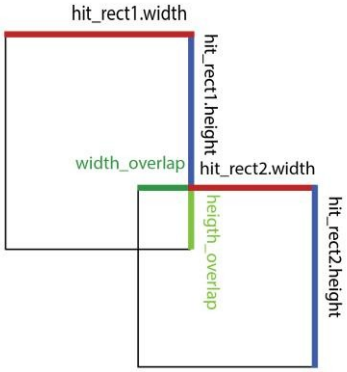
Kollision vil sige, at der sker et sammenstød mellem to objekter, hvor minimum den ene er i bevægelse. Kollision er et essentielt element, når man skal udvikle et spil. Tager man udgangspunkt i *White Rabbit*, har vi først og fremmest en Player-karakter, som skal bevæge sig gennem en bane. Det er et platformspil, hvilket betyder, at Playeren skal kunne bevæge sig frit på og mellem platforme for at komme fremad i banen. Hver gang vores Player står på en af disse platforme, er der tilført en collide-funktion. Dvs. at for at undgå at vores Player ikke blot falder igennem disse platforme, men rent faktisk kan stå på dem, er der programmeret en metode, som siger, at når Player 'rør' platformen, sker der en kollision, som her betyder, at Player ikke falder igennem. Ligeledes bliver kollisionen essentiel i forhold til at kunne tilføje enemies til *White Rabbit*. Her skal der ske noget specifikt, når disse to objekter støder sammen. Her er det muligt, at sammenstødet betyder, at Player dør, og mister et liv, men det kunne også være, at Player kolliderede med enemy-objektet under nogle bestemte betingelser, som gjorde at enemy-objektet i stedet døde. Uden kollision mister man simpelthen de essentielle rammer for, at lave et spil. Der sker nemlig konstant kollisioner i spil.

Sprite collisions i Pygame

Pygames Library tilbyder, at man kolliderer på forskellige måder. Nedenstående vil derfor forklare et udsnit af de forskellige konkrete kollisioner. Dette drejer sig om Pygames; `collide_rect`, `collide_circle` og `collide_mask`.
Ydermere vil afsnittet kort præsentere nogle af de kollisionsmetoder, vi har implementeret i *White Rabbit*. Vi har her anvendt nogle af Pygames indbyggede kollisionsmetoder, som findes under Pygames spritemodul (Pygame: Sprite — Pygame V1.9.2 Documentation, 2017).

Rect Collision

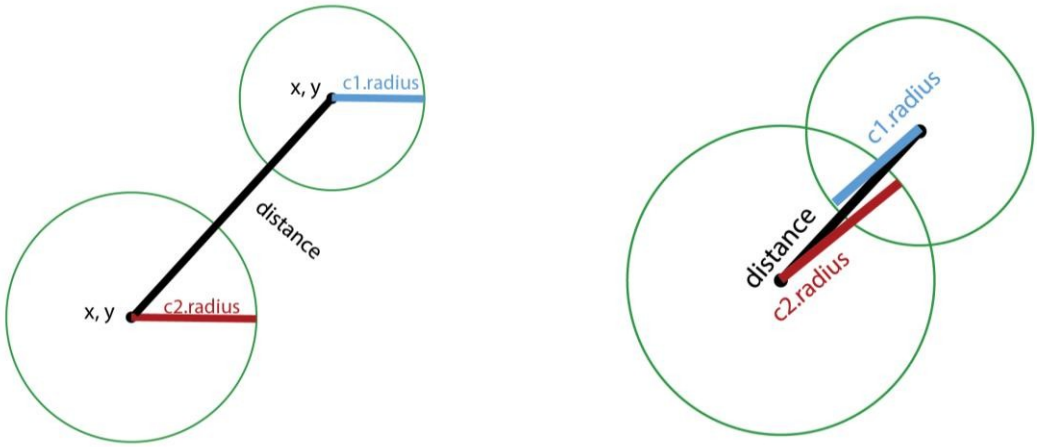
`Collide_rect` er en simpel kollision, som tilbydes af Pygame. Når et objekt oprettes i programmet, skal der tilføjes en rect-attribut til dette objekt. Denne rektangel kan bruges til at opdage kollision. Når kollision skal spores mellem to objekter, vil der tjekkes, om der er et mellemrum mellem fx Players rect og en Enemy rect. Så længe der er et mellemrum, altså luft mellem Player og Enemy, vil der ikke kunne spores en kollision. Det er således først, når de to rects står side om side, eller overlapper hinanden, at kollisionen opdages (2D Collision Detection, 2017).



Figur 9: Eksempel på rect collision

Figur 9 illustrerer, hvordan der kan tjekkes for, om to rektangler overlapper hinanden. Det er her nødvendigt at kende begge de to firkanters bredde og højde. Skitsen fortæller os her, at det *kun* er muligt at finde en kollision, når både begge firkanters bredde og højde overlapper hinanden. Da `collide_rect` ikke kræver meget processorkraft, kunne den være smart at bruge, hvis at `collide_mask` bliver for krævende. Vi har spilobjekter med forskellige former, og derfor bliver `collide_rect,` som udgangspunkt en for ‗grov' en kollision, der hurtig spænder ben for spillets muligheder. Det bliver muligvis kun ved kollisionen mellem Player og platform, samt rammen for spillet, hvor denne kollision bruges. Her er det ikke vigtigt at kollisionen er så præcis, men blot til formål at Player kan bevæge sig mellem platforme, samt kan dø og ikke komme uden for banen.

Circle Collision



Figur 10: Eksempel på circle collision

Ønsker man alligevel at implementere en kollision mellem spilobjekterne, som ikke bruger så meget processorkraft, men som er mere præcis, og visuel flot på et rundt objekt, end rect-kollisionen, kan man implementere `collide_circle,` som kan ses på figur 10. Denne form

for kollision tilføjer en radius-attribut til hver af objekterne, hvorefter den tjekker for kollision, ved at se om afstanden (*distance*) er mindre end de to objekters radiusser lagt sammen (2D Collision Detection, 2017). Er den mindre vil en kollision registreres.

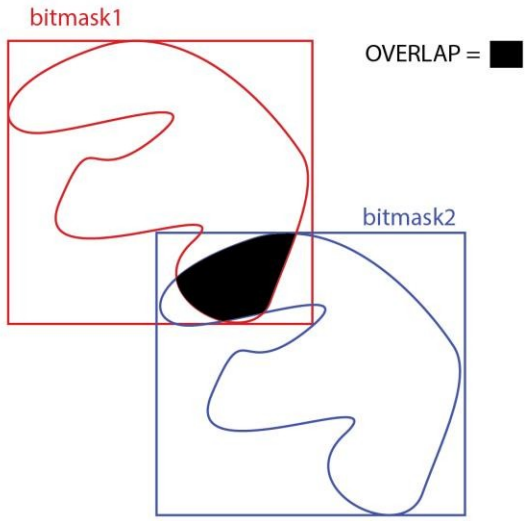
Mask Collision

`Collide_mask` er den fineste måde, der kan kollideres på, i Pygame. Dette svarer til det der også kaldes en ‗pixel perfect collision'.
I konteksten af et spil, ønsker vi, at vores Player skal kollidere så ‗tæt' som muligt med spillets Enemies. Således giver det en bedre spiloplevelse, da brugeren ikke oplever, at der sker en kollision, medmindre man tydeligt ser, at sammenstødet sker.
Her tilbyder Pygame at bruge en `collide_mask` kollision. Denne maske laves ‗omkring' de spilobjekter, man ønsker, skal have en sådan kollision.

```
class MovingGameObject(GameObject):
    def __init__(self, game):
        self.game = game
        GameObject.__init__(self, self.groups)


    def update(self):
        self.mask = pg.mask.from_surface(self.image)
```

Da vi ønsker, at det er muligt vores Player kolliderer på denne måde med alle andre spilobjekter, sørger vi i ovenstående kode for, at alle MovingGameObjects får en sådan maske. Således beskriver kommandoen i update, at der returneres en maske fra hver enkelt surface fra hver objekts billede (Se Bilag 4).

Når der tjekkes for maske-kollision, tjekkes der, om de to masker overlapper hinanden.



Figur 11 - Eksempel på mask collision

Denne maske er en såkaldt bit-repræsentation af det image, der fx er brugt ved vores Player. Dvs. at hvert pixel i dette billede er repræsenteret med en bit i masken. Når man loader billedet ind i programmet, bruger man Pygames convert.alpha() funktion, som sørger for sætte billedets baggrund, som gennemsigtig. Således vil den gennemsigtige baggrund blive repræsenteret med en bit-værdi på 0, mens de ikke-gennemsigtige pixels i billedet, bliver repræsenteret med en bit-værdi på 1. Når der sker en kollision mellem to spilobjekter, vil der på bit-niveau her blive tjekket, om der er tilfælde, hvor bits med værdien 1 overlapper hinanden (Pixel perfect collision detection in Pygame, 2012).

Udover forskellige former for konkret kollision mellem spilobjekter i *White Rabbit*, har vi ligeledes anvendt nogle af de metoder, som Pygame tilbyder, i forhold til kollision. Disse metoder findes i de basale objektklasser, som ligger under spritemodulet (Pygame: Sprite — Pygame V1.9.2 Documentation, 2017).

Enkelt Collision

Denne funktion finder spilobjekter i grupper, som kolliderer med et enkelt andet objekt (Pygame: spritecollide).

```
hits = pg.sprite.spritecollide(self.player, self.enemies, False,
pg.sprite.collide_mask)
if hits:
   self.died()
```

I ovenstående eksempel starter vi med at oprette variablen, hits, som vi sætter lig med Pygames spritecollide-metode. Metodens parametre beskriver, hvordan den tjekker om Player kolliderer med nogle Enemies, og returnerer en liste over de enemies, som kollideres. Det boolske parameter False, betyder, at der ikke fjernes nogle enemies fra denne gruppe under kollisionen. Det sidste parameter beskriver, hvordan der er tilføjet en maske-attribut til disse spilobjekter, som således kan lave en maske-kollision. Hvis der sker en kollision her, vil `self.playing` blive False, hvilket betyder, at Playeren dør.

Gruppe Collision

Denne metode finder alle de spilobjekter, som kolliderer mellem to grupper (Pygame: groupcollide).

```
pg.sprite.groupcollide(self.enemies, self.bullets, True,
      True,pg.sprite.collide_mask)
```

I ovenstående metode sørger en groupcollide for, at vi kan tilføje to grupper af spilobjekter, som parametre. De to boolske parametre sørger for, at begge objekter fra de to grupper, bliver fjernet, når de kolliderer. Som i førnævnte eksempel nedarver begge af disse grupper fra MovingObjects, hvilket vil sige, at de kan kollidere meget præcist, vha. deres maske som attribut.

Opsamling

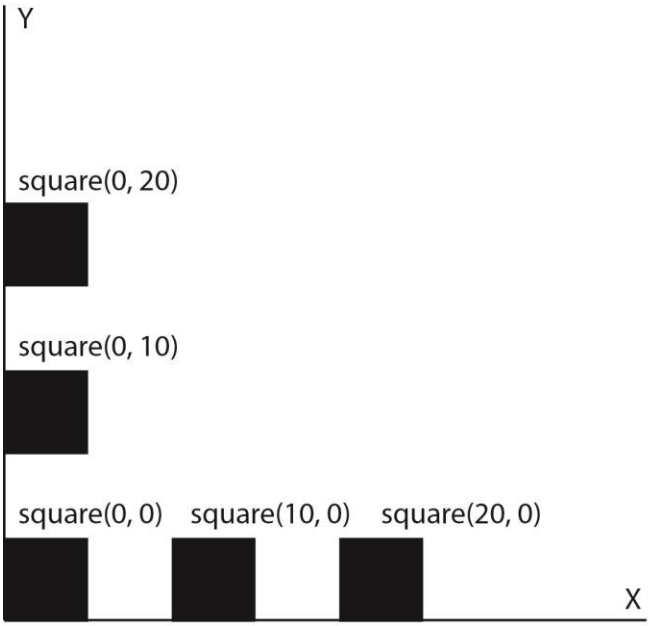
Uden kollisioner, vil et spil ikke være sjovt. Pygame tilbyder en række muligheder, for nemt at lave kollisioner, og derfor har det været lige til at implementere, i *White Rabbit*. Maske-kollision

tilbyder den mest realistiske kollision, og det har derfor været nærliggende at benytte den, til alle sprite-kollisioner. Dog kan det have en indflydelse på frameraten, og derfor er det værd at overveje, om nogle af disse kollisioner, i stedet burde være enten circle eller rect. Dette kan der læses mere om, i det senere *test-afsnit*.

Physics

I platformspil har man typisk en *Physic Engine* (PE)*,* som kan håndtere de fysiske love, objekterne i spillet bevæger sig ud fra. En PE kan programmeres på mange måder, alt efter hvilken grad af præcision, man ønsker. I de fleste platformspil er kravet om høj FPS ofte større end kravet om præcis gengivelse af virkelighedens fysiske love, da et arkadespil ikke i forvejen skal gengive den virkelige verden. Dette afsnit vil omhandle, hvordan man kan programmere en sådan simpel gengivelse af virkelighedens fysiske love. Der findes en simpel måde, hvor programmet simulerer bevægelse ved at øge eksempelvis spil-objektets x-koordinat i hvert frame, og en lidt mere præcis måde, hvor spilobjektet påvirkes af *acceleration, velocitet* og *position*. Vores program benytter begge typer fysik, og derfor vil dette blive uddybet i følgende afsnit.

X og Y fysik



Figur 12 - Eksempel på simpel simulation for bevægelse

Ovenfor, på figur 12, kan ses en firkant med forskellige x- og y-positioner. Hvis x-værdien øges, rykkes firkanten til højre, og hvis den sænkes, rykkes firkanten til venstre. Det fungerer på samme måde med y-værdien, blot vertikalt i stedet for horisontalt. Man kan derved nemt påvirke placeringen af objekter gennem x og y værdier. Hvis man fx vil skabe en simulering af et objekt, der bevæger sig mod venstre på en flade, skal man med den her fysik blot addere x-værdien med en konstant i hvert frame. Et eksempel ville her være:

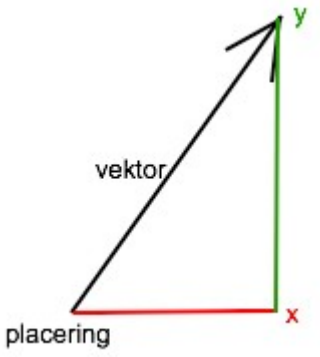
```
class Bullet(Object):
      def __init__(self, game, x, y):
            self.x = x
            self.y = y
            self.rect = (30, 30)
            self.rect.center = (x, y)

      def update(self):
            self.x += 10
```

Denne algoritme benyttes bedst til objekter, der har det samme bevægelsesmønster under hele programmets afvikling. I *White Rabbit* afspejler sådanne objekter bl.a. Bullets og Enemies. Bullets bevæger sig kontinuerligt ud af x-aksen, mens Enemies bevæger sig frem og tilbage mellem to punkter på x-aksen. Sådan en form for simpel programkode, som ikke afspejler en præcis gengivelse af faktiske fysiske love, vurderer vi i denne kontekst som acceptabel. I forhold til Player vil denne form for fysik dog blive for simpel. Her er Playerens bevægelsesmønstre bestemt af brugerens input, hvilket betyder, at han ikke har et forudbestemt bevægelsesmønster. Selve banen Player skal bevæge sig på, vil ligeledes variere med forskellige afstande mellem platforme m.m.. Således vil dette være for komplekst en opgave, at løse med den simple form for fysik, hvor man ville skulle have utallige `if/else`-statements, som konstant skulle afgøre, hvor højt eller langt Player eksempelvis kunne hoppe. Således vil Players fysiske bevægelse, i stedet være bestemt af nedenstående sammenhæng mellem acceleration, hastighed og position.

Fysik med acceleration, hastighed og position

For at lave en mere præcis gengivelse af virkelighedens fysiske love til vores Player, er det oplagt, at tage udgangspunkt i en ligning for bevægelse. En simpel udgave af sådan en ligning er:

Objektets position += Objektets hastighed + 0,5 * Objektets acceleration



Figur 13 - eksempel på vektorbevægelse

Det kræver således variablerne *position, hastighed* og *acceleration,* at lave en realistisk bevægelse i et spil. De tre variabler skal oprettes som vektorer. Vektorer er matematiske komponenter med to variabler i sig, som kan finde den korteste vej mellem to punkter, når de også kender en placering (et udgangspunkt). Således behøver man ikke bevæge sig kun på x- og y-aksen, men kan bevæge sig på tværs af disse (figur 13).

Pygame har en indbygget vektorfunktion, og vektor-variablerne kan oprettes således:

```
vec = pg.math.Vector2

class Player(Object):
      def __init__(self, game, x, y):
            self.pos = vec(x, y)
            self.vel = vec(0, 0)
            self.acc = vec(0, 0)
```

Position

Positionsvariablen tildeles samme parametre, som i den simple algoritme forklaret tidligere. Forskellen ligger i den algoritme, der bruges til at ændre bevægelsen. I stedet for blot at addere en konstant i hvert frame, adderes der med variablerne *velocity* og *acceleration*. Det er altså nødvendigt at forstå disse variabler for at forstå algoritmen.

Hastighed

Hastighed er den ændring der foregår i positionen over tid. I denne algoritme er der tale om en vektorvariabel med to variabler. Første variabel skal kontrollere objektets hastighed på x-aksen og anden variabel skal kontrollere objektets hastighed på y-aksen. Hvis man skal tilføje tyngdekraft i spillet, kan man således tildele en negativ værdi i hastighedens anden variabel. For at gøre dette realistisk, kan man tildele hastigheden gennem den sidste variabel i bevægelsesformlen *acceleration.* Uden acceleration bliver hastigheden nemlig unaturlig og vil gå fra 0% til 100 % ligeså snart den påbegyndes.

Acceleration

Acceleration er hastighedens ændring over tid. Dette komponent er også en vektor variabel med x og y akse. Det er denne værdi øger eller sænker konstant hastigheden, alt efter brugeren og de fysiske loves inputs. Hvis brugeren vil have Player-objektet til at gå til højre, skal accelerationens x værdi øges, således at hastighedens x værdien øges. Der skal tilføjes en friktion, der sænker accelerationen, så hastigheden sænkes ved manglende brugerinput. Desuden skal accelerationens y-værdi have en konstant negativ værdi, så Player trækkes ned, når der ikke er nogle forhindringer eller platforme.

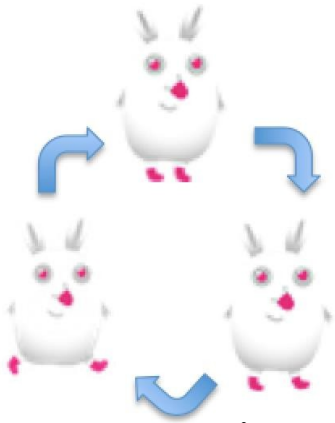
Physics i White Rabbit

Vi benytter os af physics, bestemt af acceleration, position og hastighed. Således kan playerens bevægelse bestemmes af brugeren, men samtidig minde om noget realistisk. Disse er alle bestemt af nogle variabler. X og y bestemmes af brugeren mens variablerne for hastighed og acceleration, nemt kan ændres og tilpasses ethvert spil. På den måde, bidrager implementationen af physics til spillets skalérbarhed. Det ville fx

også være nemt at lave et islandskabs-level, hvor Player-objektet gled rundt på platformene. Disse physics kunne evt. gøres endnu mere lig virkeligheden, men da det er et 2D tegneserie-lignende spil, føler vi, at denne form for physics passer bedst.

Animation

Følgende afsnit redegør for brugen af animation i *White Rabbit*. Herunder vil vi forklare, hvorfor vi vælger at animere vores spilobjekter, hvordan vi har gjort det, og hvordan Pygames Library bidrager til dette.
Vi har valgt at tilføje animation til vores spilobjekter. Formålet er her, at gøre dem mere livagtige og således forvandle spillet til et levende univers, som brugeren i højere grad vil blive draget af. Ligeledes kan animation bidrage til, at brugeren bedre kan tolke og forstå spillets implicitte agenda/formål.



Figur 14 - Eksempel på Walk-Cycle

De animerede objekter vil kunne signalere, at de har en funktionalitet i spillet og således guide brugeren i forhold til, hvad der kan interageres med og ikke blot er baggrund.
Dette afsnit vil tage udgangspunkt i animationen af vores Player. Animation vil sige, at man skaber en illusion om livagtig bevægelse. I dette spil animerer vi ved, at skifte mellem billeder. Disse billeder refereres her til som frames. Vores Player har eksempelvis 3 forskellige frames, som bliver vist efter hinanden i en gående tilstand. Disse 3 udgør tilsammen en 'walk-cycle' (se figur 14). Disse billeder, indlæses således:

```
self.walk_frames_r = [game.player_walk1, game.player_walk2,
game.player_walk3]
self.walk_frames_l = []
...
...
for frame in self.walk_frames_r:
   self.walk_frames_l.append(pg.transform.flip(frame, True,
False))
self.current_frame = 0
self.last_update = 0
```

Ovenstående kode findes i Playerens initialiserende metode, i Player Class. Her oprettes først en liste med de 3 walking-frames, som udgør Players _walk-cycle'. Disse walking-frames ligger

her i den rigtige rækkefølge, der skaber og animerer en naturlig, gående tilstand mod højre, når de aktiveres. For at vores Player også kan gå mod venstre, oprettes herefter en tom liste. Denne liste fyldes ved at lave et for-loop, som løber igennem alle frames fra Players oprindelige _walk-cycle', og tilføjer dem samtidig til denne tomme liste. Når frames tilføjes til listen, bliver de her transformeret vha. Pygames transform.flip(), som kan transformere en Surface i Pygame (se bilag 4) og flippe den enten horisontalt eller vertikalt. Således genbruger vi de 3 billeder i en spejlvendt udgave, som kan gå til venstre.
De sidste to linjer beskriver to variabler, vi bruger i vores animate-metode. Disse oprettes overordnet til at holde styr på, hvor hurtigt frames skal vises efter hinanden. Current frame holder øje med hvilket frame, man er nået til, mens last_update ved, hvor lang tid der er gået, siden der sidst er sket en ændring (skiftet frame).

Nedenstående kode tilhører selve Players animate-metode i Player class. Her er der først defineret en *now*-variabel. Denne er sat lig Pygames time, som således beskriver tiden der er gået siden Pygame blev initialiseret.

```
if self.walking:
    if now - self.last_update > 100:
        self.last_update = now
        self.current_frame = (self.current_frame + 1) %
len(self.walk_frames_r)
        if keys[pg.K_LEFT]:
            self.image = self.walk_frames_l[self.current_frame]
        if keys[pg.K_RIGHT]:
            self.image = self.walk_frames_r[self.current_frame]
```

I selve Players animate-metode, definerer vi forskellige tilstande, som vores Player kan befinde sig i. Ovenstående kode beskriver, hvad der sker i programmet, *hvis* Player befinder sig i en walking tilstand.
For at styrke en naturlig walking-animation, må vi tage højde for, hvor hurtigt der skiftes mellem de 3 frames, ved tilstanden Walking. Dette bør bedst muligt matche Players fysiske bevægelse henover skærmen. Således kan man opstille betingelserne, at *hvis* forskellen mellem tiden, der er gået (*now*) og sidste ændring (*last_update*) er mere end 100 millisekunder, så sæt `current_frame` til at være `current_frame + 1`, altså næste frame/billede, og sæt `last_update` til 'now', for at opdatere `last_update`. Til sidst findes remainder(%) af `current_frame+1` og længden af listen `walk_frames_r.` Denne returneres ved at bruge: `len(self.walk_frames_r).` Remainder beskriver det overskydende, når de to tal divideres med hinanden. Således starter `current_frame` forfra, når enden af `walk_frames_r` nås. Derfor vil der, ved en Walking-tilstand, konstant skiftes mellem billederne i listen hvert 100 millisekund. Dette bliver dog først aktiveret, *hvis* højre eller *hvis* venstre piletast trykkes.

Denne animation med frames gælder for alle animerede objekter, udover `items.` Da de kun har ét billede, og blot skal bevæge sig op og ned, så brugeren ved de kan interageres med, animeres de i stedet ved, at deres rect rykkes op og ned.

```
    if self.animatingDown:
        if now - self.last_update > 200:
            self.last_update = now
```

```
            self.rect.y += 5
            self.animatingDown = False
    if self.animatingDown == False:
        if now - self.last_update > 200:
            self.last_update = now
            self.rect.y -= 5
            self.animatingDown = True
```

Her er animationen i stedet bestemt af en boolean, som markerer om animationen går op eller ned. Når den går ned (`animatingDown`), rykkes y-værdien på items rect 5 pixel nedad. Herefter sættes `animatingDown` til False, således at den går videre til næste loop, hvor y-værdien i stedet rykkes 5 pixel op. Her sættes `animationDown` igen til True, og på den måde loopes de to tilstande hele tiden igennem. Igen bruges now og last_update til at måle tiden, således at items rykker op og ned hver 200 millisekunder.

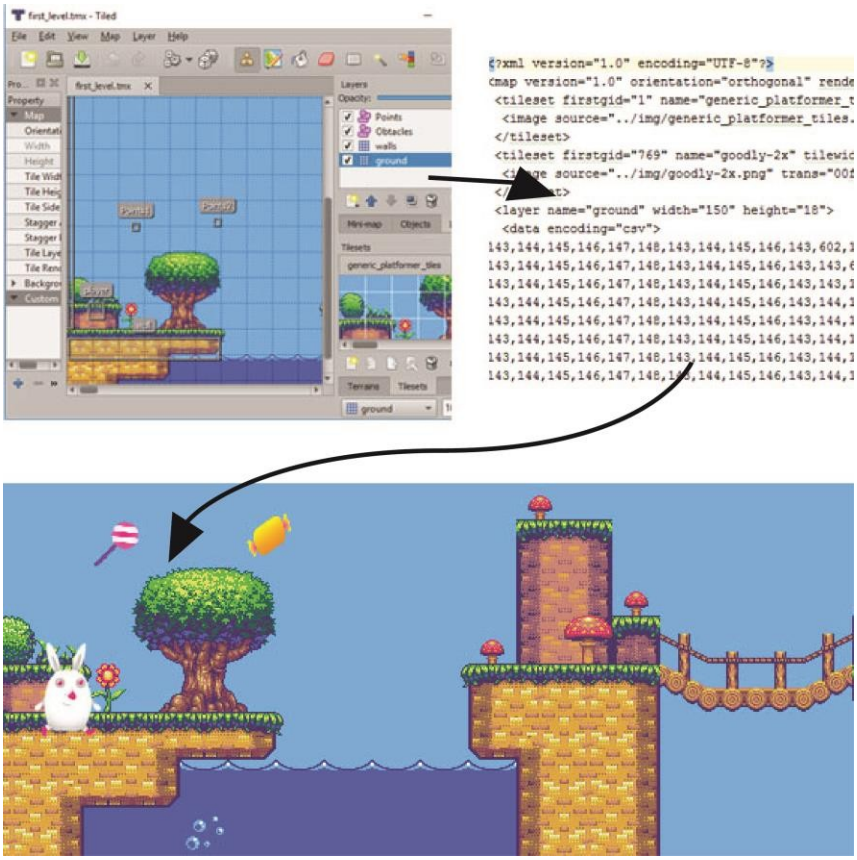
Opsamling

Vi har valgt at animere alle vores objekter, da det giver et mere levende spil. Det er ikke strengt nødvendigt, da det ikke direkte tilføjer nogen funktion til spillet. Dog bliver det meget flottere at se på, og at leve sig ind i som bruger. Det giver desuden også nogle implicitte hints om, hvilke objekter der kan interageres med. Da vi har animeret ved enten at gennemkøre billede-lister eller skifte mellem tilstande, kan disse nemt ændres/genbruges ift. at lave spillet om, eller genbruge koden til andre spil.

Leveldesign

Nedenstående afsnit beskriver programmet Tiled, som vi har brugt til at lave vores leveldesigns, og hvordan dette fungerer. Desuden vil vi beskrive, hvordan vi har benyttet det som en væsentlig del af *White Rabbit*s visuelle udtryk, og hvilke fordele og ulemper, der kan være ved at opstille en del af sit spil, i et Tilemap.

Tiled er en 2D Tilemap-editor, som kan bruges til mange forskellige slags opgaver, fx RPG-spil, Platformer-spil, eller lignende. Programmet er GUI baseret, og kan bruges til at designe levels til spil, baseret på Tiles. I *White Rabbit* laves hvert level, som et tilemap. Tiled er flexibelt, og sætter ikke nogle begrænsninger på størrelsen af maps. Man kan også arbejde i forskellige lag, hvor der kan indgå tiles eller objekter, som så kan benyttes i koden. Et lag kunne fx være tiles der kollideres med, og mens et andet lag har andre (eller ingen) funktioner. Disse tilemaps kan så importeres i spil-koden, og hvert enkelt objekt, kan hentes ind som et objekt der fx kan kollidere med andre objekter. Det vil fx typisk være platforme, som playeren kan gå og stå på. Et map kan bruge flere forskellige tilesets, og der er mulighed for, at ændre størrelsen på disse i tile-mappet (Tiled Documentation, 2017). Det er desuden muligt at definere kollisons-områder eller steder hvor specifikke GameObjects skal spawne. Disse positioner kan defineres i tiledmappet, mens objektet først indlæses i koden. I *White Rabbit* definerer vi fx hvilke positioner enemy skal spawne fra, og hvilke positioner objects skal være på.



Figur 15 - Eksempel på processen fra Tiled til Tilemap.

På figur 15 ses et lille udsnit af et tileset, der er blevet hentet ind i Tiled editor. Når et tileset er loadet ind i Tiled, markerer man blot hvilke tiles man vil bruge, og så kan de kopieres ind i et map. Således kan man designe levels på en nem og hurtig måde.

Når man har lavet et map bliver det eksporteret i formatet TMX (Tile Map XML).
For at hente Tiled TMX filer ind i Pygame-programmer, skal man bruge PYTMX. Der er Pygames map-loader, lavet til at hente Tiled Maps ind i Python-baserede spil. PYTMX håndterer de genererede Tiled objekter, og henter deres metadata, hvilket muliggør at ændringer kan ske i Tiled, i stedet for direkte i Python kildekoden (Tiled TMX Loader - Pygame, 2017).

Tiles i et Tilemap bliver gemt i en TMX-fil som har en XML lignende syntaks (Figur 16), hvor koden struktureres i tags.

```
313  <layer name="Tile Layer 1" width="50" height="300">
314   <data encoding="csv">
```

Figur 16 : Eksempel TMX syntaks

```
603  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
604  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
605  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
606  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
607  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
608  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
609  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
610  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
611  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
612  0,0,854,855,856,857,858,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
613  0,0,886,887,888,889,890,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
614  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

Figur 17 : Eksempel på udsnit af TMX-fil

På Figur 16, ses et udsnit af en TMX fil. På linje 313 ses et start tag med metaoplysninger om Tile Layer 1. Her ses højden og bredden, og på linje 314 gives oplysning om at den har en csv kodning. På linje 603-614 (figur 17) ses et lille udsnit af Tile Layer 1, hvor hvert tile har et unikt nummer, og en placering, i en csv-fil. Csv-filer er kommaseparerede filer, der bruges til at flytte større mængde data, mellem uafhængige databaser (CSV-Fil - e-conomic, 2017). Her ses en af fordelene ved at Bruge Tiled GUI editoren, da vi ellers skulle tegne alle vores maps på denne måde (altså give dem en unik placering), uden hjælp fra et Grafisk User Interface. Således kan man se ændringer i et map med det samme og placere tiles hvor man vil. Alternativet ville være, at skulle _gætte' den rigtige højde/bredde placering på et objekt, og så køre spillet for at se, om det passer ind. Dette sparer en masse tid og _kedelig' programmering, og giver samtidig mulighed for at lave et grafisk flot spil. Dette er også med til at gøre spillet mere skalerbart, da det vil være nemt, at udvikle, designe og implementere nye levels.

Implementering af Tiled i Pygame

Til at indlæse og oprette Tiledmap i programmet, har vi lavet en TiledMap Class i en tiledmap-fil. Her loades pytmx-filen og dens højde og bredde fastsættes. Desuden oprettes en render-metode, der tegner alle tiles på en Surface. Det gøres i et for-loop, der får x-, y- og image-variablerne på hvert tile i et tile-lag. Disse info (position og billede på det enkelte tile), blittes (tegnes) derefter på en Surface, der er lig tilemappets width og height. Alt dette gøres for hver enkelt tile i tile-laget.
I en make_map class, oprettes denne Surface, og render()-metoden bruges på denne. Således er der nu oprettet en Surface med vores tiled-map.

Desuden loades selve mappet fra game-folderen, hvilket sker i main. Her oprettes et image af vores map, som findes i make_map() fra tiledmap-filen.

Alle objekter fra object-map, loades i Main, og tildeles et objekt. Er objektet i object-map, fx en Enemy, tildeles det et enemy-objekt, og på den måde får Enemy en position at spawne i. Således:

```
if tile_object.name == 'enemy':
   self.enemy = Enemy(self, tile_object.x, tile_object.y)
   self.enemies.add(self.enemy)
```

Opsamling

Vi har valgt at bruge Tiled, da dette gør det meget nemmere og hurtigere, at designe nye levels. Når man bruger Tiled, får man nemmere et overblik over objekter og levels, end hvis vi skulle programmere alle elementer i et level, direkte i vores program-koden. Det giver nemt et grafisk flot spil, og mulighed for at designe nogle sjove og komplicerede levels. Når Tiled skal hentes ind i programmet, skal der selvfølgelig skrives en Class til det. Alt i alt virker det dog som om, at vi har sparet en masse ensformig kode (og tid) på at bruge Tiled. Derudover bliver spillet mere skalerbart. Det er derfor også nemt at ændre på levels og tilføje flere. Det ville desuden være muligt for andre, at designe helt egne levels, men ellers benytte koden til *White Rabbit*.

Camera

Dette afsnit vil beskrive, hvordan et kamera kan fungere i spil, og hvordan vi har implementeret det.
De fleste spil indeholder en kamera-funktion, til at bestemme hvilken del af spillets map, der skal tegnes på skærmen. Denne funktion er selvsagt vigtig i spil, hvor level-mappet er større, end hvad skærmen viser. Uden kamerafunktionen ville brugeren ikke kunne følge Player-objektet uden for skærmen. Et typisk spilkamera virker ved, at følge Player-objektets position og opdatere sin egen position til at passe hermed. Derudover flytter kameraet alle andre objekter i modsat retning, i forhold til Players bevægelser. På den måde ligner det, at kameraet følger Player, men det der sker er i virkeligheden, at alle andre objekter bliver rykket i et offset, der afhænger af Players bevægelser. Der findes flere typer kameraer; nogle flytter blot alle spillets objekter rundt, mens andre sørger for, kun at tegne de objekter, som er inden for kameraets frame.

Implementering af kamera i Pygame

Vores kamera har sin egen klasse, som initialiseres som en pygame.rect. Derudover har den metoderne apply(self, entity), apply_rect(self, rect) og update(self, target). apply og apply_rect, som implementeres således:

```
    def apply(self, entity):
        return entity.rect.move(self.camera.topleft)

    def apply_rect(self, rect):
        return rect.move(self.camera.topleft)
```

Begge metoder returnerer en ny firkant til enten et gameObject eller til en pygame.rect, og denne nye position afhænger af kameraets position. I programmets drawfunktion gennemløbes alle objekter, og her kalder vi så også disse funktioner:

```
self.screen.blit(self.map_img, self.camera.apply(self.map))
    for sprite in self.all_sprites:
        self.screen.blit(sprite.image,
self.camera.apply(sprite))
```

For at dette skal virke har kameraet en update-funktion:

```
    def update(self, target):
        x = -target.rect.centerx + int(WIDTH / 2) - self.shake.x
        y = -target.rect.centery + int(HEIGHT / 2) - self.shake.y

        x = min(0 - self.shake.x, x)  # left
        y = min(0 - self.shake.y, y)  # top
        x = max(-(self.width - WIDTH + self.shake.x), x)
        y = max(-(self.height - HEIGHT + self.shake.y), y)

        self.camera = pg.Rect(x, y, self.width, self.height)
```

Kameraet kører sin egen update()-funktion i hvert frame, da det er placeret i *spillets* update()-funktion. Her modtager det også sit target-argument. I *White Rabbit* er Player-target for kameraet. Kameraets firkant er således styret af Players x og y position. Da vi gerne vil have vores Player i midten af framet, har vi adderet x med halvdelen af bredden, og y med halvdelen af højden på mappet. Vi bruger desuden Pythons indbyggede min() og max() funktioner, til at sørge for, at kameraet stopper med at bevæge sig, når Player kommer tæt på mappets kant. Derudover har vi tilføjet variablen shake, som kan bruges til at skabe screenshake; en effekt hvor skærmen ryster.

Opsamling
Vi har implementeret den mest simple form for kamera, der tegner alt hele tiden og følger Playerens position. Dette virker rigtig fint, til et platformspil. Spillet vil teoretisk set køre mere stabilt og effektivt, hvis det kun er de objekter inden for kameraets rektangel der tegnes. Dette kunne være en oplagt måde, at forsøge at forbedre FPS.

Heads Up Display

Dette afsnit vil belyse, hvad Heads Up Display er, og hvordan dette er implementeret.

Figur 18: Eksempel på HUD, der viser overblik over Player-objektets score og liv

.

Et Heads Up Display (HUD) (figur 18) findes i langt de fleste spil. HUD bruges til at vise vigtige informationer om karakteren og spillets progression, altså udviklingen der måtte ske i spillets forløb. Det stræbes efter, at disse informationer ikke forstyrrer selve gameplay'et, men altid er visuelt tilgængelige for brugeren (Beal, 2017). I meget realistiske spil, vil man ofte stræbe efter, at inkorporere HUD'et i selve spillet, så det ikke blot er en række informationer, vist på toppen af skærmen. Dette afhænger ligeledes af, hvor mange informationer, der er nødvendige for brugeren.

I *White Rabbit* har vi valgt at lave en meget klassisk HUD, der viser informationer i toppen af skærmen. Her vises en information om karakteren, nemlig antal liv samt info om spillets progression i antal points. Dette er blot information til at hjælpe brugeren videre i spillet. Da det ikke er et realistisk spil, mener vi denne form for HUD passer bedst.
Da HUD'et er meget simpelt i *White Rabbit*, er det blot implementeret i Main. Man kunne overveje om det fx var Player class, der burde holde styr på sine egne informationer, eller Camera class der burde holde styr på, hvad der vises på skærmen. Da vi kun viser to informationer, har vi valgt blot at gøre det i Main. Her er først lavet en metode til at tegne tekst:

```
def draw_text(self, text, size, x, y, color):

    font = pg.font.Font(font_name, size)
    text_surface = font.render(text, True, color)
    text_rect = text_surface.get_rect()
    text_rect.midtop = (x, y)
    self.screen.blit(text_surface, text_rect)
```

Denne metode har brug for en tekst, en størrelse, en placering og en farve. Desuden sørger den for, at teksten får en surface (med `text_surface`), og at den surface får en rect (`text_rect`), som kan tildeles en placering på skærmen (x, y). Dette vises på skærmen ved `self.screen.blit()`. Denne metode benyttes i draw-metoden, hvor teksten for hhv. score og liv tegnes således:

```
self.draw_text("SCORE: "+str(self.score), 30, WIDTH / 8, HEIGHT
- 565, GREEN)
self.draw_text("LIVES: " + str(self.lives), 30, WIDTH / 3,
HEIGHT - 565, GREEN)
```

Her defineres hvad der skal stå, hvor på skærmen det skal stå, og med hvilken farve. `str(self.score)` printer variablen score, således at det er den opdaterede score, der tegnes. Det samme gælder for `str(self.lives)`, der tegner variablen lives, der holder styr på Players liv (se implementering i figur 18).

Opsamling
En HUD er en vigtig del af et spil, da det giver brugeren en mulighed for at gennemskue hvad der sker i spillet, og reagere på det. Her kunne være givet endnu flere informationer, men da spillet i sin natur er rimelig intuitivt, har vi valgt at holde det simpelt.

Screens

I vores HUD kunne vi også have valgt at vise, hvilket level der spilles i. Da det ikke er vigtig information under selve gameplayet, har vi valgt i stedet at lave nogle screens, der mellem hvert level viser, hvilket level man lige har vundet, samt ens score. Der er desuden en start-screen før første level, og en game-over screen, hvis man løber tør for

liv. Start-screenet viser information om, hvordan spillet spilles, mens game-over screenet viser den samlede score. Der kunne nemt tilføjes andre informationer. Ligesom HUD'et, er disse screens implementeret i Game class. Her kunne også tænkes i, om de kunne være en del af en anden klasse, eller have deres egen gameObject Class.
Her bruges det screen der vises mellem levels, som eksempel. En del af koden gennemgås ikke, da den minder om foregående:

```
def show_go_screen(self):
    self.screen.fill(BGCOLOR)
    self.draw_text("GAME OVER", 48, WIDTH / 2, HEIGHT / 4, WHITE)
    self.draw_text("Score: " + str(self.score), 22, WIDTH / 2,
HEIGHT / 2, WHITE)
    pg.display.flip()
    self.wait_for_key()
```

Denne del af kode tegne screenet, og _flippes' på skærmen.`self.wait_for_key()` kalder en metode, som _venter' på, at en tast bliver trykket, og spillet kører videre. Denne kaldes og skærmen vises når player kolliderer med win_object, og således har vundet et level.

Opsamling
Ligesom HUD, giver disse screens brugeren et indblik i spillet. Især den første screen der introducerer til, hvordan spillet spilles og Game Over-screenen, som lader brugeren forstå at spillet er tabt, er essentielle i et spil.

Perfomance Tests

For at vurdere forskellige designvalg og løsninger i forhold til *White Rabbits* performance, har vi måtte teste dem og sammenligne de forskellige resultater. I dette afsnit vil vi belyse hvilke tests vi har lavet, samt hvad vi har lært ved at udføre dem. Disse tests er udført ved, at vi har lavet ændringer i vores programkode, og derefter kørt *White Rabbit*, og noteret, hvilke ændringer i frameraten, de har skabt. Det er her ændringer i forhold til mængden af de forskellige spil-objekter, der er til stede i spillet og deres forskellige slags kollisioner.

Som tidligere nævnt har vi valgt at vores program skal forsøge at opnå en FPS på 60. Udover hardwarens indvirkning på FPS, kan denne også påvirkes af gode eller dårlige designvalg. Især designvalg omkring valg af kollisionsmetoder, antal objekter eller programmets evne til kun at beregne på objekter der vises på skærmen, har indvirkning på spillets framerate. Derfor har vi undersøgt hvordan forskellige typer kollision har påvirket frameraten, og om man kunne forbedre frameraten, ved at fjerne objekter når de forlader skærmen. Vi har testet på både en Mac-Laptop og en Raspberry Pi 3.

Tests: Kollisioner på MacBook Pro 13”

Dette afsnit vil opstille nogle test, som vi har foretaget i forhold til at undersøge, hvilke designvalg der påvirker FPS, og således være afgørende for, hvilke endelige valg vi foretager. Afsnittet vil beskrive nogle test foretaget med Pygames forskellige kollisionsmuligheder.

Som tidligere nævnt i afsnittet *Collisions*, ønsker vi at bibeholde en god brugeroplevelse, som betyder at netop kollisioner mellem vores objekter ser realistiske ud. Her ville vi gerne have så mange realistiske kollisioner som muligt. Som beskrevet i afsnittet *Collisions*, er Pygames Mask Collision, her den mest optimale i forhold til at imødekomme dette kriterie.

I følgende afsnit har vi foretaget nogle tests, som kigger på, *om* vi nedsætter, samt *hvor* og *hvornår* vi nedsætter FPS med forskellige kollisioner. Overvejelserne går på, om vi udelukkende vil benytte maske-kollisioner, eller om det kan betale sig at bruge enten rect- eller circle-kollisioner på nogle objekter. Således er det både den visuelle oplevelse og spillets performance, der skal medtænkes.

Disse tests er foretaget på en MacBook Pro, OS X Yosemite 10.10.5, processor 2,7 GHz Intel Core i5. Dette var nemt at teste, da det ligeledes er på denne platform, vi har udviklet spillet. Da spillet i sidste ende skal køre på en Raspberry Pi 3, med fx mindre processorkraft, vil resultaterne af disse første tests på en Laptop, således hjælpe os med at overveje, hvordan vi får den mest optimale ‗performance‘ på en Raspberry Pi.

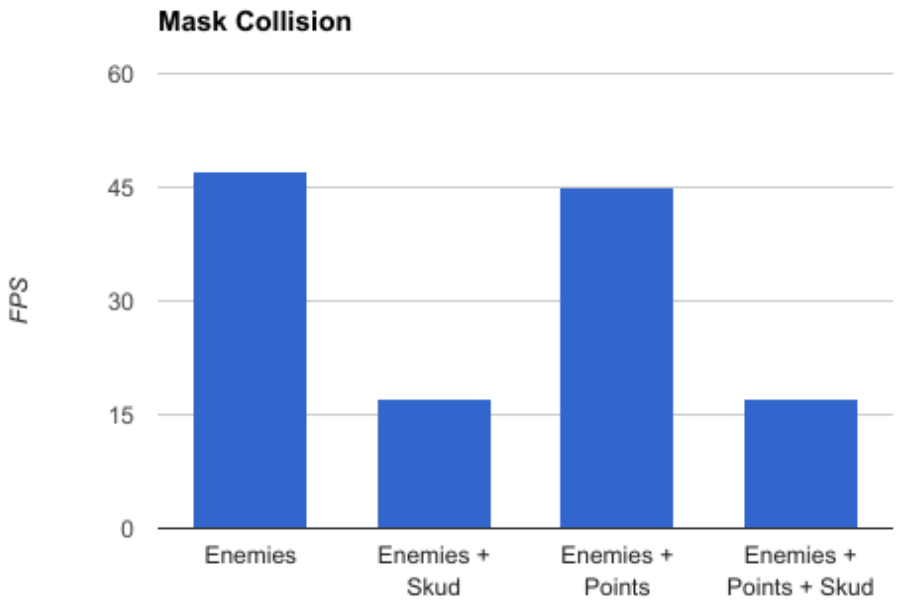
(*Der bør tages forbehold for at disse tests er lavet på en specifik hardware, med et givent styresystem. Derfor vil der også være mange andre processer kørende parallelt med det spil der testes, hvilket vil kunne give resultater, der påvirkes af dette. Vi synes stadig at testene er præcise nok til at give en forståelse form hvordan forskellige typer kollisioner påvirker et spil, som er konstrueret med Pythons framework Pygame*)

De tests der er foretaget i dette afsnit undersøger hvad der sker, når der foretages ændringer i kollisionsmetoder mellem udvalgte spilobjekter. Herunder hvordan påvirkningen er, når man har et højt antal af forskellige typer spilobjekter.

Følgende tre søjlediagrammer viser FPS, når der enten bruges `mask_collision` på alle spilobjekter, eller når der bruges `rect_collision` mellem udvalgte af dem. Hvert diagram beskriver fire tests med en kollision. Hver test er repræsenteret med en søjle, som beskriver, hvilket eller hvilke spilobjekter, der specifikt testes. Når et objekt testes, har vi oprettet ekstra af dem, for tydeligere at kunne gennemskue en sammenhæng.

Helt specifikt har vi testet med 44 enemies, og derefter tilføjet hhv. skud og points hver for sig og derefter sammen. Når vi har tilføjet skud har vi ikke et præcist antal, men derimod blot affyret så mange skud som muligt og dermed nået op på et antal, der passer realistisk med, hvad en bruger ville kunne tilføje selv. Når vi har testet med points har der været 17 points objekter, mod de 2 der plejer at være.

Nedenstående diagram beskriver sammenhængen mellem tilføjede spilobjekter og FPS, når alle spilobjekter kolliderer med en Mask Collision. På Y-aksen kan FPS afmåles (Figur 19).

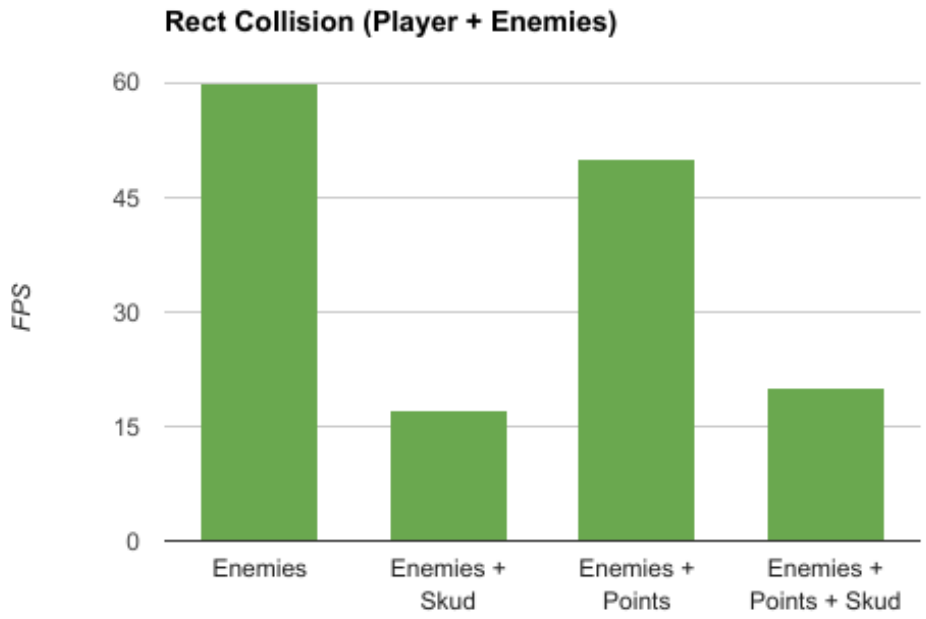


Figur 19: Diagram med Mask Collision

Overordnet viser diagrammet, hvordan ingen af de fire blå søjler, kan opretholde en FPS på 60, når alle objekter skal kollidere med en Mask Collision. Således kan man argumentere for, at der generelt er en sammenhæng mellem Mask Collision og nedsat FPS. Det er dog tydeligt at gennemskue, at der er en markant forskel mellem de søjler, som ikke tilføjer ekstra skud, og dem som gør. Eksempelvis falder første søjle, med flere tilføjede Enemies, fra omkring 45 FPS ned til ca. 15 FPS, når der samtidig tilføjes mange skud. Derfor er det her særligt skud, som påvirker FPS negativt.

Ligeledes kan man se, hvordan, FPS er en smule højere, såfremt der *kun* oprettes flere Enemies på skærmen, end hvis der både spawnes flere Enemies og Points. Således kan man muligvis samtidig bedst bibeholde en højere FPS, hvis Points er begrænset eller ikke er til stede, eller hvis der generelt set er færrest muligt objekter på skærmen.

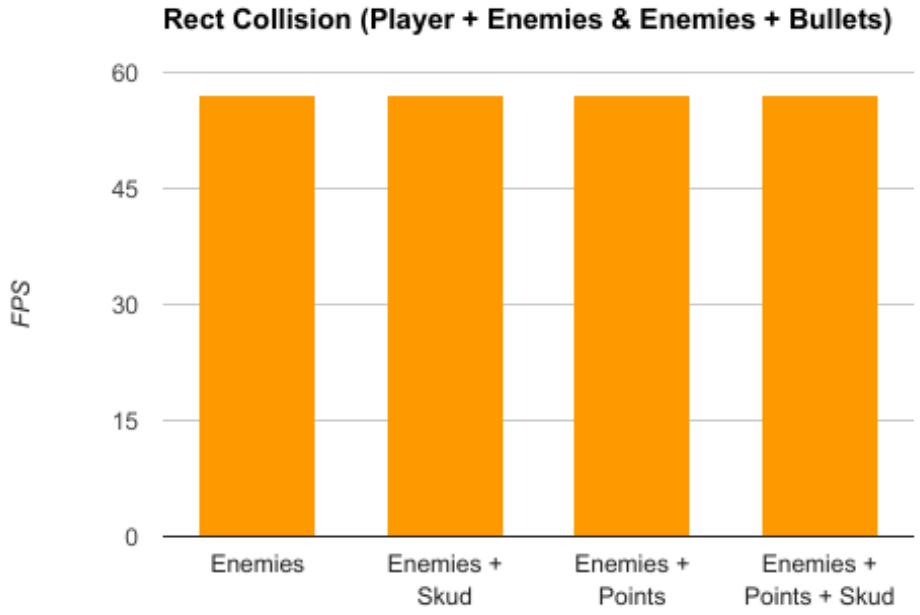
Nedenstående diagram beskriver sammenhængen mellem tilføjede spilobjekter og FPS, når alle point og skud kolliderer med en Mask Collision, og Player og Enemies, har en Rect Collision. På Y-aksen kan FPS afmåles (Figur 20).



Figur 20: Diagram med Rect Collision mellem Player og Enemies

Kigger man på dette diagram, forekommer der ikke længere noget fald i FPS, når det kun er Enemies, som der spawnes ekstra af (første søjle). Således er der tilsyneladende en sammenhæng mellem en stabil FPS og brugen af Rect Collision. De resterende 3 søjler ligner umiddelbart øverste diagram (figur 19). Dog er der en smule bedring. Dette kan betyde, at det således også har en betydning, hvor *mange* Mask Collisions, der er tilføjet mellem de forskellige spilobjekter. Som figur 20 viser, sker der altså en bedring i FPS mellem de resterende Mask Collisions, når Enemies og Player i stedet kolliderer med en Rect Collision.

Nedenstående diagram beskriver sammenhængen mellem tilføjede spilobjekter og FPS, når alle points kolliderer med en Mask Collision og Player og Enemies, samt Enemies og Bullets, har en Rect Collision. På Y-aksen kan FPS afmåles (Figur 21).



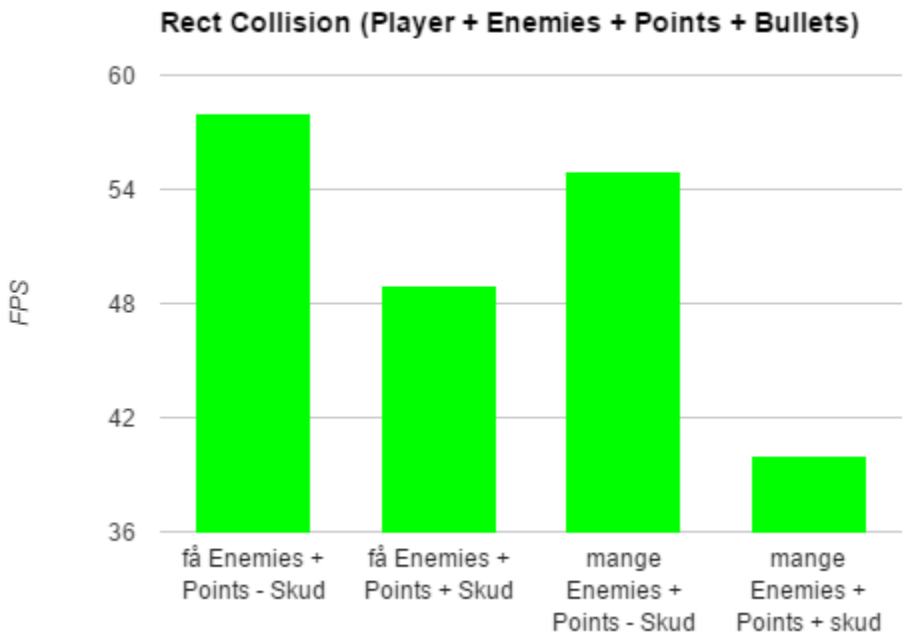
Figur 21: Diagram med Rect Collision mellem Player og Enemies, samt Enemies og Bullets

Dette diagram bekræfter, hvordan Rect Collision giver den mest stabile FPS. Her er alle søjler næsten på 60 FPS. Det er muligt at de ligger lige under 60, da der stadig er tilføjet Mask Collision mellem Player og Items og Player og WinObjects, således at der stadig bruges kræfter til at tjekke for en sådan kollision.

Tests på Raspberry Pi 3

I dette afsnit er der lavet tests på Raspberry Pi 3 med operativsystemet Raspian GNU/Linux 8 (Jessie). Disse tests er lavet med henblik på, at prøve at have samme tilgang som test på Macbook pro 13ll i tidligere afsnit.

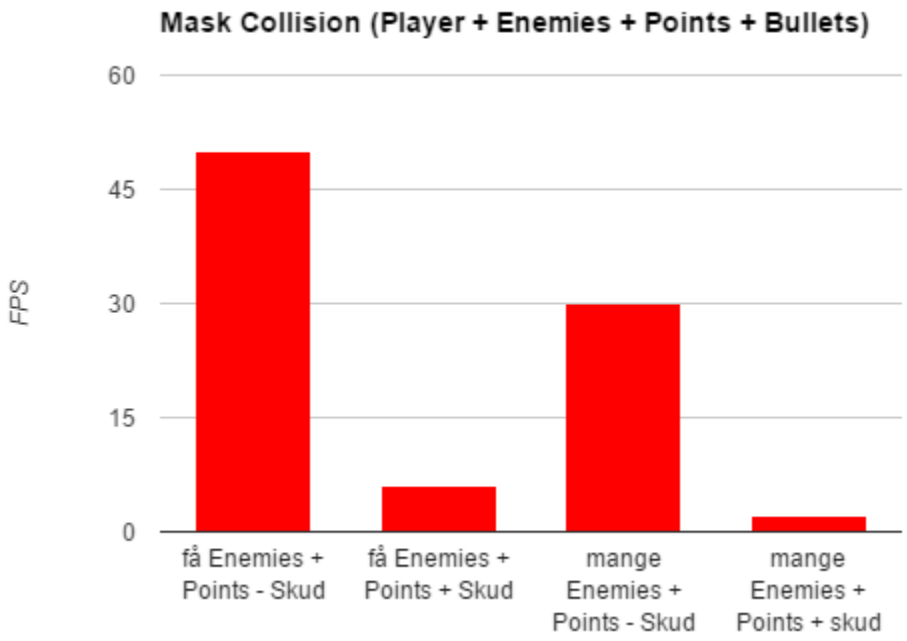
Første test er lavet med Rect Collision med 11 enemies og 2 points, på alle gameobjects. Derefter er der lavet en test med mange enemies, og items, hvor den mest hyppige framerate er blevet noteret. Efter dette er der tilføjet mange skud, på samme måde som i den tidligere test. Derefter blev frameraten noteret, som den var, når der var flest affyrede skud på skærmen. Herefter blev tilføjet gameobjects, så der var 44 enemies, og 11 points items, og samme tilgang blev brugt, til at notere frameraten. Til sidst blev samme tilgang brugt, med `collide_mask` på alle gameobjects, den kunne tilføjes til altså, player, bullets, enemies og items. Og her blev framerate også noteret med få enemies og mange enemies.



Figur 22: Diagram med Rect Collision tilføjet til Player, Enemies, Points og Bullets

På figur 22 ses et søjlediagram over frameraten på Raspberry Pi 3, når programmet blev testet med Rect Collisions.

Hvis man kigger på denne test i forhold til den lignende test på laptop som den ses på figur 21, ses det, at Raspberry Pi allerede har svært ved at følge med, når der kun anvendes Rect Collisions, i modsætning til på laptoppen, hvor at alle søjlerne er på næsten 60 FPS, med denne indstilling.

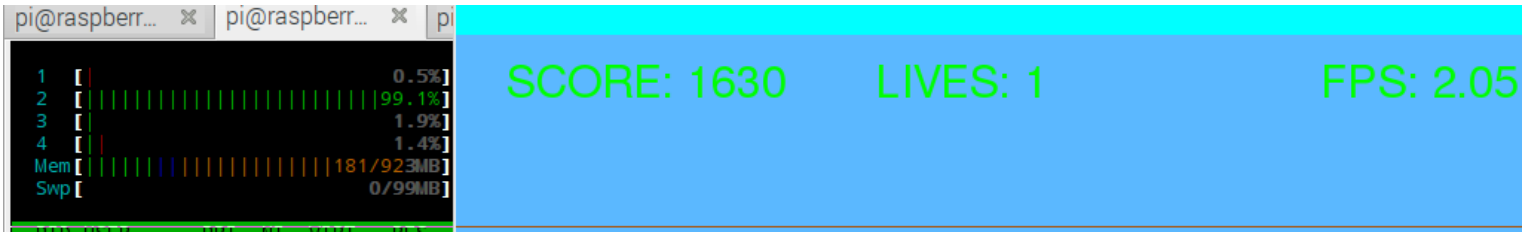


Figur 23: Diagram med Mask Collision tilføjet på Player, Enemies, Points og Bullets

På figur 23 ses et søjlediagram over frameraten på Raspberry Pi 3, hvor testen er lavet med `collide_mask`.

Hvis man kigger på denne test i forhold til den lignende test på laptop som den ses på figur 19, vil man bemærke at den første søjle her, faktisk har en højere framerate end på laptop. Men de efterfølgende søjler viser, at Raspberry Pi yder væsentligt dårligere end laptoppen, når der tilføjes flere skud eller flere enemies.

Udover dette, blev det også bemærket under disse tests, at FPS ofte var ustabil på Raspberry Pi, da FPS'en faldt til 30 ligegyldigt hvilke kollissionsmetoder der blev benyttet. Raspberry Pi's perfomance gav desuden 5-10 højere FPS, når spillet ikke var fullscreen.



Figur 24: Eksempel på lav FPS og fordeling af Raspberry Pi's processorkraft

Figur 24 viser et screenshot, hvor spillet er startet og en masse skud er affyret, og frameraten således er kørt helt ned til omkring 2 FPS. Samtidigt var der åbnet en terminal, med programmet Htop; en interaktiv proces monitor, der viser forskellige perfomance relaterede informationer. Dette er det sorte område, der kan ses til venstre på figur 24. Htop viser blandt andet hvilke processer, der kører på computeren og hvor meget processorkraft de bruger. Dette program viser også hvor mange procent processorkraft der bruges på de forskellige kerner, hvis det køres på en computer med mere end en processorkerne. På denne måde blev det bemærket at kerne 2 brugte omkring 99% processorkraft, imens de 3 andre kerner, kun brugte omkring 1-2 % processorkraft.

Udfra dette virker det til, at Pygame spil kun får tildelt 1 processor-kerne på Raspberry PI, som har 4 kerner. Perfomancen vil teoretisk set kunne øges, hvis der blev tildelt flere kerner til programmet. Dette ville være et område, der kunne undersøges nærmere, hvis performance skulle forbedres.

Test af objekter uden for skærmen

Under udvikling af spillet har vi forsøgt at teste, hvor stor en betydning det har for FPS, at objekter ikke bliver fjernet, når de forlader skærmen.

Vi har målt FPS når der konstant blev sendt skud afsted, uden at de blev slettet igen. Her faldt FPS hurtigt til omkring 27. For at fjerne og slette skud efter de forlader den del af skærmen der kan ses, har vi tilføjet en kill-funktion i metoden Update() i Bullet class. Dette gør, at når skud forlader det synlige område af skærmen, forsvinder de. Dette viste sig, at have stor betydning for FPS på en PC, hvor FPS nu forblev på omkring 60, selvom der var mange skud på skærmen. Samme test blev herefter foretaget på en Mac, hvor der ingen ændring opstod i FPS. Det kan dette skyldes at Mac har et bedre grafikkort, CPU eller hukommelse.
Vi kan altså konkludere, at for at opnå en god FPS, giver det kun mening at indlæse dét i spillet, der vises på skærmen. Det kunne derfor også give god mening, kun at indlæse den del af tile-mappet, der vises, eller kun spawne enemies når playeren kom tæt på.

Diskussion

Tests

På baggrund af vores testresultater vil vi diskutere, hvilken viden vi har fået fra vores tests, på hhv. MacBook Pro og Raspberry Pi 3, omkring spillets FPS og vurdere hvilke løsningsforslag, der er mest oplagte at arbejde videre med.

Vi fandt frem til, at de valgte kollisions-metoder kan have en stor indvirkning på FPS. Vi har dog i sidste ende vægtet brugeroplevelsen i spillet højt og derfor bibeholdt Mask Collisions på alle objekter. Ønsker man alligevel at forbedre FPS en smule i forhold til kollisioner, kan man vælge at tilføje `circle_collision` på eksempelvis skud. Disse havde ifølge testene en stor indflydelse på FPS og en circle kollision ville forbedre FPS samtidig med, at denne kollision visuelt ville passe bedre. I dette tilfælde ville man både kunne spare på beregninger og bibeholde en optimal brugeroplevelse.

Vi mener der er andre kritikpunkter ift. spillet, der kan arbejdes med for at skabe en mere stabil FPS - især på Raspberry Pi. Vi har ikke testet disse, men ser et stort potentiale i at finde en metode, som kan få Raspberry Pi'en til at bruge alle fire processorkerner, til at afvikle spillet. Derudover regner vi med, at der kan spares mange processorkræfter, hvis der kun bliver lavet beregninger indenfor det viste område af mappet. En sådan metode ville kunne implementeres i sammenhæng med kamera-metoden, der har en firkant omkring skærmbilledet.

Derudover kan man med fordel medtænke antallet af objekter, hvis man skal bruge en Raspberry Pi til at afvikle et spil programmeret med Python og Pygame. Man vil kunne teste hvor mange objekter spillet kan indeholde af gangen, uden de påvirker FPS. Derefter kan programmet designes til at indeholde begrænsninger, fx på antallet af skud Player kan affyre ad gangen. Begrænses antallet af objekter, måske enemies eller skud, kan det dog resultere i et spil, der ikke er lige så sjov/spændende at spille. For at opveje dette, kan man forsøge at skabe level-designs, der stadig er udfordrende på trods af færre antal objekter.

For at forbedre spiloplevelsen på Raspberry Pi, kan man også sænke det ønskede FPS til 30. Dette skaber en mindre troværdig grafik, da den er langsommere, men brugeren vil til gengæld ikke opleve en hakkende grafik. Denne løsning vil være nem at implementere, og da spillet er et arkadespil, vil brugeren højst sandsynligt ikke bide mærke i den lavere framerate. Dog ville nogle variabler i spillet skulle ændres, så fx hastigheden på player og enemies sættes op, for at kompensere for den lavere framerate. Det samme gælder for animationerne.

Den sidste mulige optimeringsmulighed er, at ændre gameloopet til at være af typen —Variable Time Step‖. Dette vil muligvis, som forklaret i afsnittet *Gameloop*, resultere i en hakkende grafik, især når FPS er så lav som testene på Raspberry Pi viste. Metoden opretholder til gengæld brugerens oplevelse af spil-tidens sammenhæng med reel tid, og kunne derfor også afhjælpe problemet med lav FPS, hvis altså det skyldes processorkraft. Det ville være værd at teste denne metode, hvis de andre muligheder ikke er tilstrækkelige.

Dokumentation, tutorials og eksperimenter

I konteksten af en læringsproces, med formålet at lære hvordan man programmerer og udvikler et spil, er det interessant at overveje sammenhængen og brugen af henholdsvis Pygames

dokumentation, tutorials og vores egne eksperimenter. Derfor vil følgende afsnit forsøge at reflektere over disse tilgange, og hvordan og hvornår hver af disse muligvis har været en fordel eller ulempe.
Da vi skulle lære et nyt sprog at kende, har det været essentielt at Pygame havde en dokumentation, man kunne ‗slå op i' i starten. Vi har primært brugt dokumentationen til at forstå Pygames moduler og deres tilhørende metoder, samt hvordan disse bør implementeres i forhold til fx parametre og attributter. Ligeledes har vi anvendt tutorials for at få eksempler på, hvilke fokuspunkter, som er aktuelle i udviklingen af et spil og måder disse kunne blive implementeret på. Det har her været konkrete eksempler på opbygning og sammenhæng mellem disse fokuspunkter i en spilkontekst.

Samtidig hjalp tutorials os med konkrete eksempler i forhold til at overveje koncepter, der har betydning i udviklingen af et platformspil. Havde man i imidlertid *kun* brugt tutorials uden dokumentationen, ville det være usikkert, om man ville forstå Pygames funktioner ordentligt, og i stedet blot have kopieret uden at forstå programkoden ordentligt. Det ville muligvis senere ville have forhindret os i at eksperimentere på egen hånd. Ligeledes kan man argumentere for, at havde man *kun* brugt dokumentationen, havde det været en større udfordring, at begynde at implementere funktioner til et program, da vi muligvis havde manglet en forståelse af selve opbygningen af et spil-program med tilhørende fokuspunkter. Dokumentationen er her kort og præcis til fx at vise, hvordan en funktion skrives, samt hvilke parametre mm. denne skal have. Det er dog sværere at finde eksempler, som viser en kontekst, som kan hjælpe med konkrete udfordringer i konteksten af et specifikt spilkoncept.

Samtidig kan man argumentere for, at såfremt Pygame havde været dårligt dokumenteret, og der ikke fandtes tutorials på nettet, havde det været en større udfordring og taget længere tid at starte udviklingen af selve spillet. Det kan også argumenteres, at såfremt man aldrig har prøvet at programmere før, vil især dokumentationen muligvis være for overvældende. Her forventer Pygame dokumentationen, at man har en forforståelse for overordnet Python syntaks, mht. fx metoder, parametre og variabler.
Således har det været en vekselvirkning mellem dokumentationen og tutorials, hvor begge har bidraget til en dybere forståelse af det konkrete sprog og frameworket Pygame, samt generelle spilkoncepter. Dokumentation og tutorials dominerer i starten af læringsprocessen, men efter at have etableret en grundlæggende forståelse for det nye sprog, samt fundet frem til -og startet på at implementere vigtige fokuspunkter, begynder processens eksperimentelle tilgang at træde til. Dvs. den del af processen, hvor vi programmerer, fuldstændig uden hjælp fra eksempler/tutorials. Det er parallelt med, at vi får skrevet mere på programmet, at vi begynder at udforske og udfordre det.

Vi modificerede på egen hånd i programmet, for at få en bedre og mere dybdegående forståelse af Pygame-metoder og sammenhænge. Vi tilføjede eksempelvis en shake-metode til kameraet for *endnu* bedre at forstå denne klasse. Ligeledes er det især i denne eksperimentelle fase, at nye idéer opstår. Programmerings-eksperimenterne bidrager til ny viden, herunder både noget som var målrettet, og noget som var tilfældigt. Dette var især gavnligt i den kreative proces i forhold til spillet, samt spilobjekternes evner og opførsel. Derfor blev det en fordel, at eksperimenterne kom senere i læringsprocessen, hvor der var et bedre udgangspunkt pga. en dybere forståelse for grundlæggende spilkoncepter, samt frameworket.
Således har både dokumentation, tutorials og egne eksperimenter betydning i en sådan proces. Det bliver her svært at undvære én af disse, da de hver især dominerer på forskellige tidspunkter i læringsprocessen og samtidig alle konstant er aktive.

Skalerbarhed

Dette projekt har ikke blot handlet om, at programmere et spil, men derimod om, at finde ud af, hvordan dette gøres bedst. Derfor har vi kigget på, hvordan en spilstruktur bedst laves, så den er skalérbar og flexibel, således at spillet nemt kan vokse sig større, eller at strukturen kan bruges som skabelon til at lave et helt andet spil. Det ville især kunne bruges som skabelon for andre spil-udviklere, hvis der blev tilføjet yderligere dokumentation, ift. hvordan programmet og de enkelte objekter er opbygget. Det er desuden også en del af Pythons filosofi (Rossum 2009), at programmer skal være skalerbare, både ift. at andre evt. skal kunne videreudvikle programmet i Open Source-ånden, men ligeledes for at gøre det nemmere, for den der selv sidder og programmerer.
Dét at vores leveldesign bygger på Tiled, er en af de bærende elementer i denne skalérbarhed og fleksibilitet. Uden det, ville hver bane skulle tegnes i program-koden, og hvert objekt sættes ind med en skrevet x- og y-værdi, som kan være svære at _gætte' sig til og det er dermed en udfordring at placere objektet det rette sted på ens level. Nu har hvert objekt i stedet et navn, så en bane kan laves i Tiled, og indlæses i programmet uanset, hvordan banen ser ud. Dette gør det muligt, nemt at lave et spil med et helt andet tema og udseende, og måske endda med et helt andet gameplay. Det er muligt, at *White Rabbit* hovedsageligt er skrevet som et platformspil, men der er ingen grund til, at levels skal bevæge sig mod højre, og ikke fx opad. Dog ville det muligvis kræve flere ændringer i koden, hvis spillet skal laves om til andet end et platformspil. Tiled egner sig desuden kun til 2D- og isometriske spil.

Vi har benyttet objekt-orienteret programmering, hvilket også bidrager til programmets skalérbarhed. Pga. GameObject class og MovingGameObject class, kan nye objekter nemt implementeres. Det burde desuden være nemt at lave en ny enemy, ved at nedarve enemy eller lave to players fra Player class. Dog kunne MovingGameObject class være endnu bedre implementeret i spillet, da flere af de klasser, der arver fra MovingGameObject class, har metoder og variabler tilfælles. Disse kunne være skrevet i MovingGameObject class, fx benytter alle animations-metoder Pygames time.get_ticks. Således kunne det være endnu nemmere, at implementere et helt nyt bevægelses-objekt i spillet. Her gælder måske generelt, at jo mere Objektorienteret et program er, desto nemmere er det at skalere. Derfor kunne der være implementeret endnu flere superklasser, således at disse var mere specifikke. På den anden side, ville for mange af disse øge programmets koblingsgrad. En øget koblingsgrad kan besværliggøre fejlsøgning og dermed er det ikke nødvendigvis en god ting med et stort klassehierarki.
Endnu en ting, implementeret for at gøre spillet nemmere at forstå og mere fleksibelt og skalerbart, er settings-filen. Den indeholder essentielle variabler, der benyttes i spillet, og sætter alle billeder, lyde og forskellige objekters rects. Derfor kan meget af spillet rigtig nemt ændres, ved blot at ændre i settings-filen, fx kunne playerens billede hurtigt skiftes ud, og dens bevægelsesmønstre ændres (fx gravity eller friction).

For at gøre klasserne endnu mere fleksible, ville man kunne tilføje bolske variabler til at styre ydre påvirkninger. Her kunne det fx være smart, at kunne sætte en bolsk variabel til at styre, om Player kan påvirkes af tyngdeloven. På den måde ville man ved at skifte en enkelt bolsk variabel, kunne lave et spil set fra oven. Her skal man blot være opmærksom på, at implementere dette uden at gøre resten af programmet forvirrende.

Game Class i main-filen kunne også optimeres. Den indeholder en del metoder, hvoraf et par stykker muligvis kunne implementeres i en anden eller deres egen klasse, hvilket kunne gøre programmet, og specifikt denne klasse, nemmere at overskue. Dette ville ikke nødvendigvis gøre spillet *mere* skalerbart eller fleksibelt, men gøre det nemmere at overskue, og derfor

indirekte nemmere at skalere eller ændre. På den anden side, er det Game class, der kører hele spillet, og derfor vil den oftest være lidt mere kompleks end nogle af de mere simple spil-objekter.
Alt i alt, vurderer vi programmet til at være relativt skalerbart og flexibelt, og kunne bruges som 'skabelon' for lignende spil, også selvom nogle ting kan optimeres.

Python

Dette afsnit vil reflektere og diskutere Pythons filosofi, ift. hvordan vi har oplevet at bruge og lære sproget.

En af fordelene, ved Python og Pygame i et læringsforløb, har været at der findes meget materiale, på mange forskellige niveauer. På denne måde har det været muligt at finde materiale, der gjorde det nemt at starte på et lavt niveau, hvor vi hurtigt kunne implementere og teste forskellige løsninger. Derefter har det også været muligt for os, at bygge videre på disse ved at forstå dem i dybden, og tilføje mere avanceret funktionalitet og struktur hen ad vejen.

Som det nævnes i Python afsnittet, er der nogle forskellige designprincipper, som har været med til at forme programmeringssproget. Hvis vi kigger nærmere på disse, med henblik på hvordan vores egne erfaringer med sproget har været, så kan følgende diskuteres: Ift. til udsagnet om, at ‖Beautiful is better than ugly‖ (Peters, 2009), mener vi, at Pythons påtvungne indrykninger i mange tilfælde har sørget form at koden har været nemt at få et overblik over. I Java vil begyndere hurtigt kunne komme til at lave et rodet program uden indrykninger og linjeskift. Indrykninger har været gode at bruge, da de har fungeret som en slags _nudging', der har hjulpet til at strukturere koden. Dog, ved fx at have mange nestede loops, kan man risikere at indrykningerne vil komme udenfor skærmvinduet, og derved skabe et dårligt overblik. En af de andre filosofier er dog at ‖flat is better than nested‖ (ibid.), hvilket peger i retning af, at Pythonsproget generelt bør skrives uden for mange nestede loops o.lign..

Endnu et af Pythons designprincipper lyder: ‖There should be one-- and preferably only one --obvious way to do it‖ (ibid.) Dette føler vi ikke altid er korrekt, da programmer bliver brugt til at løse problemer af forskellig art, og at der derved vil være skræddersyede løsninger, som vil passe bedre til et specifikt problem. På den anden side giver det dog god mening, at løse mindre problemer på samme måde hver gang, hvis denne måde er en god løsning, som er gennemtestet og virker. Hvis man skal arbejde sammen med andre, giver det også god mening, at alle bruger de samme metoder.
‖Readability counts‖ (ibid.) har i mange tilfælde været en god tilgang, da man hurtigere kan få et overblik over hvad en kodeblok gør. Derimod kan ulempen ved dette være, at koden er lavet på en måde der gør den forståelig for mennesker, men måske ikke viser lige så tydeligt, hvad der rent faktisk sker inde i computeren når koden køres. At koden har været mere læsbar, har også været med til, at vi hurtigere har kunne forstå Python-kode og derfor har kunne komme hurtigt i gang med at lære det.

‖The Python implementation should not be tied to a particular platform. It's okay if some functionality is not always available, but the core should work everywhere.‖ (ibid.).
Dette anser vi som en styrke ved Python, da vores program nemt kunne implementeres på forskellige platforme. Vi har fx kunne kode til Raspberry Pi, på MAC og PC. En ulempe ved dette kunne være, at et programmeringssprog udviklet til en specifik platform, nok ville kunne udnytte denne platforms ressourcer bedre, og derfor yde mere.

Vi har generelt haft en positiv oplevelse med at lære spiludvikling med Python og Pygame. Pythons filosofi omkring visuel og struktureret programmering, har hjulpet til, at man hurtigt kunne forstå programeksempler og med muligheden for at udvikle til flere platforme, har det været nemt at udvikle på Laptops til en Raspberry Pi.

Konklusion

I arbejdet med at udvikle et 2D-platformspil til Raspberry Pi i Python med frameworket Pygame, har vi tilegnet os viden om programmeringssproget Python og vigtige fokuspunkter i spiludvikling, samt hvordan man kan teste et spil. Denne konklusion vil være en opsummering af, hvilke erfaringer vi finder særligt vigtige for at kunne optimere fremtidige spiludviklingsprocesser.

Vores proces har afspejlet en undersøgende tilgang, med ønsket om at udvikle et spil til en Raspberry Pi. Vi har således gennem undersøgelser på nettet fundet frem til et passende programmeringssprog og framework. I starten har vi således programmeret ved hjælp af Pygames dokumentation og tutorials. Dette skabte et godt grundlag for at implementere de elementer *White Rabbit* består af. Efter den hurtige implementation, har det været muligt at eksperimentere med disse elementer og på den måde har både eksperimenter, programdokumentation og tutorials bidraget positivt til læringsprocessen.

Når man skal udvikle et 2D-platformspil fandt vi, at det var særligt essentielt at arbejde med elementerne kollision, fysik, animation, gameloop, kamera og leveldesign.
Gameloopet er altafgørende for at spillet kører og for spillets FPS. Her har vi benyttet os af Pygame, og det har været simpelt at implementere et *Fixed Time Step* gameloop.
Et platformspil ville ikke fungere uden kollisionsmetoder, og her kan `rect_collision`, `circle_collision` og `mask_collision` bruges i forskellige situationer, til at imødekomme både en god brugeroplevelse og en acceptabel FPS. Sammen med kollisionsmetoder kan man implementere simulationer af fysiske love, til at gøre spiloplevelsen realistisk for brugeren. Vi har desuden oplevet, at simple animationsmetoder, kan skabe en højere grad af forståelse for spillets objekter/karakterer, og en langt bedre visuel oplevelse af spillet. Desuden er det vigtigt at implementere et kamera, såfremt at levels skal være større end skærmopløsningen.
Vi har implementeret vores leveldesigns gennem XML-filer skabt med programmet Tiled, hvilket har haft en stor indvirkning på, hvor nemt det er skabe nye leveldesigns.

Da vi primært har udviklet og testet *White Rabbit,* på laptops, opdagede vi først sent at perfomance på Raspberry Pi var utilfredsstillende. Vi lærte på denne måde vigtigheden af at teste regelmæssigt på den ønskede platform, samt at tænke performance ind tidligt i et udviklingsforløb. I diskussionsafsnittet ‖tests‖ bliver forskellige løsninger på dette problem diskuteret.

Vi har lært, at med forhåndskendskab til Java, som har mange ligheder med Python, har det været nemmere og dermed en fordel at lære et nyt sprog. Desuden har vi anvendt tutorials til hurtig implementation og et hurtigt indblik i Python. Da vi har haft fokus på spil, er kendskabet til nogle dele af Python måske lidt overfladiske, men pga. sprogets simple syntaks, er det nok generelt nemmere at få et hurtigt indblik i og lære, end fx Java.

Frameworket Pygame har bidraget til at implementere funktioner, som ville have taget lang tid, selv at skrive. Det er desuden meget normalt i programmering at man ‘låner‘ hvor man kan og må, og vi ser ingen grund til, fx at skrive en kollisionsmetode selv, hvis vi ikke kan gøre det bedre end Pygame. Det har heller ikke været målet med opgaven, at lære at skrive disse metoder, men i stedet rent faktisk at få programmeret et spil.

Igennem vores arbejde med spillet, har vi fundet ud af, at det ved et større spil, er en god idé at indtænke skalerbarhed. Dette bliver især nemmere, hvis det indtænkes fra starten, således at fx OOP og nedarvningsstrukturer med superklasser laves før subklasserne. Da vi først implementerede dette, efter vi havde lavet visse subklasser, brugte vi unødvendig tid på, at få det til at fungere. Dog har det sparet tid i sidste ende, da det har været nemmere at implementere nye subklasser. Vi har fx, pga. Tiled, haft nemt ved at implementere nye levels og kan derfor i princippet udvide spillet til at have 40 levels, næsten uden at skulle ændre i koden. Desuden har vores Settings-fil har gjort, at vi nemt kunne ændre på nogle globale variabler ét sted, som nævnes flere steder i koden. At indtænke skalerbarhed og fleksibilitet i et spil-program, gør ikke blot at programkoden muligvis kan bruges som skabelon for andre spil, men gør det også meget nemmere, i løbet af programmeringen.

Vi har i løbet af denne proces opnået stor viden, både omkring dét at lave et 2D-platformspil, hvordan denne proces kan foregå og hvilke metoder og objekter, der er vigtige i et spil. Vi har opnået stor forståelse omkring brugbarheden af OOP og nedarvning, og generelt dét at lave et spil-program skalérbart. Vi har desuden fået forståelse for, hvordan forskellige platforme har indvirkning på FPS, og hvordan dette må medtages i programmeringen. Samtidig har vi lært Python at kende som sprog og Pygame som framework, disses fordele og ulemper, samt hvordan de kan benyttes til at lave et platformspil.

Litteraturliste

—2D collision detection‖. Developer Mozilla, 2017. Web. Set: 11. Maj https://developer.mozilla.org/en-US/docs/Games/Techniques/2D_collision_detection

—About - Pygame wiki‖. pygame.org. 2017. Web. Set 26. Maj, 2017. http://www.pygame.org/wiki/about?action=view&id=5659

Beal, Vangie, —HUD - Heads Up Display‖. 2017. Set: 21. Maj, 2017 http://www.webopedia.com/TERM/H/HUD.html

"Comparing Python To Other Languages‖. python.org. 2017. Web. Set: 19. Maj, 2017. https://www.python.org/doc/essays/comparisons/

"CSV-Fil". e-conomic.dk. 2017. Web. Set: 19. Maj 2017. https://www.e-conomic.dk/regnskabsprogram/ordbog/csv-fil

"Design And History FAQ — Python 3.6.1 Documentation". docs.python.org. 2017. Web. Set: 19. Maj, 2017. https://docs.python.org/3.6/faq/design.html?highlight=strong%20type

"General Python FAQ — Python 3.6.1 Documentation". docs.python.org. 2017. Web. 19. Maj, 2017. https://docs.python.org/3/faq/general.html

—Guido van Rossum‖ - Wikipedia. en.wikipedia.org. 2017. Web. Set: 26. Maj, 2017 https://en.wikipedia.org/wiki/Guido_van_Rossum

"Kidscancode.org". kidscancode.org. 2017. Web. Set: 19. Maj, 2017. http://kidscancode.org/

Klein, Bernd, —Python Course, Inheritance‖. 2011-2017. Set: 20. Maj, 2017 http://www.python-course.eu/python3_inheritance.php.

Nystrom, Robert, —Game Loop‖. 2014. Set: 9. Maj, 2017 http://gameprogrammingpatterns.com/game-loop.html.

Peters, Tim, —The Zen of Python‖. 2004. Web. Set: 26. Maj, 2017. https://www.python.org/dev/peps/pep-0020/

‖Pixel perfect collision detection in Pygame‖, sivasantosh.wordpress.com. 2012. Set: 11 Maj, 2017 https://sivasantosh.wordpress.com/2012/07/23/using-masks-pygame/

—Pygame: groupcollide‖, Pygame. 2017. Set: 11. Maj, 2017 https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.groupcollide

"Pygame: Sprite‖ — Pygame V1.9.2 Documentation. pygame.org. 2017. Web. Set: 19. Maj 2017. https://www.pygame.org/docs/ref/sprite.html

—Pygame: spritecollide‖, Pygame. 2017. Set: 11. Maj, 2017 https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.spritecollide

—Pygame (Python Game Development)‖-Playlist. YouTube.com. The New Boston. 2014. Web. set: 26. Maj 2017
https://www.youtube.com/watch?v=K5F-aGDIYaM&list=PL6gx4Cwl9DGAjkwJocj7vlc_mFU-4wXJq

"Python (Programming Language)" - Wikipedia. en.wikipedia.org. 2017. Web. Set: 19. Maj, 2017. https://en.wikipedia.org/wiki/Python_(programming_language)

Radcliffe, Tom, "Python Vs. Java: Duck Typing, Parsing On Whitespace And Other Cool Differences". ActiveState. 2016 Web. set: 19. Maj, 2017.
https://www.activestate.com/blog/2016/01/python-vs-java-duck-typing-parsing-whitespace-and-other-cool-differences

Rossum, Guido. 2009. "Python's Design Philosophy". python-history.blogspot.dk. Web. Set: 24. Maj, 2017. http://python-history.blogspot.dk/2009/01/pythons-design-philosophy.html

Sestoft, Peter, "Programming Language Concepts For Software Developers". 2009. Web. Set: 19. Maj, 2017.
https://pdfs.semanticscholar.org/7379/c281e748a532c02b96a8a42aeba38dda48b2.pdf

Shaw, Zed A. "Learn Python The Hard Way ". Addison-Wesley Professional, 2013. Web. Set: 19. Maj, 2017. https://learnpythonthehardway.org/book/preface.html

Shinners, Pete, 2017. —Python Pygame Introduction‖. Web. Set: 26. Maj, 2017.
http://www.pygame.org/docs/tut/PygameIntro.html

—Simple DirectMedia Layer‖ - Wikipedia. en.wikipedia.org. 2017. Web. Set: 26. Maj 2017
https://en.wikipedia.org/wiki/Simple_DirectMedia_Layer

—Tiled Documentation‖. doc.mapeditor.org. 2017. Set: 26. Maj, 2017
doc.mapeditor.org

—Tiled TMX Loader‖, Pygame. 2017. Tiled TMX Loader. Set: 26. Maj, 2017
http://pygame.org/project-Tiled+TMX+Loader-2036-.html.

—Wrapper Library‖ - Wikipedia. en.wikipedia.org. 2017. Web. Set: 26. Maj, 2017.
https://en.wikipedia.org/wiki/Wrapper_library