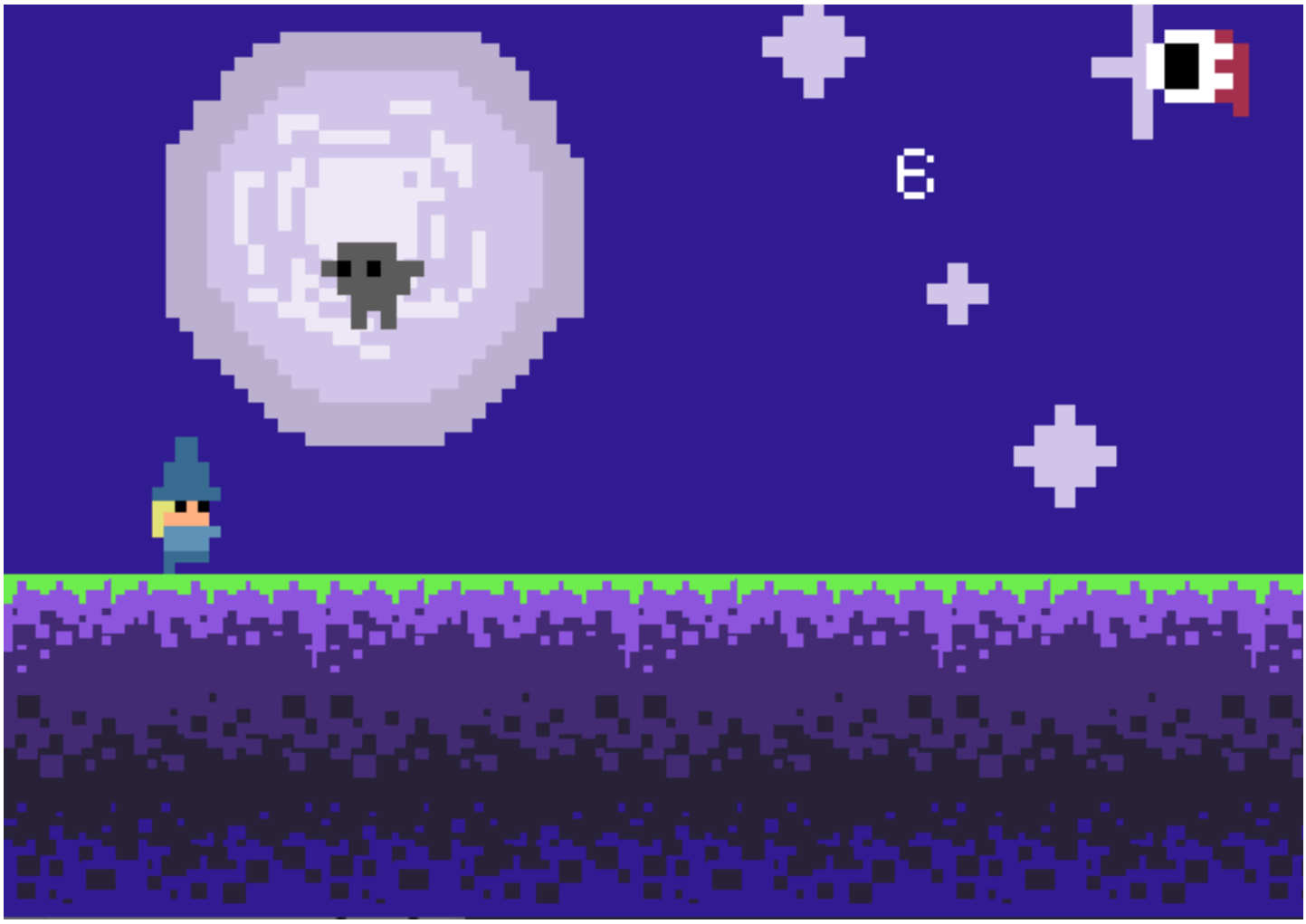
Computerspil med A.I.

19/02/2023



Screenshot af det færdiggjorte produkt (Fig.1)

Resume

Opgaven var at skabe et simpelt spil med en simpel A.I. I starten af projektet var planen at arbejde med javascript såvel som libGDX-biblioteket, men den plan blev ændret halvvejs under udviklingen på grund af teamwork-problemer. Det færdige produkt blev skrevet i python og pygame. Spillet er et "flappy bird" type spil, hvor du skulle undgå fjender. Under udviklingen løb fjendens A.I ind i en fejl, hvor den blev ude af stand til at operere med brugen af vektorfunktionerne, der fulgte med pygame-pakken, som desværre ikke blev patched til sidst.

Problemformulering:

Problemet, der skulle løses i denne opgave, var at skabe et simpelt spil med en simpel AI.

Program beskrivelse:

Oprindeligt var planen at lave spillet i javascript ved hjælp af libGDX-biblioteket, da begge medlemmer af gruppen havde erfaring med javascript.
Det viste sig dog senere, at et af gruppemedlemmerne ikke dukkede op til det meste af programmeringsprocessen, så sproget blev ændret til python for at lette arbejdet, da der var problemer med libGDX-biblioteket.
Ideen med programmet var at lave et spil, hvor man skulle skyde fjender, men den idé ændrede sig, da programmeringssproget blev ændret på grund af tidsbegrænsninger.

Det færdige produkt blev skrevet i programmeringssproget Python og biblioteket Pygame. Spillet handler om at undgå fjender og forsøge at overleve så længe som muligt, hvor slutresultatet vises i spillet over skærmen.

Funktionalitet:

I starten af opgaven fik vi til opgave at lave en liste over ønskede systemer, samt en tidsplan for, hvordan vi planlagde at dele arbejdet op. Her er den originale liste over de ønskede funktioner i programmet:

Need to have:	**Nice to have:**

Player movement	3 eller flere forskellige typer fjender
2 forskellige typer af fjender	Bosskamp, med flere specielle mønstre
1 skydevåben til spilleren	Tilfældigt genereret bane
Liv baseret system	Consumables
----------	Pæn pixel-art
----------	Meta progression
----------	En “shop” med opgraderinger til spilleren
----------	God UI

De færdige funktioner i det færdige program var:

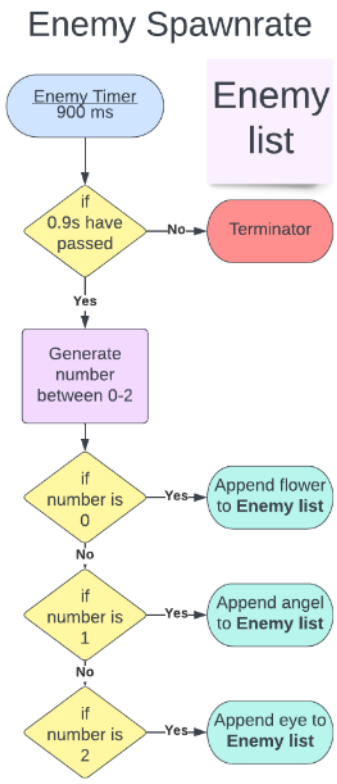
Completed features
Player movement
2 forskellige typer af fjender + 3 eller flere forskellige typer fjender
Tilfældigt genereret bane (tilfældigt genereret enemy spawnrate)
Pæn pixel-art (kan diskuteres)
God UI

Rutediagrammer

Flowcharts bruges generelt til at beskrive, hvordan tingene fungerer sammen, enten i kode eller andre arbejdsmiljøer. I dette tilfælde mener jeg, at det ville være en god idé at lave en simpel visuel fremstilling af, hvordan fjendens tilfældige spawn fungerer.

Flowchart for snippets of the code (Fig.2)

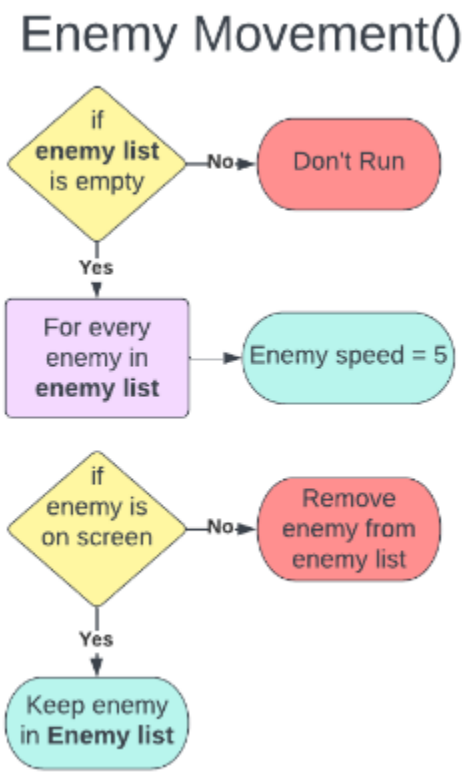
Lad os tage et kig på hver af dem separat:



Enemy Spawnrate flowchart (Fig.3)

Måden fjendens spawn-system fungerer på i spillet er ved hjælp af timerfunktionen, som er en del af Pygame-biblioteket. Timerfunktionen lader os vælge en bestemt mængde tid, der skal gå for at udløse en hændelse.
I dette tilfælde er den nødvendige tid til udløseren 900ms. Hver 900 ms vil programmet generere et tal mellem 0-2 og tilføje en specifik fjende til fjendelisten baseret på resultatet. Dette er en lidt enklere måde at beskrive det på. I virkeligheden tilføjer programmet kun beskrivelserne af fjenden til fjendelisten. Det betyder, at selve tegningen ikke gemmes og skal tilføjes i bevægelsesfunktionen senere.

Lad os tage et kig på Enemy Movement flowchartet som det næste:



Enemy Movement flowchart (Fig.4)

Fjendens bevægelsesfunktion er ret enkel. Først tjekker programmet, om fjendelisten er tom eller ej. Hvis listen er tom, vil programmet ikke køre, og hvis det er, vil det give hver fjende 5 hastigheder. I virkeligheden er hastigheden sat til -5 for at flytte fjenden i den rigtige retning, men den blev ændret til et positivt tal i flowchartet for lettere at forstå.

Programmet tjekker derefter for fjendens position, indtil dets x-position er væk fra skærmen. Når den er på skærmen, slettes fjenden fra listen for at optimere programmet. Mere specifikt opdateres fjendelistevariablen til at kopiere sig selv, undtagen de dele, der er uden for skærmen.

Hvad der ikke er vist i rutediagrammet, er hvordan funktionen bestemmer hvilken fjende der skal tegnes, hvilket faktisk er ret simpelt. Funktionen kontrollerer blot fjendens y-position og baserer tegningen på det.

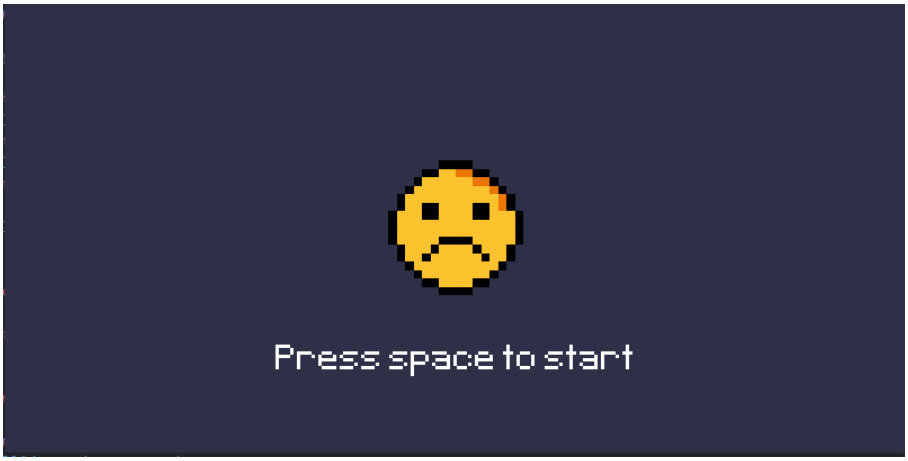
Brugergrænseflader

Brugergrænsefladen i spillet er ret enkel, men der er UI. Lad os tage et kig på et par skærmbilleder fra spillet:

Start UI (Fig.5)

Når spillet først åbnes, vil spilleren blive introduceret til introduktionsskærmen, som siger "Tryk på mellemrumstasten for at starte", hvilket giver brugeren en forståelse af, hvordan spillet fungerer, da det er det eneste rigtige input, som spillet faktisk har brug for.

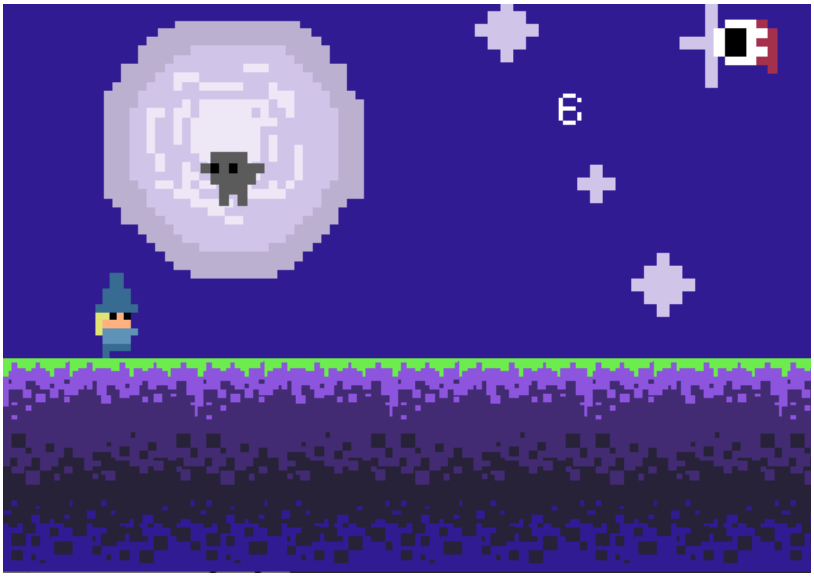
Når først spilleren dør, bliver introduktionsskærmen dog opdateret:



Game Over UI (Fig.6)

Teksten ændres til at sige "Game Over" såvel som at vise den score, som spilleren har opnået. Den måde, spillet tæller score på, er ret enkel. Den bruger simpelthen timerfunktionen fra Pygame-biblioteket og trækker derefter 1000 fra for i stedet at få sekunder (viser kun hele tal).

Og sidst men ikke mindst ingame UI:



(Fig.1)

I spillet kan et par ting ses. Hovedsageligt spilleren, fjenderne og den aktuelt opnåede score, som konstant opdateres. Brugergrænsefladen er ganske enkelt, men nok til, at brugeren forstår, hvad der foregår.

Pseudokode

Pseaudocode er en god måde at få noget grundlag på, før du begynder at kode eller hjælpe en med at forstå præcis, hvad koden gør. Selvom det er meget nyttigt, er det også meget enkelt, så vi vil kun gennemgå et kort eksempel:

```
if player is not touching ground
    player_sprite1 = player_jump_sprite
else:
    every 10 frames(player_sprite1 = player_sprite2)
```

Her har vi nogle meget enkle pseudokode, der beskriver, hvordan afspilleranimationen skal fungere. Først tjekker programmet, om spilleren hopper eller ej. Det gør den ved at kontrollere, om spilleren rører y-positionen, hvor jorden er placeret. Hvis spillerens y-værdi er lavere end 250, vil springspriten blive kaldt. Ellers ønsker vi, at programmet skifter mellem sprite1 og sprite2 hver 10. frames for at skabe en kørende animation.

Kode

Lad os tage et kig på, hvordan animationen faktisk fungerer:

```
def player_ani():
    global player_surface, player_index

    if player_rect.bottom < 250:
        player_surface = player_jump
    else:
        #10 frames to walk 2
        player_index += 0.1
        if player_index >= len(player_walk):
            player_index = 0
        player_surface = player_walk[int(player_index)]
```

Her har vi animationsfunktionen. Som nævnt tidligere kontrollerer programmet for spillerens y-position, og hvis den er mindre end 250 ændres spriten.
Den måde, gå-animationen udføres på, er ved at skabe en indeksværdi, som stiger med 0,1 for hvert billede.

```
player_walk1 = pygame.image.load('Art/Player/Player2.png').convert_alpha()
player_walk1 = pygame.transform.scale(player_walk1,(30,60))
player_walk2 = pygame.image.load('Art/Player/Player3.png').convert_alpha()
player_walk2 = pygame.transform.scale(player_walk2,(30,60))
player_walk = [player_walk1,player_walk2]
```

Længere nede har vi en liste over de to sprites, hvilket betyder, at indekset starter ved 0, hvilket udløser player_walk1-spriten i arrayet, og når player_index-variablen rammer 1, skifter den til spiller 2.

Dette er koden for at få spillet til at skifte spilstadier:

```
game_active = collision(player_rect,enemy_rect_list)
    else:
        screen.fill((45,48,71))
        screen.blit(sad_smiley,sad_smiley_rect)
        enemy_rect_list.clear()
        player_rect.midbottom = (80,300)
        player_gravity = 0
```

```
        score_message = test_font.render(f"Score: {score}",False,"White")
        score_message_rect = score_message.get_rect(center = (400,320))

        if score == 0:

            screen.blit(game_start,game_start_rect)
        else:
            screen.blit(score_message,score_message_rect)
            screen.blit(game_over,game_over_rect)
```

Den måde spillet skifter til spillet over skærmen på er ved at tjekke for kollisioner mellem spilleren og fjenden. Det gør den ved at kalde kollisionsfunktionen:

```
def collision(player,enemies):
    if enemies:
        for enemy_rect in enemies:
            if player.colliderect(enemy_rect):
                return False
    return True
```

Kollisionsfunktionen er som standard indstillet til at returnere True, og kun når spilleren kolliderer returnerer den en falsk. Hvis game_active-variablen er lig med false, tegnes den på skærmen, der overlejrer spillet. Også de fleste af spilfunktionerne slås fra, mens game_active er falsk. Kodestykket kan findes i kodefilen.

Test af programmet

Under udviklingen af programmet dukkede der en fejl op, som jeg desværre ikke var i stand til at rette.
Fejlen kom fra funktionen move_towards_player, som skabte en retningsvektor mellem spilleren og fjendens rekt.

```
def move_towards_player(enemy_rect, player_rect):
    dirvect = pygame.math.Vector2(0, enemy_rect.y - player_rect.y)
    return dirvect
```

Men når retningsvektoren blev tilføjet til y-værdien af fjendens hastighed, ville fjenderne forsvinde i de fleste billeder, hvilket gjorde, at spilleren ikke kunne se dem:

```
enemy_rect.y -= move_towards_player(player_rect, enemy_rect)[1]
```

Udover det stødte programmet ikke på mange problemer under udvikling, bortset fra den del, at et af gruppemedlemmerne havde opgivet projektet.

Konklusion

Afslutningsvis var det en fejl at dykke ned i libGDX, da det var et så ukendt bibliotek, og skiftet fra java til python burde have været foretaget tidligere for ikke at spilde nogen tid på projektet.