

**Anvendt programmering**

Forside til eksamensopgave

<b>Eksamenstermin</b> (sæt kryds)	Sommer: X	Vinter:
<b>Undervisere: Anne Buhl Bjerre</b>		

<b>Titel på eksamensopgave:</b> <i>Ikke relevant</i>	
<b>Min./max. antal typeenheder:</b>  1 studerende maks. 10 normalsider  2 studerende maks. 8 normalsider pr. stud  ekskl. forside, indholdsfortegnelse, bibliografi og bilag.  (1 normalside = 2400 typeenheder)	<b>Din besvarelses antal typeenheder<sup>1</sup>:</b>  <b>34.193</b>
Du skal være opmærksom på, såfremt din besvarelse ikke lever op til det angivne (min./max) antal typeenheder (normalsider) i studieordningen vil din opgave blive afvist, og du har brugt et forsøg.	
(sæt kryds)	
<input type="checkbox"/> Eksamensopgavebesvarelsen må gerne bruges i forbindelse med undervisning (navn, studienr. og lignende fjernes <i>ikke</i> fra opgaven).	
<input checked="" type="checkbox"/> Eksamensopgavebesvarelsen må gerne i anonym form bruges i forbindelse med undervisning (navn, studienr. og lignende fjernes fra opgaven).	
<input type="checkbox"/> Eksamensopgavebesvarelsen må ikke bruges i forbindelse med undervisning.	

<b>Tro og love-erklæring</b>
Det erklæres herved på tro og love, at undertegnede egenhændigt og selvstændigt har udformet denne eksamensopgave. Alle citater i teksten er markeret som sådanne, og eksamensopgaven, eller væsentlige dele af den, har ikke tidligere været fremlagt i anden bedømmelsessammenhæng.

<sup>1</sup> Læs mere om reglerne for omfang af opgaver i studieordningens § 5 Bedømmelse af prøver - Generelle regler vedr. formalia. Se i øvrigt eksamensbestemmelserne for disciplinen i studieordningen.

## Anvendt Programmering

**Afleveret af** (skriv kun fødselsdato og navn):

05-08-1989 Lasse Jensen

17-12-1991 Lasse Lotzkat Thorsen

14-07-1992 Christian Møllegaard Madsen

### Indholdsfortegnelse

Introduktion .....	5
Design og inspiration .....	5
Målgruppen .....	6
Problemformulering .....	7
Specifikation .....	7
Metode .....	8
Phaser JS .....	8
Objektorienteret programmering .....	8
Tiled .....	9
Spritesheets .....	9
Modellering .....	10
Pakke-Diagram .....	10
Programmering .....	11
Maps .....	11
Karakterer .....	15
Level-design .....	16
JavaScript .....	18
Game.js .....	18
Load.js .....	19
Menu.js .....	19
Narrative.js .....	20
Play.js .....	20
UI.js .....	26
Help.js og pause.js .....	27
Test og evaluering .....	29

## Anvendt Programmering

Instruktioner .....	30
Help-tekst placering .....	31
Konklusion .....	32
Refleksion .....	32
Fjender der vandrer igennem vægge .....	32
movePlayer() .....	32
Optimering af Tiled .....	33
Funktioner ved genstande .....	33
Gemmefunktion .....	34
Sværhedsgrad .....	34
Referencer .....	35

### Introduktion

Lasse Jensen & Christian M. Madsen

Som børn af slut 80'erne og start 90'erne er vi alle tre i studiegruppen vokset op sideløbende med den enorme udvikling, som spilindustrien har været igennem. Vi har alle haft vores første spilerfaringer på de første generationer af spillemaskinerne som Nintendo 64, Game Boy og PlayStation. Derfor var der ingen tvivl om, da vi skulle vælge mellem animering og spil, at vi ville udvikle et spil der refererede til de gamle dage, hvor spillene havde et simpelt layout og som krævede mindre kontekstuel forståelse af brugeren for at komme i gang. Da vi i undervisningen har arbejdet med side-scroller spillet Super Coin Box (Palef, 2021), var vores idé at arbejde med et andet type layout end side-scroller, for spillets opbygning. Ud fra disse aspekter, har vi udviklet spillet "Go North!", som vi i denne rapport vil beskrive nærmere, herunder hvordan spillet spilles, hvilken målgruppe vi har valgt, hvordan spillet er bygget op, og nogle refleksioner omkring hvordan vores spil vil kunne udvides, eller endda forbedres.

### Design og inspiration

Christian M. Madsen

Vi vil designe og producere et spil til aldersgruppen 5 år og opefter, der har til formål at være et underholdende spil, der er rimeligt let at gå til og komme i gang med, og som giver mulighed for at kunne udfordre sine venner og se hvem der gennemfører spillet hurtigst. Spilleren bliver en del af en fiktiv verden, hvor man indtager rollen som en modig ridder, der bevæger sig mod nord igennem 3 visuelt og stemningsfyldte tematisk forskellige maps, i jagten på tre items (en nøgle, et kort og en sandwich). Alle tre items skal være i ridderens besiddelse, før han kan komme igennem banen og gå videre til den næste. Alt imens skal man forsøge at undgå orkerne, der nådesløst jagter én. Får de fat i ridderen er det "Game Over", og man må starte helt forfra! Ridderen kan ikke tage kampen op mod orkerne, da han er arm-løs og uden våben, så hans eneste håb er flugten.

Spillet spilles ved at bruge piletasterne til at manøvrere rundt på banen, for at indsamle ressourcer i form af en nøgle, noget mad, og et kort. Kortet giver adgang til et mini-map, man kan bruge til at gøre det nemmere at få et overblik over banen og navigere rundt. Når alle 3 ressourcer er indsamlet, kan spilleren bevæge sig mod toppen af kortet, også defineret som at gå nord på. Når spilleren er på vej ud af den nordligste del af banen, vil banen skifte til næste bane, og målene for at indsamle ressourcer skal gennemføres igen, for at kunne skifte til næste bane.

Go North! Er stærkt inspireret af designet fra Pokémon spillene (*Pokemon Gold Version Download*, n.d.), vi som børn spillede på Game Boy, hvor man løber rundt i en åben verden set oppe fra og skal samle, træne og kæmpe,

## Anvendt Programmering

med og mod Pokémon's samt interagere med verdenen og andre NPC'er (non-playable characters). Vi ville således gerne lave en verden hvor man som spiller kigger oppefra og ned på verdenen. (figur 1 og 2.)



Figur 1 - Pokémon Game Boy



Figur 2 - Go North!

## Målgruppen

Lasse Jensen

Vi ønsker vores spil skal understøtte en meget bred målgruppe. Da vores spil leverer underholdning, er det oplagt at vores målgruppes alder starter ved omkring 5 år. Vores spils visuelle design skal appellere til børn fra den alder, hvilket betyder, at vi skal være påpasselige med at benytte voldsomme figurer eller visuelle effekter som f.eks. blod. Spillet skal stadig være udfordrende, så børn ikke mister interessen for hurtigt. Spillet må ikke have for mange komplicerede elementer, hvilket betyder at de opgaver der skal gennemføres undervejs i spillet, skal være let forståelige, evt. med visuelle elementer, fremfor tekst.

Da vores inspiration til spillet er fra da vi var selv var unge, håber vi, at vores spildesign vækker minder hos et ældre publikum, som kan genkende det nostalgiske visuelle design. Vi forventer således at vores spil vil tiltrække et det lidt ældre publikum op til alderen 40 år. Vi vil, når spillet er færdigudviklet, gennemføre en evaluering hos et repræsentativt udvalg i forhold til om spillet understøtter vores valg af målgruppe.

### Problemformulering

Hvordan kan vi udvikle et objective-based top-down spil, produceret i Phaser og Tiled, som har til formål at underholde spillere i alderen 5 år og opefter.

### Specifikation

*Christian M. Madsen*

Følgende afsnit vil omhandle de kravspecifikationer vi har sat til vores spil.

Spillet skal have en menu-scene, hvor man bliver introduceret til spillets navn og generelle grafiske stemning.

En instruktion til spillet vises ved at trykke på "H" tasten under spillet, og det skal være muligt for spilleren at pause spillet.

Når man starter spillet, skal der være en ny scene, der introducerer spilleren for historien i spillet og skal virke stemningsgivende. Hovedideen med spillet er at have en hovedkarakter der kan bevæge sig rundt på et map og interagere med forskellige items i spillet som skal samles op for at kunne avancere i spillet og til sidst gennemføre. Grafik og lyd skal understøtte hinanden i de forskellig tematiserede baner, og passe til de bestemte miljøer, som skal være vidt forskellige i hver bane.

De forskellige maps skal have hvert deres tema, og banerne skal indeholde terrængenstande som begrænser spillerens bevægemuligheder, ved at man kan kolliderer med miljøet. Yderligere skal der være fjender i skikkelse af orker, som er på jagt efter hovedkarakteren.

Spillet skal slutte hvis man kommer i kontakt med fjenderne, hvor spilleren herefter vil skulle starte helt forfra.

Der skal være ikoner i øverste venstre hjørne, som er de tre items man skal samle op i de forskellige baner. Dette har til formål at give spilleren et overblik over, hvad der allerede er samlet og hvad man mangler at samle for at kunne avancere til næste bane. Når spilleren samler disse items op, går ikonerne fra at være sorte til at være synlige.

Når man gennemfører hele spillet, skal der etableres en slutscene, der skal indeholde credits til udviklerne af spillet, samt en sluttekst der skal lykønske spilleren for at have gennemført spillet. Denne skal vise spilleren at man nu er nået til vejs ende og har vundet.

### Metode

#### Phaser JS

Lasse Lotzkat Thorsen

Et JavaScript-framework er et bibliotek af allerede skrevet kode og værktøjer, som kan hentes og benyttes af udvikleren, for at forbedre produktiviteten, kodens struktur samt holde det modulært. Det hjælper også andre udviklere ved at etablere et fælles sprog om koden.

Phaser JS er et sådant framework, der fokuserer på spiludvikling (Game Development) og leveres derfor med allerede sammensat kode til f.eks. fysikken i et spil (herunder tyngdekraft), input (tastaturtryk) samt kamerastyring. Da Phaser JS vises gennem en HTML-fil, er det let tilgængeligt på alle platforme og er derfor også et godt valg til udvikling af eventuelle mobilspil.

Phaser JS opererer ved hjælp af det, der kaldes "Scener". Disse kan startes, lukkes, pauses osv. På den måde kan udvikleren styre hvad "spilleren" oplever, og hvordan han/hun bevæger sig rundt i spillet mod slutningen.

En scene indeholder typisk 3 metoder:

- *"preload()"*, der indlæser de elementer, der skal bruges/manipuleres i koden, f.eks. billeder, lyde og spritesheets.
- *"create()"*, der opsætter de ovennævnte elementer og klargør dem på lærredet (den "rude" som "spilleren" ser).
- *"update()"*, der fungerer som en konstant løkke, der tjekker bl.a. eventuelle input fra spilleren og kollisioner.

Hver scene indeholder sin egen kode og er typisk gemt i separate JavaScript-filer.

#### Objektorienteret programmering

Lasse Lotzkat Thorsen

Phaser JS er bygget vha. objektorienteret programmering (OOP) principper, dvs. alt er opbygget omkring opdelingen af elementer i JavaScript-klasser. Dette gøres, så udviklere kan genbruge den modulære kode, holde koden organiseret og dermed også gøre det nemmere at vedligeholde. Andre fordele ved OOP er også



## Anvendt Programmering

nedarvning af kode fra f.eks. *Phaser.Physics.Arcade* til *Phaser.Physics.Arcade.Sprite*, polymorfi (evnen til at ændre allerede definerede klasser) og indkapsling, der kontrollerer tilgangen til koden.

### Tiled

Lasse Jensen

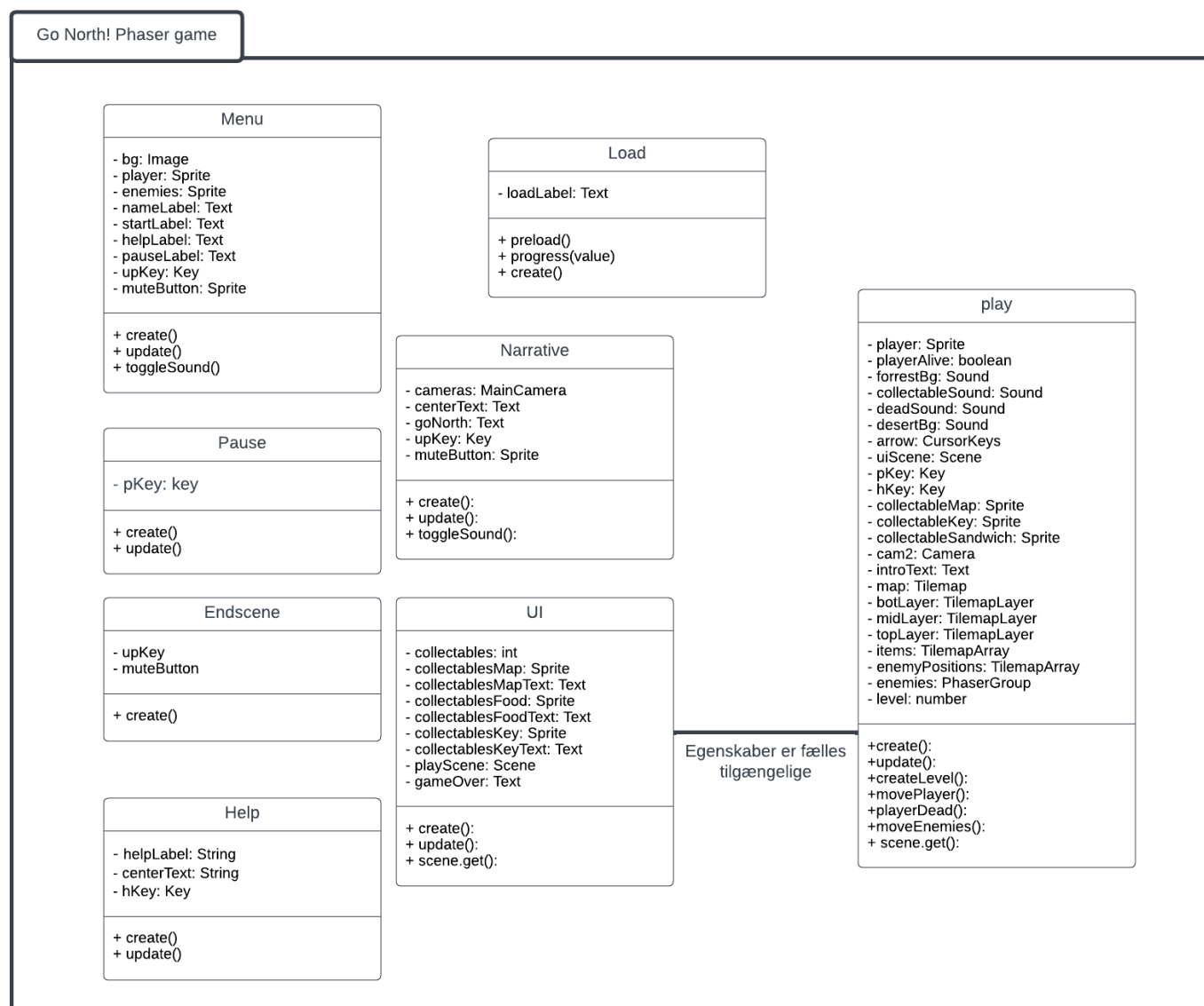
For at udarbejde vores baner til spillet, har vi valgt at benytte et program der hedder Tiled. Tiled er et program der gør det muligt at hente store designpakker, såkaldte tilesets, som indeholder mange små tiles, der kan sammensættes til at udarbejde et tilemaps. (Palef, 2021)

Selve Tiled-programmet gør det herefter muligt for os at placere de enkelte tiles i et tileset på et brugerdefineret map, uden at vi selv skal placere koordinater eller lignende. Selve processen i at udarbejde banen bliver mere visuel, og minimerer derfor tiden brugt på banedesign. Andre fordele ved at bruge Tiled er f.eks. at vi kan tildele de enkelte tiles egenskaber, samt at vi kan placere flere forskellige lag, for at skabe dybde i banen.

### Spritesheets

Lasse Jensen

Vi har valgt at benytte spritesheets til at designe og animere vores hovedperson og vores fjender. Et spritesheet består af mange mindre sprites, som hver især udgør en frame af den samlede animation, eller det samlede billede. Det gør det muligt for os at få vores ellers statiske billeder til at fremstå som en animation.



Figur 3 - Pakke Diagram

Vores spil består af otte klasser, som alle nedarver fra klassen *Phaser.Scene*. Vi har valgt ikke at inkludere denne klasse her. Spillet er organiseret i et pakke-diagram, der viser alle de klasser, der bliver brugt, samt deres egenskaber og metoder. Disse egenskaber kan ikke tilgås på tværs af hinanden, hvilket betyder, at der er indkapsling. Der er dog en forbindelse mellem "play"-klassen og "ui"-klassen ved hjælp af *scene.get()*-metoden, hvilket gør det muligt for dem at tilgå hinandens egenskaber. Dette bliver også forklaret senere i vores kodeforklaring. Vi har valgt at bruge et pakke-diagram, da der i virkeligheden ikke er nogen nedarvning eller anden forbindelse mellem de ovenstående forskellige klasser. (figur 3).

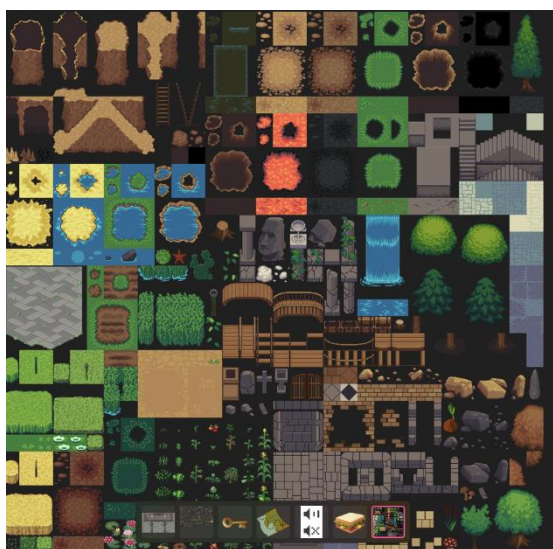
### Programmering

#### Maps

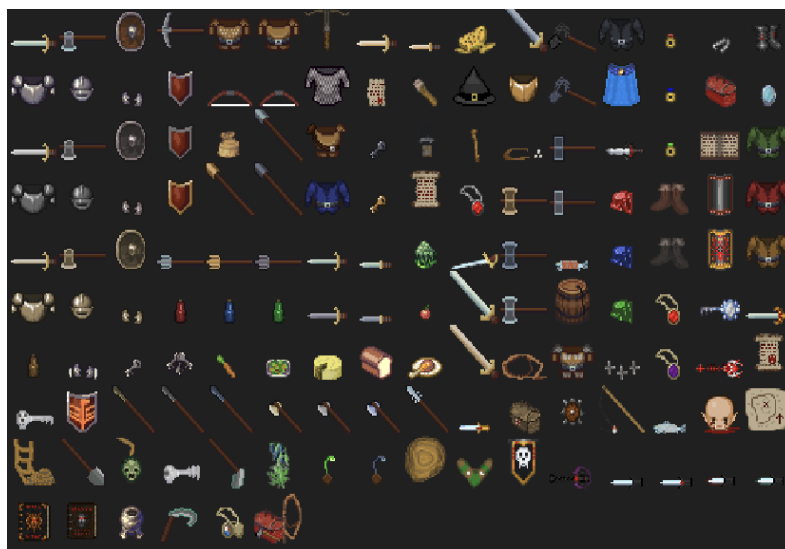
Lasse Jensen

Som nævnt tidligere i vores metodevalg, har vi valgt at benytte os af programmet Tiled, for at kreere vores bane og håndtere forskellige layers, herunder et layer til vores fjender. Jeg vil i det følgende afsnit forklare mere dybdegående hvordan vi har benyttet Tiled, som en grundsten for vores spilopbygning.

Vores udgangspunkt var at designe 3 forskellige baner, med hver deres tema, skov, ørken og vulkan. For at kunne skabe vores baner har vi fundet et tileset, der indeholder de elementer som vi gerne vil benytte i vores baner. Som nævnt tidligere er Tilesets en samling af mange mindre tiles, som vi kan markere og indsætte enkeltvis på vores map. (Jest array, 2019) (figur 4-5)



Figur 5 - Tileset Terrain\_atlas



Figur 4 - Tileset Items

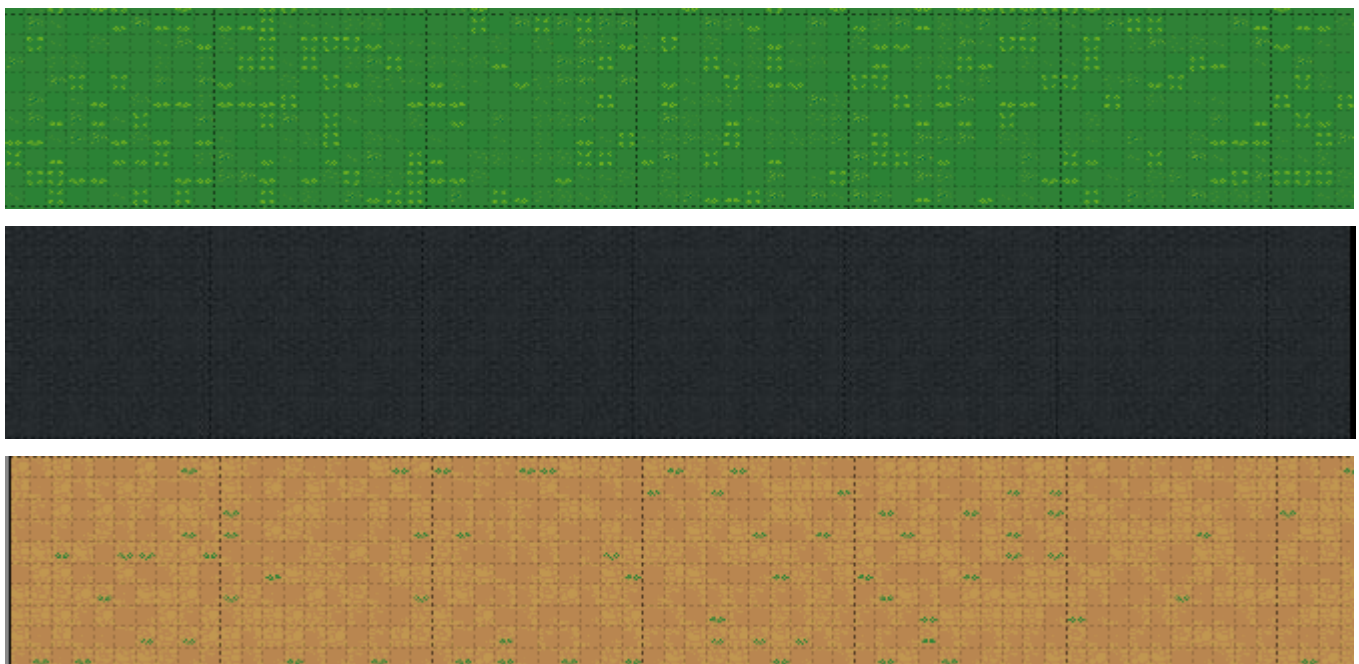
For at skabe dybde i banens design har vi lavet 3 forskellige layers. "top" som fungerer som det øverste lag, "mid" som beskriver det midterste lag, og "bot" som beskriver det nederste lag. Det nederste lag fungerer som baggrunden for banernes opbygning, og ved at benytte os af de to øvrige lag, kan vi placere forskellige tiles ovenpå dette lag. Det skaber et visuelt indtryk af at banen er i flere dimensioner. Det gør os også i stand til at få vores Sprites til at bevæge sig bag ved forskellige tiles, f.eks. træer. (figur 6).

## Anvendt Programmering



Figur 6 - Tilelayers & ObjectLayers

”Bot”-layer fungerer som vores canvas, som vi kan placere vores andre tiles på. Da hver bane har deres eget tema, har banerne således hver deres ”bot”-layer. (figur 7)




Figur 7 - "bot"-layers

Når vores ”bot”-layer er på plads, kan vi på de to øvrige lag placere tiles der repræsenterer vand, træer, vægge, murværker eller lignende. (figur 8).



Figur 8 - Tiled - Map1

For at undgå at vores spiller eller fjenderne kan vandre på tværs af de tiles vi har placeret, f.eks. vand eller mure, som gerne skulle obstruere spilleren, har vi i Tiled mulighed for at tildele hver enkel tile en egenskab. I dette tilfælde har vi f.eks. givet muren en egenskab, 'collides'. (figur 9).



Property	Value
▼ Tile	
ID	917
Class	
Width	32
Height	32
Probability	1,000
▶ Image Rect	[(672, 896), 32 x 32]
▼ Custom Properties	
collides	<input checked="" type="checkbox"/>

Figur 9 - Egenskab: Collides



## Anvendt Programmering

Hvordan denne egenskab implementeres og tilgås, vil vi gennemgå senere i opgaven.

Vi forudså nogle problemstillinger med det at designe et 2D-spil, hvor vores tilesets var relativt lille. Vi ville for eksempel få problemer med at benytte disse tiles på alle lag, hvor man på nogle tidspunkter måske ville ønske at spilleren skulle gå henover disse tiles, og på andre tidspunkter skulle blokeres. Derfor har vi kun tildelt egenskaben 'collides' på "mid" – og "top" layer. Når vi senere kalder egenskaben, vil den kun reagere, hvis egenskaben forefindes på de to lag, og uagtet om de samme tiles benyttes på "bot"-layer.

Foruden vores 3 tile-layers, har vi også oprettet 2 object-layers. Et "enemies"-layer der benyttes til at placere koordinater på vores fjender, og et forberedt "items"-layer som benyttes til de objekter som vi evt. ville have i banen.

Tiled gør det muligt at placere koordinater direkte på banen, hvorefter vi kan kalde dem senere i vores kode, og derved indhente et array med fjendens positioner. Igen fungerer dette visuelt godt, da vi ikke skal udregne hver fjendes position, men blot placere punkter i "enemies" object-layer. Implementeringen af dette vil blive gennemgået senere i opgaven. (figur 10).



Figur 10 - Objectlayer - Enemies

## Anvendt Programmering

### Karakterer

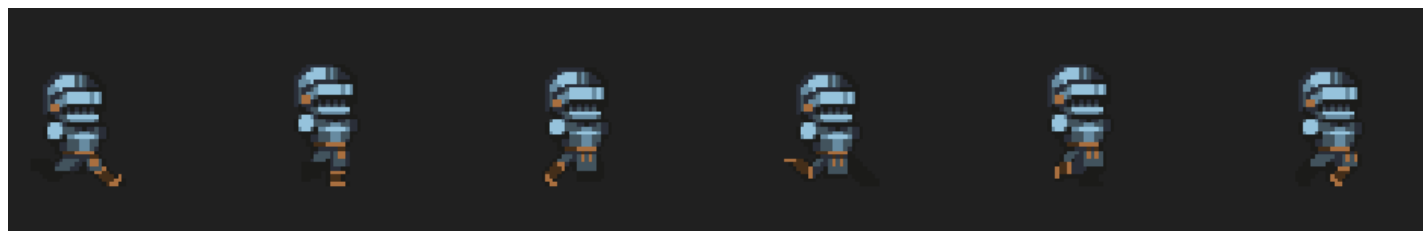
Lasse Jensen

Vores spil består af to karakter-typer, vores hovedperson som defineres ”knight”, og en fjende som defineres som ”Zombie”, da vi i vores første versioner af spillet, benyttede zombie-sprites. For at holde os til spillets visuelle udtryk blev zombierne dog erstattet af Orker, men selve definitionen er fastholdt som zombies.

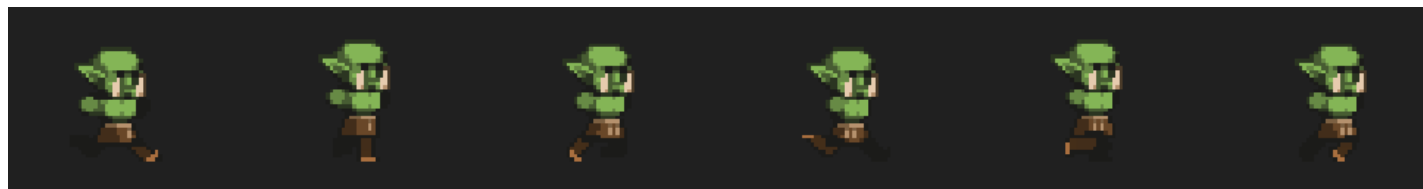
Vores karakterer er hentet ved at benytte et spritesheet. Spritesheet’et indeholder flere forskellige frames, vi kalder i load.js. (figur 11-13)

```
2  preload() {
3    this.load.spritesheet('knight', 'assets/Heroes/Knight/Run/Run-Sheet.png', {
4      frameWidth: 64,
5      frameHeight: 64,
6    });
7    this.load.spritesheet('zombie', 'assets/Enemy/Run-Sheet.png', {
8      frameWidth: 64,
9      frameHeight: 64,
10   });
11 }
```

Figur 11 - Load.spritesheet



Figur 12 - Spritesheet knight



Figur 13 - Spritesheet Zombie / Orc

Vi definerer frameHeight og frameWidth, på spritesheet jf. filernes dimensioner. Herefter benytter vi *this.anims.create-metoden* til at definere hvor høj en framerate vores animation skal have, samt repeat -1, der gør at vores animation gentager sig indtil stoppet. (figur 14 og 15).

```
64  this.anims.create({
65    key: 'move',
66    frames: 'knight',
67    frameRate: 8,
68    repeat: -1,
69  });
```

Figur 14 - Animation

## Anvendt Programmering

```
81     this.anims.create({
82       key: 'moveZombie',
83       frames: 'zombie',
84       frameRate: 8,
85       repeat: -1,
86     });
```

Figur 15 - Framerate

I vores play.js kalder vi vores key: "move", som vi definerede i load.js, under metoden "movePlayer()". Denne key refererer til de forskellige animationer, hvad enten der er spilleren eller fjendens idle-tilstand, eller deres bevægelser på banen. (figur 16)

```
253 movePlayer() {
254   if (this.playerAlive == true) {
255     if (this.arrow.left.isDown) {
256       this.player.setVelocityX(-300);
257       this.player.setFlipX(true);
258       this.player.anims.play("move", true);
259       this.player.setOffset(25, 38);
```

Figur 16 - metode "movePlayer()"

## Level-design

Lasse Lotzkat Thorsen

Vi har valgt at skifte banen i spillet ved at indlæse tre forskellige JSON-filer, som hver repræsenterer en ny verden, som vores spiller kan bevæge sig rundt i. Dette gøres ved hjælp af vores *createLevel()*-metode. Vi har valgt denne tilgang for at undgå gentagelse af kode flere gange, såsom ved at skifte scene. Da et sceneskift "nulstiller" al kode, mener vi, at vi ved at skifte JSON-filen i stedet for at skifte scene kan minimere mængden af kode og dermed holde koden letlæselig.

For at sikre at banerne ikke ligger i lag ovenpå hinanden, benytter vi metoden *createLevel(level)*.

*If (this.map != undefined){this.map.destroy();}* bliver kaldt ved start. Den tjekker om der er defineret et map. Ved starten af spillet vil der ikke være defineret noget map, hvorfor der ikke er noget at destruere, og map1 bliver defineret. Ved næste kald, er map1 således defineret og dette bliver nu destrueret, hvorefter map2



## Anvendt Programmering

bliver defineret. Afhængig af hvilket argument der bliver sendt med *createLevel()*-metoden skabes en af 3 verdener. (figur 17)

```
166 // creates a map based on the level parameter 1,2 or 3
167 createLevel(level) {
168   if (this.map != undefined) {
169     this.map.destroy();
170   }
171
172   if (level == 1) {
173     this.cameras.main.setBackgroundColor('#2B8235');
174     this.map = this.make.tilemap({ key: 'map1' });
175     this.collectableMap = this.physics.add.sprite(1519, 3330, 'map');
176     this.collectableMap.setScale(0.025);
177     this.collectableKey = this.physics.add.sprite(273, 1200, 'key');
178     this.collectableKey.setScale(0.01);
179     this.collectableSandwich = this.physics.add.sprite(1583, 1887, 'sandwich');
180     this.collectableSandwich.setScale(0.015);
181     this.forrestBg.play('', 0, 1, true);
182     this.forrestBg.setVolume(0.2);
183     this.level = 1;
184   }
185
186   else if (level == 2) {
187     this.player.setX(850);
188     this.player.setY(4044);
189     this.cameras.main.setBackgroundColor('#ff00ff');
190     this.map = this.make.tilemap({ key: 'map2' });
191     this.collectableMap = this.physics.add.sprite(1773, 801, 'map');
192     this.collectableMap.setScale(0.025);
193     this.collectableKey = this.physics.add.sprite(1510, 2280, 'key');
194     this.collectableKey.setScale(0.01);
195     this.collectableSandwich = this.physics.add.sprite(210, 580, 'sandwich');
196     this.collectableSandwich.setScale(0.015);
197     this.forrestBg.stop();
198     this.desertBg.play('', 0, 1, true);
199     this.desertBg.setVolume(0.2);
200     this.level = 2;
201   }
202
203   else if (level == 3) {
204     this.player.setX(914);
205     this.player.setY(4064);
206     this.cameras.main.setBackgroundColor('#5865F2');
207     this.map = this.make.tilemap({ key: 'map3' });
208     this.collectableMap = this.physics.add.sprite(1008, 3071, 'map');
209     this.collectableMap.setScale(0.025);
210     this.collectableKey = this.physics.add.sprite(599, 1755, 'key');
211     this.collectableKey.setScale(0.01);
212     this.collectableSandwich = this.physics.add.sprite(144, 709, 'sandwich');
213     this.collectableSandwich.setScale(0.015);
214     this.desertBg.play('', 0, 1, true);
215     this.desertBg.setVolume(0.2);
216     this.level = 3;
217   }
218 }
```

Figur 17 - Metode "createLevel()"

En af de 3 forskellige JSON filer er blevet valgt ved hjælp af *this.make.tilemap()*-metoden og nedenstående kode efterfølger, hvorefter vores objekter; mad, nøgle og kort, bliver indlæst som separate sprites. Vi kunne have valgt at indlæse vores objekter i Tiled, hvorved vi også nemmere ville kunne placere dem direkte i programmet, men for at spillet kan genkende hvilke objekter man som spiller samler op, er vi nødsaget til at indlæse dem direkte i koden, hvorved vi kan se, præcist hvilket objekt der interageres med. I modsætning til vores fjender, som er placeret i Tiled på et object-layer, kan vi her ikke se, hvilken fjende der interageres med når spilleren har kontakt.

De forskellige tilesets, "terrain\_atlas" og "itemset" bliver bragt ind på hvert lag, "bot", "mid" og "top", med *"this.map.createLayer('top(mid,bot)',[terrain, itemset])*. Vi kalder vores fjenders positioner med object-layeret *"this.enemyPosition = this.map.getObjectLayer('enemies').objects;"* som vi har defineret i Tiled.

## Anvendt Programmering

Vi looper herefter igennem fjendernes positioner på object-layer for at placere modstandere på kortet, samt tilføje dem til en gruppe kaldet *this.enemies*. (figur 18).

```
117 const terrain = this.map.addTilesetImage("terrain_atlas", "terrain");
118 const itemset = this.map.addTilesetImage("items");
119 this.botLayer = this.map.createLayer("bot", [terrain], 0, 0).setDepth(-2);
120 this.midLayer = this.map.createLayer("mid", [terrain, itemset], 0, 0).setDepth(-1);
121 this.topLayer = this.map.createLayer("top", [terrain, itemset], 0, 0);
122 this.items = this.map.getObjectLayer('items').objects;
123
124 //Collects the enemy positions from the object layer
125 this.enemyPositions = this.map.getObjectLayer('enemies').objects;
126
127 this.enemies = this.physics.add.group();
128 // looping through object layer positions and adding enemies to the group
129 for (let i = 0; i < this.enemyPositions.length; i++) {
130   let enemy = this.enemies.create(this.enemyPositions[i].x, this.enemyPositions[i].y, 'zombie');
131   enemy.body.setCollideWorldBounds(true);
132   enemy.body.setImmovable(); // what does it do?
133   enemy.body.setBounce(1, 1);
134   enemy.followingPlayer = false;
135   enemy.anims.play('moveZombie', true);
136   enemy.setDepth(-1);
137   enemy.setSize(25, 35);
138   enemy.setOffset(20, 30);
139 };
```

Figur 18 - Generering af fjenden

## JavaScript

Spillet består af 9 JavaScript filer der hver især (undtagen game.js) repræsenterer en scene i spillet. Derudover findes en index.html fil der viser spillet og dens indhold i en browser.

### Game.js

Lasse Lotzkat Thorsen

Phaser JS konfigureres i game.js filen, og her defineres spillets dimensioner, baggrundsfarve, hvilken slags physics (arcade) spillet skal have og hvor spillets skal vises i HTML filen index.html.

Efter konfiguration tilføjes scenerne "load", "menu", "narrative", "play", "pause", "help", "ui", og "endscene" ved hjælp af *.add()-metoden*.

Til sidst startes "load"-scenen ved hjælp af *.start()-metoden*. (figur 19).

```
11 game.scene.add('load', Load);
12 game.scene.add('menu', Menu);
13 game.scene.add('narrative', Narrative);
14 game.scene.add('play', Play);
15 game.scene.add('pause', Pause);
16 game.scene.add('help', Help);
17 game.scene.add('ui', UI);
18 game.scene.add('endscene', Endscene);
19
20 game.scene.start('load');
```

Figur 19 - game.js - add

## Anvendt Programmering

### Load.js

Lasse Lotzkat Thorsen

I load-scenens *preload()*-metode indlæses alle de assets, som spillet skal kunne tilgå, herunder spritesheets, billeder, lydfiler og tilemap JSON filer. De indlæses ved hjælp af metoderne *load.spritesheet()*, *load.image()*, *load.audio()* og *load.tilemapTiledJSON*. (figur 20).

```
42 this.load.image("terrain", "../assets/terrain_atlas.png");
43 this.load.image("items", "../assets/items.png");
44
45 this.load.image("bgImageMenu", "../assets/bgImageMenu.png");
46
47 this.load.tilemapTiledJSON("map1", "../assets/map1.json");
48 this.load.tilemapTiledJSON("map2", "../assets/map2.json");
49 this.load.tilemapTiledJSON("map3", "../assets/map3.json");
```

Figur 20 – load.js – load files

I load-scenens *create()*-metode skabes animationer der bliver brugt i spillet af hhv. spillerens sprite samt fjenden. Når load-scenen færdiggør indlæsningen, startes menu-scenen. (figur 21).

```
41 create() {
42
43     this.anims.create({
44         key: 'move',
45         frames: 'knight',
46         frameRate: 8,
47         repeat: -1,
48     });
49     this.anims.create({
50         key: 'idle',
51         frames: 'idleknight',
52         frameRate: 10,
53         repeat: -1,
54     });
55     this.anims.create({
56         key: 'dead',
57         frames: this.anims.generateFrameNumbers('deadknight', { start: 0, end: 4 }),
58         frameRate: 5,
59     });
60 }
```

Figur 21 - load.js animationer

### Menu.js

Lasse Lotzkat Thorsen

*create()*-metoden i menu.js skaber en baggrund, en sprite for spilleren, en sprite for modstanderen. *This.player-anims.play("idle", true)* animerer vores sprites til at benytte sig af deres "idle" animation. Yderligere tilføjes forskellige tekstelementer der instruerer i, hvordan spillet startes, samt titlen på spillet. For at skabe lidt dynamik i menuen har vi valgt at tilføje tweens til begge tekstelementer. Tweens er en mekanisme til at skabe og kontrollere animationer. Den tillader dig at definere en overgang mellem to tilstande for et objekt og en bestemt varighed. Titlen har et koordinatpunkt -50 på y-aksen, og bevæger sig mod koordinatpunktet 300 på Y-aksen på 1 sekund. Den har en "ease" der hedder "bounce.out". (figur 22).

## Anvendt Programmering

```
14 this.player = this.physics.add.sprite(100, 300, "knight");
15 this.player.setScale(3);
16 this.player.anims.play("idle", true);
17
18 this.enemies = this.physics.add.sprite(1000, 300, "zombie");
19 this.enemies.setScale(3);
20 this.enemies.setFlipX(true);
21 this.enemies.anims.play("idlezombie", true);
22
23 const nameLabel = this.add.text(550, -50, 'Go North!', { font: '70px MedievalSharp', fill: '#fff' });
24 nameLabel.setOrigin(0.5, 0.5);
25 this.tweens.add({
26   targets: nameLabel,
27   y: 300,
28   duration: 1000,
29   ease: 'bounce.out',
30 });
31
```

Figur 22 - menu.js - animation & tweens

I *create()-metoden* "mappes" op pilen til *this.upKey* som derefter kan benyttes i *update()-metoden*. *update()-metoden* tjekker om op pilen bliver tastet og starter derefter en ny scene kaldet Narrative.

### Narrative.js

Lasse Lotzkat Thorsen

I *create()-metoden* indstilles baggrundsfarven til sort og en tekst der forklarer spillets præmisser sættes til at fade in vha. en tween der langsomt ændrer alpha-værdien fra 0 til 1 (figur 23).

```
10 centerText.setOrigin(0.5);
11 centerText.alpha = 0;
12
13 this.tweens.add({
14   targets: centerText,
15   alpha: 1,
16   ease: 'Linear',
17   duration: 2000
18 });
```

Figur 23 - narrative.js

Igen tjekkes der i *update()-metoden* om op pilen presses ned, og starter derefter scenen Play.

### Play.js

Lasse Lotzkat Thorsen

I *create()-metoden* skabes spilleren (*this.player*) og der tilføjes lyde med *this.sound.add()*. Derefter bliver inputs fra keyboard gemt som *this.arrow* og banen bliver skabt vha. metoden *createLevel()*.

*createLevel()-metoden* skaber en bane ved, afhængig af hvilket argument der bliver sendt, at skabe lag defineret i programmet Tiled. Disse lag inkluderer topLayer, midLayer, botLayer, items og enemies hvortil de sidste to er objekt lag (se figur 24 hvor bane 3 genereres). Ved at hente disse lag kan vi skabe en verden spilleren kan bevæge sig rundt i, samt indlaste "collectables" på bestemte positioner.

```
188     else if (level == 3) {
189         this.player.setX(914);
190         this.player.setY(4064);
191         this.cameras.main.setBackgroundColor('#5865F2');
192         this.map = this.make.tilemap({ key: 'map3' });
193         this.collectableMap = this.physics.add.sprite(1008, 3071, 'map');
194         this.collectableMap.setScale(0.025);
195         this.collectableKey = this.physics.add.sprite(599, 1755, 'key');
196         this.collectableKey.setScale(0.01);
197         this.collectableSandwich = this.physics.add.sprite(144, 709, 'sandwich');
198         this.collectableSandwich.setScale(0.015);
199         this.desertBg.play('', 0, 1, true);
200         this.desertBg.setVolume(0.2);
201         this.level = 3;
202     };
203
204     const terrain = this.map.addTilesetImage("terrain_atlas", "terrain");
205     const itemset = this.map.addTilesetImage("items");
206     this.botLayer = this.map.createLayer("bot", [terrain], 0, 0).setDepth(-2);
207     this.midLayer = this.map.createLayer("mid", [terrain, itemset], 0, 0).setDepth(-1);
208     this.topLayer = this.map.createLayer("top", [terrain, itemset], 0, 0);
209     this.items = this.map.getObjectLayer('items').objects;
210
211     //Collects the enemy positions from the object layer
212     this.enemyPositions = this.map.getObjectLayer('enemies').objects;
```

Figur 24 - play.js

Efter banen er skabt, loopes der igennem positionerne defineret af objekt-laget *this.enemyPositions*, der returnerer et array af koordinater, og der placeres en modstander i en Phaser gruppe på hver position defineret (se figur 25).

*"setCollideWorldBounds(true)"* gør at fjenderne ikke kan bevæge sig ud over banens grænser. *"followingPlayer = false"*, er en variabel der bestemmer om vores tween, der får vores fjender til at følge efter spilleren hvis disse når ind for en bestemt afstand, aktiveres. Dette sættes som standard til false, da vi ikke vil have at alle fjender flokkes om spilleren ved spillets start. Dette beskrives nærmere senere i opgaven.

Vi sætter *enemy.setDepth(-1)*, da vores fjender ellers ville vandre ovenpå alle lag i vores baner. Metoden *"setOffset"* beskrives senere i opgaven.

```
224     //Collects the enemy positions from the object layer
225     this.enemyPositions = this.map.getObjectLayer('enemies').objects;
226
227     this.enemies = this.physics.add.group();
228     // looping through object layer positions and adding enemies to the group
229     for (let i = 0; i < this.enemyPositions.length; i++) {
230         let enemy = this.enemies.create(this.enemyPositions[i].x, this.enemyPositions[i].y, 'zombie');
231         enemy.body.setCollideWorldBounds(true);
232         enemy.body.setBounce(1, 1);
233         enemy.followingPlayer = false;
234         enemy.anims.play('moveZombie', true);
235         enemy.setDepth(-1);
236         enemy.setSize(25, 35);
237         enemy.setOffset(20, 30);
238     };
```

Figur 25 - play.js - enemy positions

## Anvendt Programmering

Da banen samt modstanderne nu er skabt, startes en separat scene, kaldet UI, vha. *this.scene.launch()*-metoden, som fungerer som et konstant overliggende lag på spillet. Dvs. spillerens User Interface. (figur 26)

Denne scene uddybes senere.

```
24 // starts UI scene with collectables
25 this.scene.launch('ui');
```

Figur 26 - ui.js

Til sidst i *create()*-metoden justeres scenens kamera så det følger spilleren, samt viser kun et udsnit af den samlede bane. Metoden "setViewport" bestemmer størrelsen for, hvor meget af banen der vises ad gangen. Da kameraet følger spilleren ved hjælp af metoden *startFollow(this.player)*, er vi nødt til at begrænse kameraet, det dette ellers vil vise områder der er uden for banernes område. Dette skaber derved illusionen om en kæmpe verden (se figur 27).

```
33 // sets the camera on the player
34 this.physics.world.setBounds(0, 0, 2048, 4096);
35 this.cameras.main.setBounds(0, 0, 2048, 4096);
36 this.cameras.main.setViewport(0, 0, 1100, 600);
37 this.cameras.main.startFollow(this.player);
38 this.cameras.main.setZoom(1.5);
```

Figur 26 - play.js - cameras

I *update()*-metoden tjekkes efter hvorvidt spilleren ønsker at pause spillet (press P) eller have hjælp (press H). Her kaldes metoden *movePlayer()* også for at give muligheden for bevægelse af *this.player* (se figur 28).

```
246 movePlayer() {
247     if (this.playerAlive == true) {
248         if (this.arrow.left.isDown) {
249             this.player.setVelocityX(-100);
250             this.player.setFlipX(true);
251             this.player.anims.play('move', true);
252             this.player.setOffset(25, 38);
253         } else if (this.arrow.right.isDown) {
254             this.player.setVelocityX(100);
255             this.player.setFlipX(false);
256             this.player.anims.play('move', true);
257             this.player.setOffset(25, 38);
258         } else if (this.arrow.up.isDown) {
259             this.player.setVelocityY(-100);
260             this.player.anims.play('move', true);
261             this.player.setOffset(25, 38);
262         } else if (this.arrow.down.isDown) {
263             this.player.setVelocityY(100);
264             this.player.anims.play('move', true);
265             this.player.setOffset(25, 38);
266         } else {
267             this.player.setVelocityX(0);
268             this.player.setVelocityY(0);
269             this.player.anims.play('idle', true);
270             this.player.setOrigin(0.5, 1);
271             this.player.setOffset(8, 7);
272         }
273     };
274 }
```

Figur 27 - play.js - movePlayer()

## Anvendt Programmering

*this.player's* fremdrift ændres afhængig af hvilken piletast er trykket ned og samtidig afspilles en passende animation. Da de Spritesheets vi har brugt til disse animationer havde forskellige størrelser af "hitboxe", har vi været nødsaget til at ændre størrelsen ved hjælp af *.setSize()* og *.setOffset()-metoderne* ( figur 29-30).



Figur 29 - Hitbox før



Figur 28 - Hitbox efter

Ydermere tjekkes der i *update()-metoden* for kollision mellem spiller/modstander og verden ved hjælp af *this.physics.collide()-metoden*. Vi har ved hjælp af programmet Tiled, tildelt visse elementer af tilemap'et egenskaberne *collide/collides*, for på den måde at undgå den, i vores tilfælde, lidt langsommelige måde at nedskrive samtlige Global ID's i *collide* metoden (figur 31 / 32).

```
237 // sets collision based on property given to the tile i Tiled software
238 this.topLayer.setCollisionByProperty({ collide: true });
239 this.topLayer.setCollisionByProperty({ collides: true });
240 this.midLayer.setCollisionByProperty({ collide: true });
241 this.midLayer.setCollisionByProperty({ collides: true });
```

Figur 3130 - play.js - collide(s)

```
92 //adds a collider with the objects in the world
93 this.physics.collide(this.player, this.topLayer);
94 this.physics.collide(this.player, this.midLayer);
```

Figur 31 - play.js - collides player/objects

Derudover tjekkes der hvorvidt *this.player* og modstanderen overlapper og hvis de gør startes metoden *playerDead()*.

Metoden "*playerDead()*" kaldes, når spilleren dør, hvilket udløser lydeffekter og animationer, nulstiller UI scenen og skifter til menu-scenen.

For at sikre metoden ikke bliver kaldt gentagne gange, benytter vi "*if (!this.playerAlive) {return;}.*" Uden denne variabel vil vores "*playerDead()*" metoden kaldes gentagne gange, hvor lyde vil blive afspillet så længe at fjenderne og vores spiller har et overlap. Når spilleren er død, og animationen er blevet afspillet ved brug af

## Anvendt Programmering

`"anim.play("dead", true);"`, iværksættes en tween der gør at spilleren forsvinder efter 2 sekunder, og bliver skaleret ned. Callback'en `"onComplete: () =>"` iværksætter sceneskift fra `play.js` og `ui.js` til `menu.js`. (figur 33)

```
131 ~ playerDead() {  
132   if (!this.playerAlive) {  
133     return;  
134   };  
135   this.playerAlive = false;  
136   this.forrestBg.stop();  
137   this.deadSound.play();  
138   this.player.setVelocityX(0);  
139   this.player.setVelocityY(0);  
140   this.uiScene.collectables = 0;  
141   this.player.ans.play('dead', true);  
142   this.time.addEvent({  
143     delay: 2000,  
144     callback: () => {  
145       this.player.alpha = 0;  
146       this.tweens.add({  
147         targets: this.player,  
148         alpha: 1,  
149         duration: 20,  
150         repeat: 19  
151       });  
152       this.tweens.add({  
153         targets: this.player,  
154         scaleX: 0,  
155         scaleY: 0,  
156         duration: 1000,  
157         ease: 'Power2',  
158         completeDelay: 500,  
159         onComplete: () => {  
160           this.scene.stop('play');  
161           this.scene.stop('ui');  
162           this.scene.start('menu');  
163         }  
164       });  
165     }  
166   });  
167 };
```

Figur 32 - Metode "playerDead()"

For at få modstanderne til at følge spilleren indenfor en bestemt radius, tjekkes der ved brug af `"Phaser.Math.Distance.BetweenPoint(this.player, enemy) < 100"` hvorvidt spilleren er tilpas tæt på fjenderne for at iværksætte `"enemy.followingPlayer"`, hvor fjenderne begynder at følge efter spilleren. `"duration"` bestemmer tiden fra variablen iværksættes til at fjenden rammer spilleren. Jo højere tal der står her, jo hurtigere vil fjenden ramme spilleren. Hvis modstanderen følger spilleren, afspilles en tween der ændrer modstanderens X og Y koordinater til `this.player.x` og `this.player.y`.

(figur 34).

```
114 // below loops through this.enemies array and assign whether or not to follow  
115 this.enemies.getChildren().forEach((enemy) => {  
116   if (Phaser.Math.Distance.BetweenPoints(this.player, enemy) < 100) {  
117     enemy.followingPlayer = true;  
118     this.tweens.add({  
119       targets: enemy,  
120       y: this.player.y,  
121       x: this.player.x,  
122       duration: 1000,  
123     });  
124     if (enemy.x < this.player.x) {  
125       enemy.setFlipX(false);  
126     } else {  
127       enemy.setFlipX(true);  
128     }  
129   }  
130   else {  
131     enemy.followingPlayer = false;  
132   }  
133 }  
134 });
```

Figur 33 - `play.js` - `enemy.followingPlayer`



Hvis spilleren ikke er i nærheden af en modstander, afspilles følgende metode hvert 3. sekund (figur 35 & 36).

```
41 // move enemies
42 this.timedEvent = this.time.addEvent({
43   delay: 3000,
44   callback: this.moveEnemies,
45   callbackScope: this,
46   loop: true
47 });
```

Figur 34 - play.js callback "moveEnemies"

Denne metode finder et tilfældigt nummer mellem 1 og 435, hvor der udvælges en, af os, defineret case. Hver case dikterer hvilken retning fjenden skal bevæge sig i, samt om fjendens sprites x-akse skal flippes. Denne metode gør at vores fjender vandrer rundt i tilfældige mønstre på kortet. (Westover, 2023) (figur 36)

```
318 // moves the enemies around unless they are following the player
319 moveEnemies() {
320   this.enemies.getChildren().forEach((enemy) => {
321
322     if (!enemy.followingPlayer) {
323       const randomNumber = Math.floor(Math.random() * 4) + 1;
324       switch (randomNumber) {
325         case 1:
326           enemy.body.setVelocityX(50);
327           enemy.setFlipX(false);
328           break;
329         case 2:
330           enemy.body.setVelocityX(-50);
331           enemy.setFlipX(true);
332           break;
333         case 3:
334           enemy.body.setVelocityY(50);
335           break;
336         case 4:
337           enemy.body.setVelocityY(-50);
338           break;
339         default:
340           enemy.body.setVelocityX(50);
341       }
342     }
343   })
344 }
```

Figur 36 - play.js "moveEnemies()"

UI-scenen tilføjer forskellige sprites og tekst til scenen, der repræsenterer collectables i spillet. Den lytter efter events, der udsendes af play-scenen, specifikt når en collectable samles op af spilleren. Når en collectable genstand samles op, fjernes den tilsvarende sprites sorte tint fill i øverste venstre hjørne, og antallet af collectable items, som spilleren har, forøges. Når antallet af collectables overstiger 2, og spilleren har nået en bestemt position på y-aksen, <50, udsendes en event for at skifte til et andet level. (figur 37).

Når kortet samles op første gang i level 1, bliver cam2 etableret. Cam2 fungerer som vores minimap. Efterfølgende, når banen skiftes, sætter vi alpha på cam2 til 0, ved *"this.playScene.cam2.setAlpha(0);"*. Kortet forsvinder aldrig, men er ikke synligt igen, før spilleren samler kortet op igen.

```
66     if (this.collectables > 2 && this.playScene.player.y < 50) {
67         if (this.playScene.level == 1) {
68             this.events.emit('changeToLevel2');
69             this.playScene.cam2.setAlpha(0);
70             this.collectables = 0;
71             this.collectablesMap = this.physics.add.sprite(30, 30, 'map').setTintFill('#000000').setScale(0.03);
72             this.collectablesFood = this.physics.add.sprite(80, 30, 'sandwich').setTintFill('#000000').setScale(0.015);
73             this.collectablesKey = this.physics.add.sprite(130, 30, 'key').setTintFill('#000000').setScale(0.015);
74
75         } else if (this.playScene.level == 2) {
76             this.events.emit('changeToLevel3');
77             this.playScene.cam2.setAlpha(0);
78             this.collectables = 0;
79             this.collectablesMap = this.physics.add.sprite(30, 30, 'map').setTintFill('#000000').setScale(0.03);
80             this.collectablesFood = this.physics.add.sprite(80, 30, 'sandwich').setTintFill('#000000').setScale(0.015);
81             this.collectablesKey = this.physics.add.sprite(130, 30, 'key').setTintFill('#000000').setScale(0.015);
82         } else {
83             this.scene.stop('play');
84             this.scene.stop('ui');
85             this.scene.start('endscene');
86         }
87     }
88 }
```

Figur 36 – ui.js - Collectables

Til sidst, hvis spilleren dør, vises teksten "GAME OVER" ved hjælp af en fade-in-animation. (figur 38).

```
65     // creates the game over text
66     if (this.playScene.playerAlive == false) {
67         this.tweens.add({
68             targets: this.gameOver,
69             alpha: 1,
70             ease: 'Linear',
71             duration: 3000
72         });
73     }
74 }
```

Figur 37 - ui.js - Game Over

Help og Pause-scenerne minder meget om hinanden. De har begge til formål at kunne manipuleres ved tryk på bestemte taster på keyboardet, for at generere tekst til spilleren. Nedenstående billedeksempler brugt er udelukkende fra help.js.

I *create()-metoden* bruges *add.text()-metoden*, som tager parametre der specificerer x og y akser ift. hvor på skærmen henholdsvis *'instructions'* og *'paused'* strengene, som også er det tredje parameter, skal stå. Font samt farve på fonten defineres ligeledes her.

*setOrigin()-metoden* bliver kaldet på variabelen som fra midten af strengen sætter teksten i midten af canvas. (figur 39).

```
2      create() {
3
4          const helpLabel = this.add.text(550, 50, 'instructions', { font: '50px MedievalSharp', fill: '#fff' });
5
6          helpLabel.setOrigin(0.5);
```

Figur 38 - help.js - helpLabel

*"this.cameras.main.setBackgroundColor()-metoden* bruges i help-scenen til at gøre baggrunden gråsort med en gennemsigtighed på 50%. Denne metode bruges ikke i pause-scenen. (figur 40).

```
8      // sets the scenes background to grey/black with an opacity of 50%
9      this.cameras.main.setBackgroundColor('rgba(11, 26, 12, 0.5)');
```

Figur 39 - help.js backgroundColor

I help-scenen oprettes variabelen *"centerText"* som skriver instruktionerne til spillet ude på canvas.

*setOrigin()-metoden* bliver her ligeledes kaldt på *"centerText"*, som placerer instruktionerne i midten af canvas. (figur 41).

```
12      const centerText = this.add.text(this.cameras.main.centerX, this.cameras.main.centerY,
13
14          `Use the arrow keys to move the knight
15
16          Avoid the Orcs trying to kill you
17
18          Continue going north, in order to advance to the next level
19
20          Before being able to enter the next level, collect the following:
21
22          - a map
23          - food
24          - a key
25
26          Press P to pause`,
27
28          { font: '25px MedievalSharp', fill: '#fff', wordWrap: { width: 500 } });
29
30      // set the text object's anchor to 0.5 to center it
31      centerText.setOrigin(0.5);
```

Figur 40 - help.js - tekst

## Anvendt Programmering

I begge scener defineres variabler som repræsenterer henholdsvis h-tasten og p-tasten på tastaturet. Ved hjælp af `"this.input.keyboard.addKey()"`-metoden. (figur 42).

```
31      this.hKey = this.input.keyboard.addKey('h');
```

Figur 41 - help.js - addKey

I `update()`-metoden tjekker et if-statement om tasten er nede. Hvis ja, får help/pause scenen en besked om at "sove" og play-scenen starter. (figur 43)

```
35      update() {  
36          if (this.hKey.isDown) {  
37              this.scene.sleep('help')  
38              this.scene.resume('play');  
39          }  
40      }
```

Figur 42 - help.js - update()

Vores test og evaluering er gennemført ved hjælp af et repræsentativt udtræk af vores målgruppe. (Se figur 44 – 47)

Testgruppens alder går fra 5 år op til 35 og testene er gennemført på samme måde.

Vores 4 testpersoner er blevet placeret foran computeren mens spillet er startet i dets hovedmenu. Herfra har vi vurderet og evalueret, hvordan testpersonerne har iværksat spillet, forstået instruktioner og scenarier, håndteret sværhedsgraden, samt deres evne til at spille spillet i sin helhed. Vi vil i det følgende afsnit beskrive nogle af de tilbagemeldinger som er kommet og vores egne vurderinger, samt hvordan vi eventuelt har valgt at løse de problemstillinger.



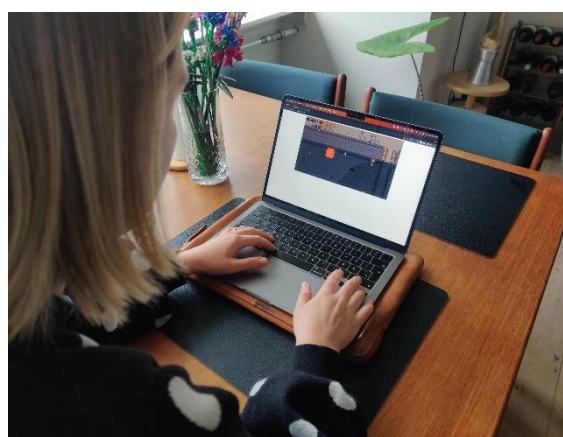
Figur 44 - Maria 35 år



Figur 43 - Sif 5 år



Figur 46 - Freja 8 år



Figur 45 - Sofie 25 år



Tilbagemeldinger omkring instruktioner har været, at spillets formål og hvordan spillet spilles, er uklare. Ligeledes er det blevet vurderet, at særligt for det yngre segment af testgruppen, var selve det at starte spillet op ikke intuitivt. Vi har derfor optimeret disse aspekter af vores spil som følger.

Første punkt var at ændre spillets startknap, da vi så, at det yngre segment intuitivt forsøgte at starte spillet ved brug af knapperne "SPACE" og "ENTER". Vi havde oprindeligt valgt at benytte "OP-PILEN", som den knap der iværksætter spillet, men efter evalueringen besluttede vi at "SPACE" var et mere intuitivt valg.

(figur 48/49).



Figur 47 - Startknap før

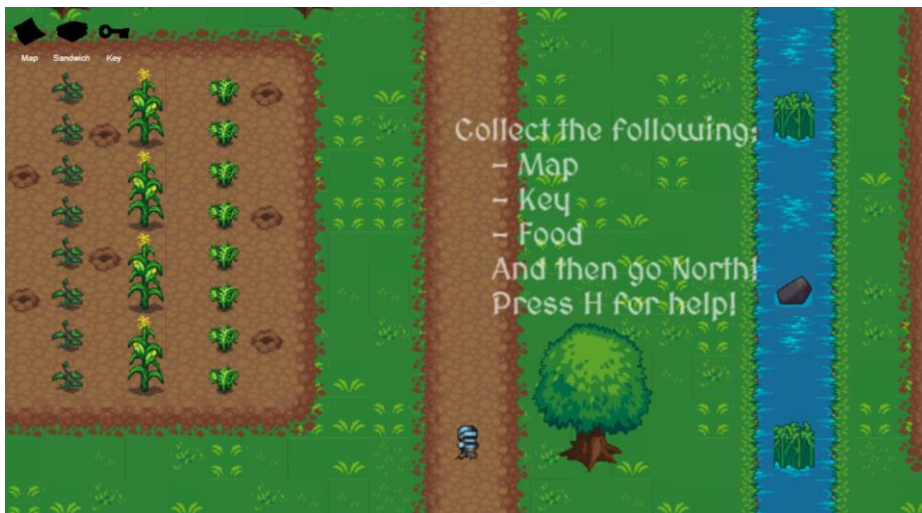


Figur 48 - Startknap efter

Oprindeligt havde vi valgt at spilleren udelukkende skulle kunne tilgå spillets instrukser ved hjælp af "H-tasten". Denne funktion stod kun beskrevet på hovedmenuen, men i spillet var muligheden for hjælp ved brug af denne metode ikke særlig tydelig. Efter vores test og evaluering, blev det derfor klart, at vi blev nødt til at rykke instruktionerne frem på skærmen ved opstart af spillet (figur 50 / 51).



Figur 49 - Help tekst før



Figur 50 - Help tekst efter

### Konklusion

Lasse Jensen, Lasse Lotzkat Thorsen, Christian M. Madsen

Vi har udviklet et top-down objective-based spil der involverer adskillige sprites, der er underlagt de regler vi har opstillet for fysikken i spillet, herunder brugen af collision. Vi har opbygget en verden ved brug af programmet Tiled, hvor vi har benyttet Tilesets til at genere flere forskellige tilemaps. Disse tilemaps, ved brug af camera styring, gør det muligt for spilleren at bevæge sig rundt i en omfattende verden, hvor fjenden vil forsøge at standse spilleren. Ved brug af tweens har vi gjort spillet mere visuelt stimulerende, interessant og spændende, for at ramme vores målgruppe ned til 5 år.

Vi har testet og evalueret vores projekt, samt iværksat forskellige korrigeringer på baggrund af den feedback som vi har fået fra vores repræsentative målgruppe.

I det efterfølgende afsnit har vi gjort os nogle refleksioner over projektet, samt hvordan vi kan optimere og forbedre spillet, hvad enten det er selve koden eller fremtidige muligheder for videreudvikling af konceptet.

### Refleksion

Lasse Jensen, Lasse Lotzkat Thorsen, Christian M. Madsen

#### Fjender der vandrer igennem vægge

For at orkerne i spillet skal kunne følge efter spilleren, har vi benyttet os af en tween. Problemet der derefter er opstået er, når tween'en startes, ophæves den såkaldte "*physics*" på vores ork, og de kan derfor ikke kolliderer med verdenen omkring dem indtil tween'en igen stoppes. Vi har forsøgt at inkludere regler for dette indeni tween ved f.eks. at inkludere *add.collider()*-metoden ved "*onStart*". Dog er dette ikke lykkedes.

#### `movePlayer()`

Metoden fungerer ikke helt optimalt da den ikke tager højde for at spilleren også kan trykke på både op-pilen og højre/venstre-pilen og derved generere en højere fremdrift end tiltænkt. Denne ekstra fremdrift gør det muligt at slippe væk fra fjenden hurtigere og nemmere end tiltænkt.

Vi har gjort os yderligere overvejelse hvordan denne problemstilling kunne løses, men har dog ikke nået en færdig løsning på nuværende tidspunkt.

På nuværende tidspunkt ser vi to eventuelle løsninger:

1. Forsøge at programmere et slags loft på fremdriften af spilleren.



## Anvendt Programmering

2. Tilføj yderligere else if – statements der kan tage højde for hvis 2 taster er trykket ned.

## Optimering af Tiled

Da vi konfigurerede Tiled, og valgte at benytte programmet, som en del af vores spiludvikling, valgte vi, kun at benytte 3 tilelayers, og 2 object-layers. Efterfølgende har vi vurderet at der er nogle problemstillinger ved kun at have de 3 tilelayers, som vi har brugt til at opbygge vores spilverden. Ved at benytte flere lag, ville vi kunne lave langt mere dybde i f.eks. skovarealer eller andre positioner, hvor der kræves lag på lag. Den nuværende opsætning betyder, at placering af tiles ovenpå hinanden, fjerner noget af det underliggende design.

Vores valg af tilesets har været udmærket til brug i vores spilverden, men det har også haft sine begrænsninger ift. de tematiske designmuligheder. Ved at finde flere varierende tilesets, ville vi have kunne producere banerne med mere variation, hvor vi i vores nuværende design har genbrugt de samme tiles. Problematikken i dette har været at finde tilesets der har de samme egenskaber som dem vi har brugt. Hvis f.eks. dimensionerne havde været anderledes, ville det have svært at implementere disse.

## Funktioner ved genstande

Vores spils formål er at indsamle 3 genstande, herunder kort, mad og en nøgle. Disse genstande gør det muligt for spilleren at avancere til næste bane. I en fremtidig version, ville det at opsamle genstandene, kunne have en vis funktion. F.eks. ligesom opsamlingen af kortet giver spilleren et mini-map, så vil maden evt. kunne give spilleren et hastighedsboost, eller måske udødelighed i en begrænset periode. Ved at indføre flere forskellige typer genstande, som hver især har deres egen funktion, vil man som spiller have flere muligheder for at gennemføre spillet. Nogle yderligere tilføjede genstande kunne måske også kunne nedkæmpe fjenden, eller gøre spilleren i stand til at skubbe orkerne væk, inden de kommer så tæt på at spilleren dør.

## Anvendt Programmering

### Gemmefunktion

I sin nuværende form, kan spillet ikke gemmes. Hvis spilleren dør, skal spilleren starte helt forfra. I vores inspiration fra gamle Game Boy spil og lignende, er dette en ofte gennemgående tendens, og i praksis virker dette også for vores spil, da vi kun har 3 baner. Hvis spillet udvides med flere baner, kunne man overveje at indføre muligheden for at gemme spillet.

### Sværhedsgrad

Vores fjender har kun én sværhedsgrad, som ikke tilgodeser hele vores målgruppe. Særligt det yngre publikum kan have svært ved at klare spillet i dets nuværende sværhedsgrad. En udvikling af vores spil kunne være at kunne vælge sværhedsgrad, hvor f.eks. fjendens hastighed sættes ned, eller deres respons på spilleren ændres så afstanden til spilleren skal være nærmere, før fjenden reagerer på spilleren. Andre muligheder kunne være, ved lettere sværhedsgrad, at give spilleren flere liv således at man ikke dør i spillet ved første berøring.

### Referencer

- Gamefabrique.com. (n.d.). fra <https://gamefabrique.com/games/pokemon-gold-version/>
- Jest array. (2019). Github.com. <https://github.com/jestarray/gate/tree/yt>
- Jest array. (2019, March 2.). *Phaser 3 Game Tutorial 9 Tiled* [Video]. YouTube.  
[https://www.youtube.com/watch?v=2\\_x1dOvgF1E](https://www.youtube.com/watch?v=2_x1dOvgF1E)
- Palef, T. (2021) *Make 2D Games in JAVASCRIPT with PHASER*.
- (n.d.). Phaser 3 API Documentation. <https://newdocs.phaser.io/docs/3.60.0>
- Westover, S. (2023, February 17). *How to Create a Phaser MMORPG – Part 3*. GameDev Academy.  
<https://gamedevacademy.org/how-to-create-a-phaser-3-mmorpg-part-3/>

### Lyde:

- Pixabay. (n.d.). *Desert Monolith*. (n.d.). Pixabay.com. <https://pixabay.com/sound-effects/search/desert/>
- SARDIN, J. (n.d.). *Fast steps on concrete (Free Sound Effect)*. BigSoundBank.  
<https://bigsoundbank.com/sound-1319-fast-steps-on-concrete.html>
- SARDIN, J. (n.d.). *Forest 4 (Free Sound Effect)*. BigSoundBank. <https://bigsoundbank.com/sound-2749-forest-4.html>
- SARDIN, J. (n.d.). *Running in the tall grass (Free Sound Effect)*. BigSoundBank.  
<https://bigsoundbank.com/sound-1843-running-in-the-tall-grass.html>

## Anvendt Programming

- UNIVERSFIELD. (n.d.). *Male Scream in Fear*. Pixabay.com. [https://pixabay.com/sound-effects/search/death/?manual\\_search=1&order=None](https://pixabay.com/sound-effects/search/death/?manual_search=1&order=None)

### PNG filer:

- mohsen. (n.d.). *3d House Key Golden Cartoon Vector Hd Images*. Pngtree.com.  
[https://pngtree.com/freepng/3d-house-key-golden-cartoon-vector\\_6960475.html?sol=downref&id=bef](https://pngtree.com/freepng/3d-house-key-golden-cartoon-vector_6960475.html?sol=downref&id=bef)
- Saradha. (n.d.). <https://www.Cleanpng.Com/png-sandwich-png-clipart-vector-picture-13158/>.  
Cleanpng.com. <https://www.cleanpng.com/png-sandwich-png-clipart-vector-picture-13158/>
- Tarubhabhi. (n.d.). <https://www.Cleanpng.Com/png-world-map-treasure-map-clip-art-route-map-400200/download-png.Html>. Cleanpng.com. <https://www.cleanpng.com/png-world-map-treasure-map-clip-art-route-map-400200/download-png.html>