Prosjektoppgave i Ingeniørfaglig innføringsemne

Prosjektoppgave fagnært prosjekt.

Milepæl 0, 1 og 2
29.11.2021

FORFATTERE
Student 1
Student 2
Student 3
Student 4
Student 5
Student 6

DATO
29.11.2021

ANTALL SIDER: **21**
ANTALL VEDLEGG: **3**
Vedlegg: Refleksjonsnotater.pdf, Milepæl 1 sortert kode, piProject-main.zip

Sammendrag

Den tekniske rapporten har bestått av tre ulike milepæler med ulikt fokus. Milepæl 0 har hatt som fokus å lære oss bruken av flere elementer, der vi tar i bruk sensorverdier og viser frem et dynamisk design på LED-matrisen. Resultatet av dette ga oss 6 forskjellige program som tok i bruk sensorverdiene og fylte LED-matrisen med informasjon over alt fra et fungerende vater (ved innhenting av gyroverdier) til «Game of Life» (ved innhenting av fuktighet). Det vi erfarte i denne milepælen var hvordan vi henter IMU – verdier fra Sense – Hat og bruker de for å styre et program. Deretter flettet vi alle programmene våre sammen til et program, ved å definere hvert av programmene våre som funksjoner, og kalle på disse i en main() loop og samtidig lagre retur verdiene til funksjonene i en CSV – fil.

I milepæl 1 var oppgaven å hente forskjellige data fra sensorene på Sense Hat og sortere dem. Deretter bruke denne dataen til å måle variasjoner i høyden og loggføre dette. Hensikten med denne milepælen var å kunne bruke sensor data fra Sense HAT i et fungerende program. For å unngå unøyaktige målinger kalibrerte vi sensorene for å få de mest nøyaktige målingene. Etter kalibrering ble Raspberry PIen tatt med på gåtur fra Gløshaugen til lenger ned i terrenget for å loggførte høydeforskjell. Data fra dette skrevet til en CSV-fil, der man fikk full log over PI'ens høyde gjennom perioden det ble målt. Gjennom denne milepælen fikk vi god innføring i hvordan datalogging kan settes opp i et program. Vi lærte også hvor viktig det var å ha kontroll på filsystemet til Pi'en.

I milepæl 2 har vi laget et fullt fungerende spill som benytter flere forskjellige teknologier for å få et spill som fungerer slik vi ønsket. Det er i hovedsak brukt Python i koden for spill – logikk og grafikk, samtidig som vi har brukt HTML, CSS og JavaScript for å danne et grensesnitt for flerspiller. Pythonmodulen Flask er brukt for API og webserver. Ved bruk av Sense Hat har vi mulighet til å innhente sensorverdier og dermed bruke dette for å kontrollere spillet vårt. Denne kommunikasjonen til sensorer, som vi sitter igjen med fra de tidligere milepælene, danner grunnlaget for styringen til spillet.

Programmet vårt baserer seg rundt en hovedløkke som bruker funksjoner til å utføre bestemte handlinger. Resultatet av dette er et spill med sensorbasert styring, et vidt spekter med grafikk (som kjører basert på forskjellige kondisjoner), funksjoner som endrer spillgrafikken i sanntid (blant annet kalkulerer bilposisjon og tegner drivstoff), samt egenprodusert musikk og lydeffekter som gir en mer dynamisk spillopplevelse.

Innholdsfortegnelse

Teori **4**

Milepæl 0 **5**

Milepæl 1 **6**

Milepæl 2 **8**
Midjo Grand Prix 1970 *8*
Mål med spillet *8*
Utvikling *9*
Grafikk *9*
Spillegrafikk 9
Visuell-grafikk 10
Programstruktur *12*
Bibliotekimporteringer 12
Konstanter 12
Klassedefinisjoner 12
Funksjonsdefinisjoner 12
Eventloop 13
Inngangspunkt 14
Implementering *14*
Multiplayer *15*
Lyddesign *17*

Diskusjon **18**

Konklusjon **21**

Referanser **21**

Figurliste

Figur a - Henting av sensordata fra Sense HAT 4
Figur b - Visualisering av buffer 9
Figur c - Bruk av buffer i koden 10
Figur d - Spillets layout 10
Figur e - LvL-grafikk (speedometer) 10
Figur f - Vinnergrafikk (pokal) 10
Figur g - versikt over en av flere grafikk-koder 11
Figur h - Noen av konstantene i programmet 12
Figur i - Eksempel på bruk av iteratoren for flytkontroll 13
Figur j - Inngangspunktet til programmet 14
Figur k - Prosjektstruktur 15
Figur l - Aktivering av mobilenhetens gyro 15
Figur m - Bilens posisjon med input fra nettleser 15
Figur n - Oppretting av ssl sertifikat 15
Figur o - Styring av spill i java og oppsett av socket 15
Figur p - SocetIO event listener i Python 16
Figur q - Utregning av posisjon 16
Figur r - Nåværende struktur 19
Figur s - Sekvensiell timed loops 20
Figur t - Multithreaded event loops 20

Teori

Sense-Hat er et tilleggsbrett for Raspberry Pi, med en rekke sensorer, en joystick og en 8x8 LED-matrise. Sense-Hatten kan programmeres enkelt ved hjelp av et eget pythonbibliotek. Med dette biblioteket kan en hente ut sensorverdier og bruke de i programmet sitt, i Midjo Grand Prix 1970 bruker vi en IMU for å hente ut informasjon om Sense-Hattens retningsendringer. (Raspberry Pi, 2021)

IMU er en forkortelse for Inertial Measurement Unit, og fungerer som et gyroskop, akselerometer og magnetometer i én sensor. Et gyroskop måler retninger og momentet til SenseHaten og gir ut verdier i 3 akser (x, y, z), Akselerometeret måler kreftene som virker i de tre aksene, og magnetometeret måler magnetfeltet, og fungerer litt som et kompass. Disse tre sensorene gir sammen ut verdier i 3 akser hver, og vi kaller det ofte for «Nine degrees of freedom». Disse sensorene kan brukes isolert alene, eller kombineres. (Raspberry Pi, 2021)

SenseHaten har også en fuktighetssensor som går under kategorien «Enviromental sensors». Denne henter ut verdien av relativ fuktighet i luften (Relative Humidity), som igjen kan brukes til å si noe om temperaturen i rommet. (Raspberry Pi, 2021)

Den siste sensoren er en trykksensor, som måler lufttrykket i rommet. Den gir ut verdier i bar, og kan også brukes til å si noe om temperaturen i rommet. Med to sensorer som kan si noe om temperaturen, kan man lettere gi en mer nøyaktig temperaturmåling. (Raspberry Pi, 2021)

For å hente ut sensorverdiene fra Sense-Haten, bruker vi pythonbiblioteket fra Raspberry Pi. Nedenfor vises hvordan pythonkode brukes til å hente ut verdiene fra fukt, trykk og IMU-sensorene. Nettsiden «https://pythonhosted.org/sense-hat/api» har en god oversikt over metodene for å hente ut informasjon fra Sense-Haten.

```python
# Importerer sensehat bibliotek
from sense_hat import SenseHat
sense = SenseHat()

# Henter relativ fuktighet fra fuktighetssensor
humidity = sense.get_humidity()

# Henter trykk fra trykksensor
pressure = sense.get_pressure()

#Henter IMU verdier
orientation = sense.get_orientation()
accel_only = sense.get_accelerometer()
north = sense.get_compass()
```

Figur a - Henting av sensordata fra Sense HAT

Milepæl 0

Hensikten med Milepæl 0 var at hvert gruppemedlem skulle lage et program der de tok i bruk av Sense-Hatten sine forskjellige sensorer. Når alle medlemmene på gruppen var ferdige med programmet sitt og hadde fått det til å fungere, så skulle alle programmene sy sammen i et felles hovedprogram. Kravene i denne milepælen var at hvert program skulle gjøre minst én sensormåling, og vise fram et dynamisk design på LED-matrisen. I det ferdige programmet er det en hovedfunksjon som skulle kjøre hvert av gruppemedlemmenes funksjon. Deretter kunne vi selv bestemme om hovedfunksjonen skal kalle på én og én funksjon etter hverandre, eller gjøre noe mer kreativt. Resultatet fra hvert av funksjonskallene samt en tidskode ble lagret i en CSV-fil, der hver linje representerer hver runde main-funksjonen kjøres. Main ble bare kalt dersom det faktisk er denne filen som blir kjørt.

Utbyttet vi fikk ut av denne milepælen var at vi lærte å bruke LED-matrisen på ulike kreative måter ved hjelp av alle de forskjellige sensorene, og det var bare fantasien som satte en stopper for hva man kunne lage. Det eneste som kan stoppe deg er at led-matrisen er 8x8, som gjør at noen design er vanskelige å vise på display. Flere erfaringer vi fikk var at vi lærte å hente data fra sense-hatten, ved å skrive et program som leser av de sensorverdiene vi ville ha. Et eksempel på dette er et program som leser av lufttrykket i rommet gjennom trykksensoren til sense-hatten.

I milepæl 0 var det fokus på det å skrive funksjoner på en best mulig måte, sånn at programmet ble oversiktlig og lett å forstå. For å oppnå dette er det viktig å ha en struktur innen programmering og dele programmet opp i funksjoner. Hvis man slenger alt under et program vil det sees som et totalt rot. I dataprogrammering er en funksjon en navngitt del av en kode som utfører en spesifikk oppgave. Dette innebærer som regel å ta noen inputs, manipulere inputs og returnere en output. Måten vi brukte funksjoner i milepæl 0, er for eksempel å hente en sensorverdi i en funksjon og kode hva vi skal med den. Når vi har fått funksjonen til å gjøre det vi vil, returnerer vi en verdi og tar den med videre i programmet. Deretter kan vi da bruke denne verdien i en annen funksjon eller skrive en helt ny funksjon med en ny returverdi. Til slutt bruker vi disse verdiene til å utføre programmet vårt.

Funksjonene som gikk i loop i løpet av programmet er:

- Animert termometer
- Liggende vater som bruker "pitch"
- Stående vater som bruker "yaw"
- Conway's game of life, med startverdier fra fuktighetsmåler

- En funksjon som printer tilfeldige verdier til skjermen og returnerer kompassverdier
- Trykkmåler med en horisontal bar
- Videoavspilling av "Never Gonna Give You Up"

For hver gang denne loopen kjører, skrives de dataene som returneres til en .CSV-fil, som logger resultatene. Dette skjer om og om igjen til loopen avbrytes.

Milepæl 1

Milepæl 1 har hatt hovedfokus rundt henting av data fra sensorer på Sense HAT. Disse dataene skulle så sorteres og brukes for å måle variasjoner i høyde. Målet etter dette var å kunne loggføre variasjoner i høyde. Som tilleggsoppgave ble det også gitt en utfordring om å kunne bruke disse dataene sammen med den innebygde LED-matrisen på Sense HAT for å kunne visualisere hvilken etasje man var i når man tok heisen opp en bygning.

Hensikten med milepælen var å kunne bruke sensordata fra Sense HAT i et fungerende program. Vi måtte som første del av milepælen lære oss å kalibrere ulike sensorer, slik at Raspberry PI'en kunne gi korrekte data til senere bruk i programmet. Denne kalibreringen var nødvendig slik at verdiene gitt til PI'en i det hele tatt kunne brukes. Vi så før og etter denne kalibreringen, viktigheten av kalibrerte sensorer for at programmet vi kjørte skulle ha ønsket utfall. Videre kom viktigheten med å kunne lese å forstå kode frem. Uten den allerede kjente kunnskapen gjennom innføringen i Python, ville vi ikke kunnet forstå hva programmet gjorde. Ved å da kunne lese og forstå koden, kunne vi starte å stokke om på de gitte kodesnuttene, slik at vi hadde et program som fungerte.

Milepælen var lagt opp i ulike stadier, der ulike deler av programmet skulle leses og forstås. Vi startet først med å skaffe en oversikt over de ulike elementene, før vi videre begynte å sette sammen kodesnutter for å få et program som fungerte mer og mer. Hensikten med dette var nok å kunne kjøre deler av koden for å verifisere at hver enkelt funksjon fungerte for seg selv, og sammen med allerede eksisterende kode. Med dette kan man verifisere at man hadde et fungerende produkt uten at vi trengte å ha et fullt fungerende program. Vi fokuserte også på å kommentere hvordan de ulike delene av programmet fungerte, slik at alle fikk en forståelse for oppbygging av programmet og programmets virkemåte.

Videre i milepælen ble det implementert data-logging fra PI'en. Sense Haten ble her kalibrert til høyden oppe på Gløshaugen, for deretter å gradvis synke i høyde over havet etter hvert som vi

bevegde oss på en gåtur. Dette skulle resultere i en CSV-fil med full logg over PI'ens høyde gjennom perioden det ble målt. Gjennom denne deloppgaven fikk vi en god innføring i hvordan datalogging kan settes opp i et program. I tillegg lærte vi viktigheten med å ha kontroll på filsystemet til PI'en slik at de filer vi ønsker skrevet gjennom programmet vårt havner der vi ønsker. For de som ikke var vant med Linux og kommando-vinduet, ble dette en pangstart i å finne plassering, åpne og bruke filer på et system der man ikke bare kan klikke på ønsket fil. Heldigvis finnes det programmer som gjør jobben enklere, men ikke alle funksjoner vi ønsker er del av programmer slik som winSCP, som er til for å gjøre filhåndtering mellom Windows- og Linux-maskiner enklere.

Når vi da hadde sortert all kode og funnet ut av ønskede funksjoner, resulterte dette i et fungerende program som kunne lese høyden over havet i sanntid eller høyden over et gitt nullpunkt samtidig som programmet kunne føre logging av Raspberry Piens høyde til enhver tid. Vi så dermed fort viktigheten i å jobbe strukturert med koden vi senere skulle utvikle. Samtidig så vi i praksis hvor godt et program kan kjøre, selv om det ikke er «ferdig» utviklet. Vi lærte også viktigheten med kontinuerlig testing av den koden vi skulle utvikle, for å kunne utelukke eventuell fremtidige bugs tidlig, slik at dette ikke skulle skape følgefeil. Milepælen var også til god hjelp til de deler av gruppen som aldri hadde vært borti Raspberry PI, da de fikk «hands-on» erfaring på mikrokontrolleren med kode som i teorien skulle funke med mindre modifikasjoner. Resultatet av lesing, koding, tyding og testing var en gjeng som følte seg mer klar for den fremtidige kode-oppgaven og kjente til hvordan Sense HAT ville fungere sammen med PI, og hvordan vi kunne unngå mulige fremtidige feilkilder og feillesing av data.

Milepæl 2

Midjo Grand Prix 1970

Midjo Grand Prix 1970 er et racing-spill utviklet av gruppe 19, for Raspberry Pi SenseHat. I Midjo GP 1970 spiller man som karakteren Arne Midjo med den fantastiske bilen «Hype one». Spillet går ut på å manøvrere bilen gjennom de ulike portene som dukker opp på racingbanen, samtidig som man må ha kontroll på drivstoffet. «Hype One» er en veldig rask bil, og brenner godt med drivstoff. Derfor er en av utfordringene ved spillet å opprettholde drivstoffnivået. Ved å fange drivstoffkanner som dukker opp på racerbanen kan man holde bilen i gang i evigheter!
Midjo Grand Prix 1970 egner seg for hele familien, og har en multiplayerfunksjon, der du kan bruke mobilen, nettbrettet eller PC-en som kontroller. Konkurrer om å få den beste highscoren!

Mål med spillet

Målet vårt med Midjo GP var å lage et racingspill som var utfordrende, samtidig som det var intuitive løsninger på kontroll, grafikk og gir spilleren en god forståelse av hva som skjer mens du spiller. Dette var løst ved å bruke «pitch» verdien som vi henter fra gyro – sensor, dette gjør at vinkling på Pi mot høyre, får bilen til å svinge til høyre, og motsatt. En fin intrografikk som blant annet starter musikken til spillet, og hvilket nivå du begynner på (starter på level 1). Når selve spillet begynner, har vi en drivstoffmåler på høyre side, som oppdaterer seg over tid (du bruker drivstoff hele tiden mens du kjører), og når du treffer drivstoff (gule piksler). Ved treff på drivstoff kommer en lyd for å indikere at du faktisk traff drivstoff, og drivstoffmåler øker. Vi har et poengsystem, så hver gang du kjører gjennom en port, får du poeng, dette indikeres både med lyd og en binærteller som er øverst på skjermen. Hvis du får en hvis poengsum, kommer du til neste nivå, hvor vanskelighetsgraden, og poengkrav blir høyere. Dette indikeres med tekst, og et speedometer som øker. Går du tom for drivstoff er det «Game Over», og en «game_over_graphic» funksjon kjører, da får du mulighet til å prøve på nytt ved å trykke på joystick (dette er også mulig hvis du fullfører spillet). Når du får 32 poeng på siste nivå (level 3), vinner du spillet, og en funksjon «winner_graphic» spiller.

Utvikling

For å utvikle et spill i en gruppe, har vi tatt i bruk GitHub, ettersom flere i gruppen har eksisterende kunnskap i GitHub, og lærte resten av gruppen hvordan dette fungerte. GitHub gir oss god oversikt over endringer vi har gjort, samt en trygghet ved at vi alltid kan gå tilbake hvis koden ikke fungerer lenger.

Vi har løst arbeidsfordeling ved å utdele funksjoner, da i hovedsak lage hele funksjoner selv. Eksempel på dette kan være at en person får i oppgave å lage intro grafikk, da lages en funksjon «intro_graphic», og denne legges inn i koden, og kan enkelt brukes i «main» ved å gjøre et funksjonskall.

Grafikk

Grafikken i spillet baserer seg på to ulike prinsipper, der selve spillet defineres som spillegrafikk og alle animasjoner defineres som visuell grafikk. Disse kjører uavhengig av hverandre og bruker ulike metoder for å oppnå det resultatet man ønsker.

Spillegrafikk

En av metodene vi bruker for å tegne grafikk til Raspberry Pi, er ved hjelp av noe vi kaller «buffer». Buffer er en liste, som i seg inneholder 8 lister, hver av disse listene, inneholder 8 elementer.

Hver liste fungerer som en y-verdi, og hvert element i disse listene, fungerer som en x-verdi. Visualisert vil det se litt ut som i figur b.

```
y_0[x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7]
y_1[x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7]
y_2[x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7]
```

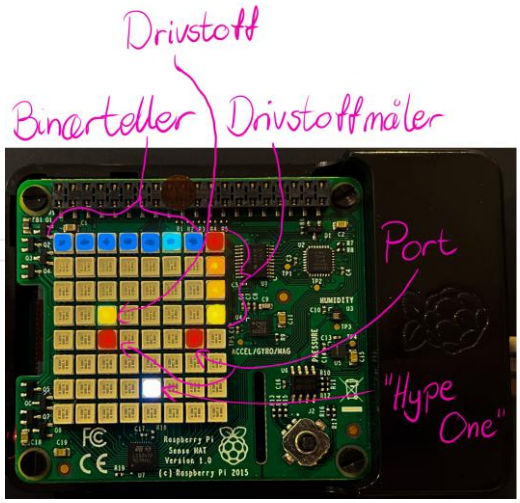
Figur b - Visualisering av buffer

Som et eksempel på dette, kan vi vise måten vi tegner drivstoff-nivået på. Når vi skal tegne drivstoff på høyre side av «skjermen» ønsker vi bare å bruke x_7 piksler, som er til høyre i figur b. Tegning av drivstoff-nivå gjøres da ved hjelp av en while-løkke, hvor y varierer, mens x er konstant. For å få fargene til å variere fra rødt til grønn, kan vi addere og subtrahere med 36, som er 255 bit delt på 7. På figur c ser vi hvordan dette fungerer i praksis. Ved bruk av både buffer og while-løkke, får vi resultatet vi kan se i figur d (x i dette tilfellet er 4).

```python
def draw_fuel(mod_buffer, x):
    """Mellom 0 og 8, 0 er null fuel, 8 er max fuel
    funksjonen tar inn buffer-variabel og x (fuel)
    returnerer en modifisert buffer"""
    x_pos_fuelGauge_lokal = 7 #sier bare hvilken kolonne du ønsker
    i = 0
    u = 0
    # En spørring for å sikre at x ikke er større enn 8, eller mindre enn 0
    if x < 0:
        return mod_buffer
    if x > 8:
        x = 8

    #Dette skriver om på buffer, og lager en kollonne med fuel
    while i < x:
        mod_buffer[i][x_pos_fuelGauge_lokal] = (255 - u, u, 0)
        i += 1
        u += 36
    """Bruker u som nullverdi for neste whileløkke
        og beholder i, ettersom i er y verdi for sorte pixler
        Sikrer at du tegner over eventuelle drivstoffpixler
        fra siste gang du tegnet"""
    u = 0
    resterende_pixler = 8 - x
    while u < resterende_pixler:
        mod_buffer[i][x_pos_fuelGauge_lokal] = NOCOLOR
        u += 1
        i += 1
    return mod_buffer #returnerer en modifisert buffer
```

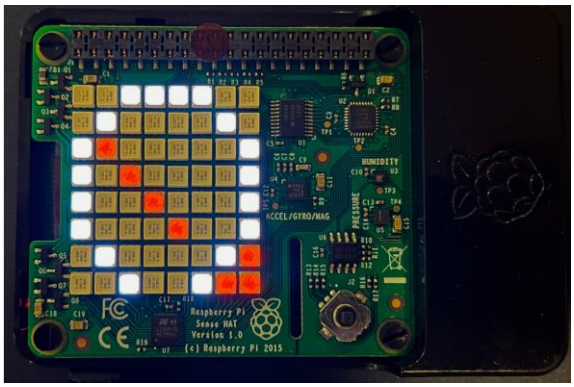
Figur c - Bruk av buffer i koden



Figur d - Spillets layout

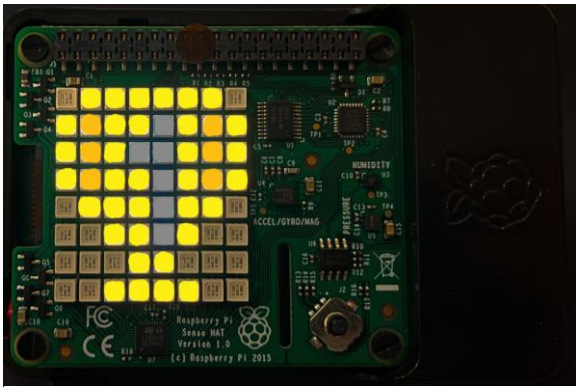
Visuell-grafikk

I Midjo GP har vi ikke bare fokusert på spillgrafikk, men vi har også lagt til visuell grafikk. Målet med den visuell grafikken er å gi mer kjøtt på beinet, slik at spillet blir mer interessant og at man blir oppslukt i det. Steder vi har implementert visuell grafikk er i starten av spillet, mellom de forskjellige nivåene, når du går tom for drivstoff, samt når du klarer alle nivåene og vinner.



Figur e - LvL-grafikk (speedometer)

I starten av spillet har vi grafikk som displayer navnet på spillet. Etterfulgt av en melding som forteller hvilket level (nivå) man befinner seg i, og et display som viser et speedometer. Denne meldingen og speedometeret, vil komme hver gang man når et nytt nivå, og da vil man se at speedometeret vil øke for hvert nivå du når.



Figur f - Vinnergrafikk (pokal)

Speedometeret i nivå 1 vises i figur e. Når du går tom for drivstoff vil du bli møtt av en blinkende hodeskalle, etterfulgt av Arne Midjo som gråter over at du ikke klarte å kjøre «Hype One» (bilen) til seier. Hvis du er en ekte «hype» sjåfør og klarer å kjøre bilen til en seier, vil du bli belønnet av et blinkende trofé samt heder og ære for den utrolige oppnåelsen. Denne seiersgrafikken kan vi se i figur f, en skikkelig belønning for spilleren.

Når vi skulle vise den visuelle grafikken på displayet, benyttet vi oss av en metode der vi brukte en matrise til å «tegne» den visuelle grafikken. En matrise består av elementer i rader og kolonner. Før vi begynner å designe, setter vi variabler til forskjellige farge koder (RGB). Dette gjør vi, fordi det er mye lettere og mer oversiktlig å ha variabler enn RGB parenteser av tall i matrisen. Når vi skal få den visuelle grafikken til å bevege seg som i speedometeret, må vi endre matrisen. Dette har vi løst litt forskjellig siden flere personer har drivet med den visuelle grafikken, men vi har i hovedsak brukt for-loops med funksjoner for å endre innholdet i matrisene. Et eksempel er når vi kodet den visuelle grafikken til speedometeret, slik sett i figur g her har vi brukt lister i lister og dictionaries til å endre speedometerviseren for hvert nivå, slik kommentering i koden viser. Speedometer matrisen består av et speedometer uten en viser.

```python
def plot_ranges(index):
    # Lister over alle speedometer nivåene fra 0 til 6
    color_ranges = [
        [(44, 45), (51, 52)],
        [(41, 45)],
        [(44, 45), (35, 36), (26, 27), (17, 18)],
        [(44, 45), (36, 37), (28, 29), (20, 21), (12, 13)],
        [(43, 44), (36, 37), (29, 30), (22, 23)],
        [(43, 47)],
        [(43, 44), (52, 53)],
        ]

    # Bytter ut pixlene i speedometeret med,
    # hvilket speedometer nivå som vises
    pixels = [v for v in speedometer]
    for (start, end) in color_ranges[index]:
        for i in range(start, end):
            pixels[i] = r

    # Clearer og printer ut speedometeret for hvert nivå,
    # helt til den når siste nivå for angitt level
    sense.clear()
    sense.set_pixels(pixels)
    time.sleep(0.4)

# Hastigheten på teksten som printes
speed = 0.04

# Sier hvilken speedometer nivåer som skal printes,
# avhengig av hvilket level vi befinner oss i
levelPixelRange = {
    1: 3,
    2: 5,
    3: 7,
    }

# Printer ut meldingen av hvilket level vi befinner oss i,
# og alle nivåene på speedometeret
print_speedometer("Lvl " + str(level) , levelPixelRange[level])

time.sleep(1)
```

Figur g - versikt over en av flere grafikk-koder

Debug-statement

Tidlig i prosjektet hadde vi en «DEBUG»-konstant som fikk programmet til å kjøre i terminal i stedet for på Raspberry Pi. Dersom «DEBUG» var satt til «True» ville ikke de Raspberry Pi-spesifikke bibliotekene bli importert, og deler av koden ble utelatt i flytkontroll. I stedet ble skjermbufferen printet til terminal vha. en egen «debug_print»-funkjon.

Programstruktur

Programmet er strukturert på ganske konvensjonelt vis, og vi har forsøkt å følge Pythons standard, PEP-8. Fraksjonering av koden i funksjoner er både god programmeringspraksis og hjelpsomt for å fordele arbeid til gruppemedlemmene. Seinere i prosjektet ble det introdusert klasser og deler av koden ble refaktorisert.

Bibliotekimporteringer

Øverst i programmet importerer vi moduler med funksjonaliteter vi trenger. Vi har forsøkt å holde oss til Pythons standardmoduler, noe som gjør det enklere å sette opp.

Konstanter

I toppen av programmet er det definert konstanter som ikke skal endres i løpet av spillet. Dette hjelper med å holde programmet for øvrig fri for tall, såkalte «*magic numbers*», som hjelper med lesbarhet og vedlikehold. Bildet under viser noen av konstantene.

```
38  FPS = 20
39  FRAME_DURATION = 1 / FPS
40
41  FUEL_SPAWN_CHANCE = 2
42  FUEL_DECREASE = 23
43  FUEL_FREQUENCY = 8
```

Figur h - Noen av konstantene i programmet

Klassedefinisjoner

Seint i prosjektet ble noe av koden reformatert til å bruke klasser. De to klassene, car og player, er praktiske siden vi ønsker å instansiere mange biler og spillere når vi kjører i flerspillermodus.

Funksjonsdefinisjoner

Det finnes i hovedsak tre typer funksjoner vi har brukt i programmet vårt.

- Selvstendige funksjoner som printer til skjermen, spiller av lyd etc. Noen av disse tar inn argumenter for å endre funksjonaliteten til den selvstendige funksjonen
- Grafikkfunksjoner som endrer «*state*» til skjermbufferen og returnerer den modifiserte bufferen
- Hjelpefunksjoner for små, konkrete oppgaver slik som å hente gyroverdier og regne ut bilens posisjon

Dette deler koden opp i håndterlige stykker som gjør det lettere å lese og vedlikeholde. Siden mange av funksjonene brukes flere plasser i koden hjelper dette med å sammenføye deler av koden som ellers ville dukket opp mange plasser.

Eventloop

Hovedfunksjonen «*main*» består av to hoveddeler:

- Initialisering av variabler og oppstart av spillet
- En loop som kjører så lenge spillet varer

Når koden i oppstartssekvensen har kjørt starter loopen. Her har vi brukt en «*while*»-loop og deklarert en iterator manuelt. Iteratoren øker med én for hver *ramme* i spillet. Det meste av logikken i spillet, både grafisk og spillmekanisk, er skrevet så de er avhengige av denne iteratoren.

Hendelser som kan skje i løpet av en ramme inkluderer:

- Legg til en ny port som du skal kjøre igjennom
- Legg til en ny drivstofftønne som du kan plukke opp
- Drivstoffnivået synker

Hvorvidt disse hendelsene skal skje eller ikke i løpet av en ramme er definert med restdivisjonsoperatoren i Python. På bildet under ser du eksempel på dette, der mengden drivstoff, «*fuel»,* synker hver gang iteratoren er delelig på «*FUEL_DECREASE»,* som er en konstant satt til 23.

```
592         #Fuelnivået synker hver FUEL_DECREASE 'te gang
593         if iterator % FUEL_DECREASE == 0:
594             if fuel_already_taken == False:
595                 fuel -= 1
```

Figur i - Eksempel på bruk av iteratoren for flytkontroll

Det er også noen hendelser som skal skje hver eneste ramme, disse inkluderer:

- Lag en ny, tom skjermbuffer (beskrevet i større detalj under «*Grafikk»)*
- Oppdater verdier fra gyroen
- Kollisjonsdeteksjon mellom bilen og porter
- Kollisjonsdeteksjon mellom bilen og drivstofftønner
- Oppdater plassering av spillelementer og legg de til i skjermbuffer
- Oppdater grensesnittelementer, slik som poengindikatoren og drivstoffindikatoren
- Printer det som står i skjermbufferen til sensehatten
- Sjekker om du er tom for drivstoff

- Sjekker om du har nok poeng til å gå til neste nivå

I slutten av loopen blir det lagt til en konstant forsinkelse. I programmet har vi satt denne slik at vi får 20 rammer i sekundet.

Inngangspunkt

Nederst i programmet er inngangspunktet. Derfra blir all koden i programmet kjørt. Den første linja i figur j sørger for at programmet ikke kjører dersom det blir importert inn til en annen pythonfil. Øverst i inngangspunktet initialiserer vi globale objekter.

```
985  if __name__ == "__main__":
986      player_database = PlayerDatabase()
987      api_controller = ApiController(player_database=player_database)
988      pixel_buffer = PixelBuffer(api_controller=api_controller)
989
990      # use command "sudo python3 Milepael_2.py host" to host multiplayer
991      if "host" in sys.argv:
992          api_controller.start()
993
994      main(
995          player_database=player_database,
996          api_controller=api_controller,
997          pixel_buffer=pixel_buffer,
998      )
```

Figur j - Inngangspunktet til programmet

Implementering

Sense HAT har bare 8x8 piksler, dette gjør tegning av flere tall på en skjerm, eller gode visualiseringer utfordrende. Dette har vi jobbet rundt ved å finne gode løsninger på forskjellige utfordringer. Bilen («Hype one»), er en eneste piksel, samme går for drivstoff, og portene du kjører gjennom, består av to. For å kunne tegne en poengsum i egentlig tid, lagde vi en binærteller, dette tar lite plass og piksler, men er fult leselig (se figur d).

I spillet vårt bruker vi sensor data fra gyroen (hentet fra sense HAT), av denne sensorverdien, henter vi kun «pitch». Dette gjør det intuitivt å styre bilen, ettersom det bare er å vinkle Rasberry den veien du ønsker, og bilen kjører dit. Et problem som blir mer dominant etter en lengere kjøretur med «Hype one», er at sensorverdiene begynner å bli «feil» (sensorverdiene «drifter»). Resultatet er at bilen blir mye vanskeligere å manøvrere, ettersom dette ikke er noe vi får endret på, har vi heller tatt det med i betraktning når vi jobbet med vanskelighetsgraden for spillet. At det blir vanskeligere å manøvrere «Hype one» etter en stund, har vi konkludert kommer av for varme dekk, og en sliten Arne Midjo.

Multiplayer

Basert på eksperimentet med websockets i pi-color-picker, valgte vi å legge til multiplayer i spillet. For at hele gruppen skulle forstå koden som lå bak ble det brukt vanilla JavaScript og minimal html. Nettsidefilene er hostet av flask, ved å peke apiet til "web" mappen som inneholder filene for klienten. Dette vises i figur k.

Figur k - Prosjektstruktur

På pc kan man bruke piltaster for å flytte bilen sin, men på devices som har innebygde posisjoneringssensorer, som pien kan disse aktiveres med en knapp på nettsiden, som vi ser i figur l. Hvordan vi bruker posisjoneringssensorene til å gi input til spillet ser vi i figur m.

```
109     if(enableGyro) {
110         window.addEventListener('deviceorientation', onOrientation)
111     } else {
112         window.removeEventListener('deviceorientation', onOrientation)
113     }
```

Figur l - Aktivering av mobilenhetens gyro

```
93  function onOrientation(event){
94          // const x = event.alpha
95          const y = event.beta
96          // const z = event.gamma
97          const newCarPosition = carPositionFrom(y)
98          if(newCarPosition != carPosition) {
99              carPosition = newCarPosition
100             emit('move_to', carPosition)
101         }
102 }
```

Figur m - Bilens posisjon med input fra nettleser

Siden orientation events kun kjører i HTTPS sider som en sikkerhetsbarriere, var konfigurasjon av SSL sertifikater nødvendig. Dette settes opp med kriterier vist i figur n.

```
port=443
ssl_certificate_folder = "/etc/letsencrypt/live/epstin.com/"
context = (ssl_certificate_folder + "cert.pem", ssl_certificate_folder + "privkey.pem")
```

Figur n - Oppretting av ssl sertifikat

I JavaScript benytter vi oss av den elegante syntaksen vi ser i figur o til anonyme funksjoner for å lage kode som er enkel å skjønne, samtidig som, den ikke gjentar seg selv for mye.

```
27  window.addEventListener('keydown', (event) => {
28      if (!event.repeat) {
29          switch (event.key) {
30              case "ArrowLeft":
31                  emit('move_left')
32                  colorize("left", "red")
33                  // Left pressed
34                  break;
35              case "ArrowRight":
36                  emit('move_right')
37                  colorize("right", "red")
38                  break;
39              case "ArrowUp":
40                  // Up pressed
41                  break;
42              case "ArrowDown":
43                  // Down pressed
44                  break;
45          }
46      }
47  })
```

```
1   const socket = io(location.origin)
2
3   connectionStatusText = document.getElementById("connection-status")
4   connectionStatusText.style.color = "orange"
5
6   socket.on('connect', () => {
7       connectionStatusText.innerHTML = "Status: Connected"
8       connectionStatusText.style.color = "green"
9
10  });
11
12  socket.on('disconnect', () => {
13      connectionStatusText.innerHTML = "Status: Disconnected. Refresh page to reconnect"
14      connectionStatusText.style.color = "red"
15  });
16
17  function emit(event, data) {
18      if (socket.connected) {
19          socket.emit(event, data)
20      }
21  }
```

Figur o - Styring av spill i java og oppsett av socket

Siden Python er et svært begrenset språk, er ikke anonyme funksjoner praktiske i bruk. Derfor bruker SocketIO et decorator-pattern for å sette opp event listeners slik vi ser i figur p. Decorators er funksjoner som brukes med @function_name syntaksen for å legge ekstra funksjonalitet til en normal funksjon. I dette tilfelle registrerer vi funksjonen on_socket_move_to som en event handler for eventen «move_to». Eventen emittes øverst til høyre på denne siden i onOrientation().

```python
@socketio.on('move_to')
def on_socket_move_to(json):
    # Json is in this case an int sent by the ws client
    move_car_to(json)
```

Figur p - SocetIO event listener i Python

Siden bilen vår kun beveger seg fra side til side, har vi totalt 8 posisjoner den kan være i. Dette er betydelig mindre enn rotasjonsposisjonen til telefonen, som i dette tilfellet har 79 verdier (-39 til 39). For å spare båndbredde er det derfor bedre å beregne den nøyaktige posisjonen til bilen på telefonen, før dette sendes til pien. Dette oppsettet er vist i figur q, som returnerer posisjonen tilbake til spillet. Dermed trenger vi kun å sende informasjon når det skjer en endring, som gir oss lavere responstid, og mindre batteribruk. Når pien kjører koden, kan man åpne grensesnittet på https://pearpie.is-very-sweet.org/site/index.html

```
70  const DEADZONE = 5
71  const ANGLE_SPAN = 40
72  const MAX_ANGLE = ANGLE_SPAN * 2
73  const CAR_MAX_POSITION = 7
74  const CAR_MIN_POSITION = 0
75
76  let enableGyro = false
77  let carPosition = 3
78
79  function carPositionFrom(angle) {
80      if (angle > ANGLE_SPAN) {
81          return CAR_MAX_POSITION
82      }
83      if (angle < -ANGLE_SPAN) {
84          return CAR_MIN_POSITION
85      }
86      if((angle > -DEADZONE) && (angle < DEADZONE)) {
87          return carPosition
88      }
89      const absoluteAngle = angle + ANGLE_SPAN
90      return Math.round((absoluteAngle / MAX_ANGLE * (CAR_MAX_POSITION - CAR_MIN_POSITION)) + CAR_MIN_POSITION)
91  }
```

Figur q - Utregning av posisjon

Lyddesign

Midjo GP har en kjenningsmelodi inspirert av Nintendo-stilen. Den er repeterende og gjenkjennelig, slik at spillerne skal huske melodien til enhver tid. Kjenningsmelodien er produsert i Ableton Live 10, med kun 2 instrumenter. «Repro-5» er synthen vi brukte til de melodiske delene i kjenningsmelodien. Denne synthen er en digital replika av den anerkjente «Prophet-5» synthen. Til rytmedelen brukte vi «Magical8bitplug2», et gratis instrument som fungerer som en 8-bit digital synth.

Spillet er har også en rekke lydeffekter som «Low Fuel», «Game Over», «+1 Fuel», «Crash» og «1+ Point» Disse er produsert i samme programvare og med samme instrumenter, for å beholde kontinuiteten i lydbildet. Lydeffektene er med på å gjøre spillet mer spennende, og gir også spilleren mestringsfølelse av å høre poengene ramle inn. Vi bruker enkle lyder, men lyder som gir mening ut ifra handlingene på spillet. Når bilen for eksempel har lite drivstoff igjen spilles en alarmliknende lyd, og når bilen krasjer spilles en litt brå og ubehagelig lyd, og når man taper spillet hører man en kort og trist atonal melodi. Derimot når man opptjener poeng og samler drivstoff, hører man en motiverende note, som gir utrykk for at noe positivt skjer.

Det blir også brukt motorlyder vi har hentet fra et lydbibliotek. Disse motorlydene er fra en Aston Martin sportsbil og skal i dette spillet være lyden av «Hype One». Motorlydene brukes i sammenheng med speedometergrafikken i spillet, og er med på å gi spilleren en forståelse av at «Hype One» er en bil og ikke et fly, romskip, båt, eller elbil.

Diskusjon

I dette gruppeprosjektet har vi programmert et fullt fungerende spill for Rasberry Pi ved hjelp av Python og GitHub, samt utviklet en multiplayer ved hjelp av HTML, CSS og JavaScript. Vi har lært å programmere sammen i en gruppe, fordele arbeidsoppgaver og sette tidsfrister. Vi har implementert musikk og lydeffekter, samt flere intuitive grafikkløsninger for å få et helhetlig spill som er utfordrende, moro og visuelt tilfredsstillende.

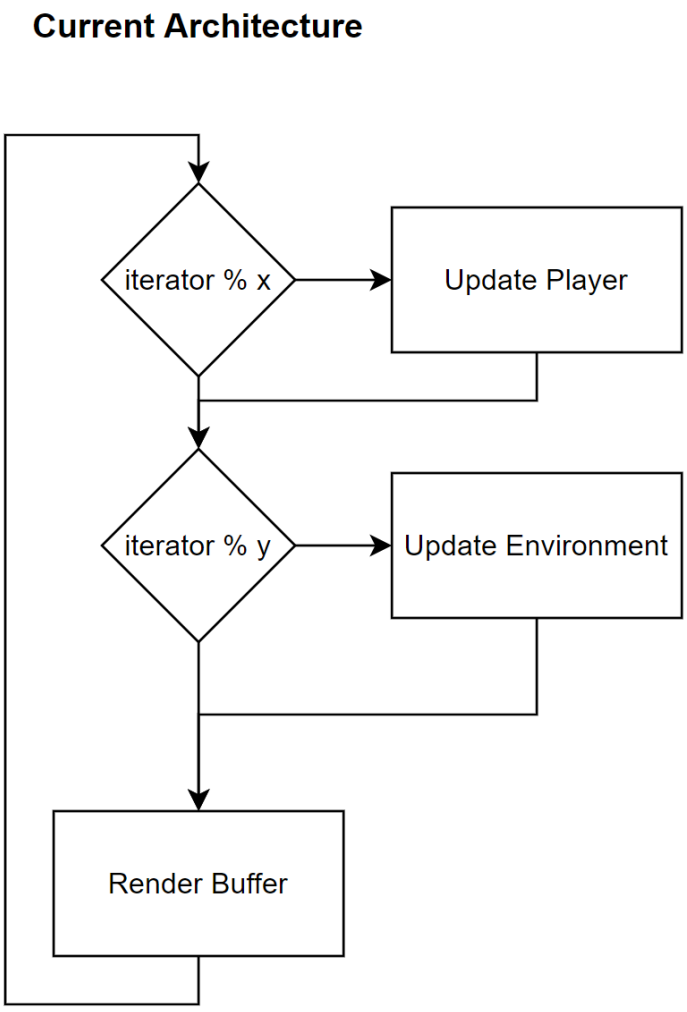
Visse abstraksjoner vi gjorde i starten viste seg å være nyttige. For eksempel ble flerspillerimplementasjonen lettere siden skjermbufferen vi brukte både kunne sendes over nett og printes til SenseHat. Bruken av konstanter gjorde innstillinger lettere, blant annet endring av vanskelighetsgrad i tre nivåer. Vi kunne nok tjent på å skrive programmet i en mer objektorientert stil, med klasser og objekter. Siden dette ikke var en del av pensum holdt vi oss borte fra det, men spesielt etter flerspillerkoden kom inn ble det åpenbart at koden nok ville blitt refaktorisert dersom prosjektet gikk over lengre tid.

Vi har kun én pythonfil, dette kunne enkelt blitt delt opp i biter slik som *multiplayer.py, graphics.py* og *main.py*, dersom prosjektet skulle vokse større. For simpelhets skyld holdt vi alt i samme fil.

Vi kunne fått spillet til å starte automatisk når Pi'en starter ved å lage en *systemd-service.* Det hadde vært praktisk så vi ikke hadde trengt å koble til med SSH for å starte spillet.

Selv om WebSockets er enkle å jobbe med, er de ikke den mest effektive måten å sende hyppig info. Når man kun sender kommandoer fra grensesnittet og minimal info som score, drivstoff og reset signal, fungerer websockets fint. Når man derimot ønsker å sende store mengder info ved korte tidsintervall, begynner WebSocket å vise svakheten sin. Over tid merker man at spillet blir mindre og mindre responsivt i multiplayer. Dette gir oss et behov for å introdusere en tredje teknologi i prosjektet. Mulige kandidater til dette kan inkludere WebRTC, Streams, og direkte bruk av tcp sockets. Dette kan eventuelt være en god ide for fremtidige prosjekter. Det vi kan konkludere fra dette er at det er bedre å være teknologi-agnostisk, i stedet for å pushe for ett universelt rammeverk. På samme måte det kan være fristende å bruke en skralle som en hammer, vil man som oftest belønnes for å bruke en teknologi til det den ble skapt for.

Når det kommer til valg av arkitektur er det visse steg som i ettertid kunne forenklet koden betraktelig. I stedet for en iterator som bestemmer hvor ofte forskjellige skjermelementer skal oppdateres, kunne vi tatt en mindre globalisert strategi. Problemstillingen som måtte løses var at spillerens horisontale bevegelse (20FPS) skal kunne oppdateres oftere enn bevegelsen til resten av spillet (2FPS). Den eksisterende løsningen teller antall loops og bruker modulus for å oppdatere resten av spillet ved riktig tidspunkt. Dette skapte en rekke forvirringer og problemstillinger, samt begrensninger for senere utvidelse. Figur s viser denne løsningen.

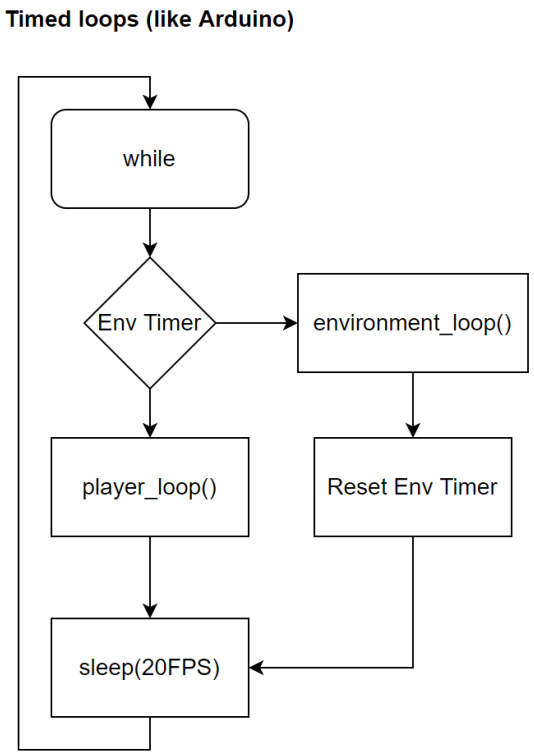


Figur r - Nåværende struktur

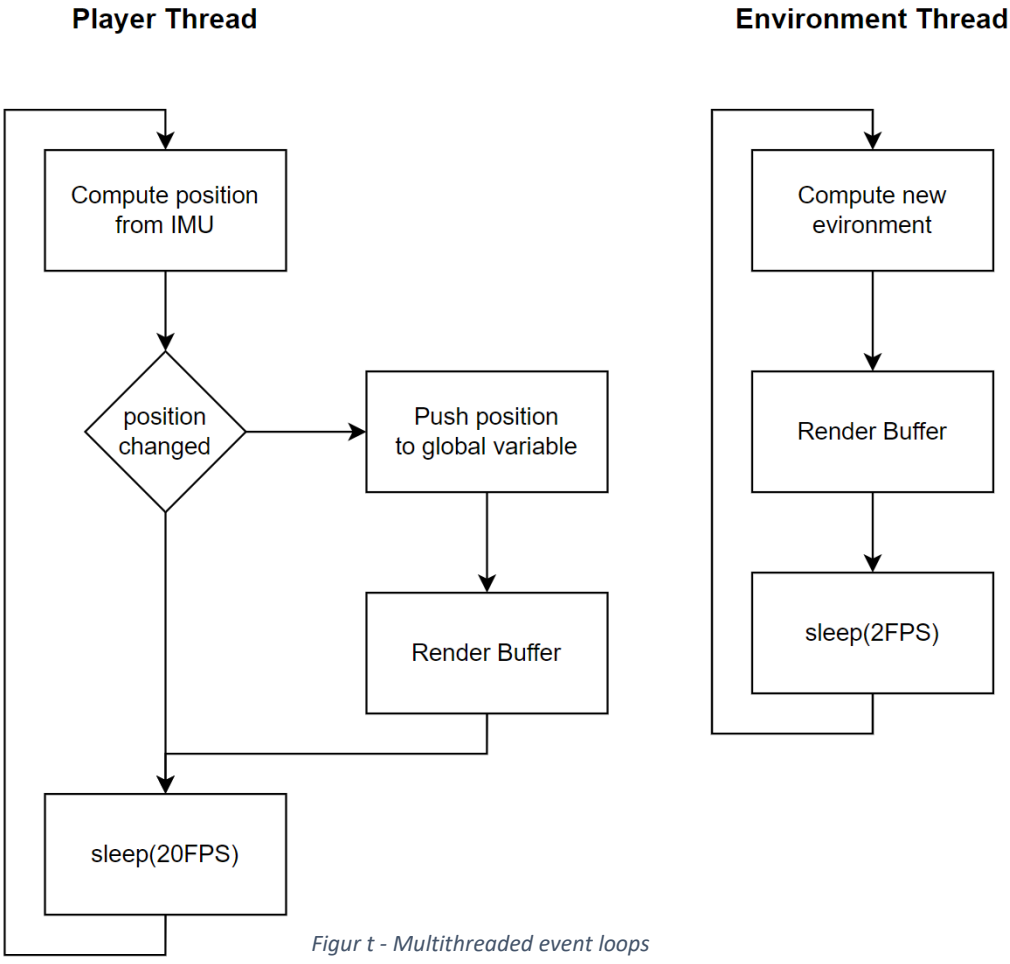
En alternativ løsning kan være å starte parallelle tråder, hvor IMU data kontinuerlig avleses, og sendes til en funksjon som oppdaterer en global variabel, samtidig som den umiddelbart pusher endringen til bufferen, hvis spillerens posisjon endrer seg (i stedet for å pushe endringer når vinkelen fra IMU endrer seg). Dette hadde også redusert arbeidsmengden på pien, som vil føre til mindre batteribruk. Som en liten bonus kunne vi dermed brukt time.sleep() for hver enkelt tråd, som skaper en mer konkret og forståelig kodeflyt. I en egen tråd kan da hoved loopen kjøre, som flytter de andre elementene på skjermen. Figur t på neste side viser denne løsningen.

En slik event basert arkitektur vil ikke bare føre til mer effektiv kode, men også gi en mer leselig og lett forståelig struktur. Dette vil også gjøre det enklere å gjøre endringer i ettertid, fordi forskjellige ansvar er separert til forskjellige loops.

Ved å isolere forskjellige arbeidsoppgaver, er det enklere å gjøre endringer i ettertid, uten å være bekymret for å påvirke andre deler av koden. Ett annet alternativ til parallelle tråder som fortsatt gir samme fordelene, er å kjøre de respektive loopene sekvensielt med en timer for funksjonen som inneholder oppdateringer som kun skal skje 2 ganger i sekundet. De abstrakte flytskjemaene viser de forskjellige arkitekturene som vi nettopp diskuterte. Figur r viser denne løsningen.



Figur s - Sekvensiell timed loops



Figur t - Multithreaded event loops

Konklusjon

Vi er fornøyde med sluttresultatet av de tre milepælene. Det har vært lærerikt å jobbe sammen på et programmeringsprosjekt, og for flere av oss var dette det første samarbeidsprosjektet med kode. Vi ser fram til å utvikle mer programvare i framtida, for dette prosjektet har gitt mersmak.

Referanser

pythonhosted.org. (2021, 12 06). *API Reference*. Hentet fra pythonhosted.org: https://pythonhosted.org/sense-hat/api/

Raspberry Pi. (2021, 12 06). *Getting started with the sense hat*. Hentet fra projects.raspberrypi.org: https://projects.raspberrypi.org/en/projects/getting-started-with-the-sense-hat/

Raspberry Pi. (2021, 12 06). *sense-hat*. Hentet fra raspberrypi: https://www.raspberrypi.com/products/sense-hat/