

OBLIGATORISK OPPGAVE 3

INF-1400-Objekt-Orientert programmering

7. april 2019

Martin Soria Røvang
Universitetet i Tromsø

Inneholder 12 sider, inkludert forside.

Innhold

1	Introduksjon	3
2	Teknisk bakgrunn	3
3	Design	3
3.1	Game	3
3.2	Player1 og Player2	3
3.3	Bullets	5
3.4	Walls	6
3.5	Fuel	6
4	Implementasjon	7
5	Diskusjon	7
6	Evaluering	8
7	Konklusjon	9
8	Part II	10
8.1	1	10
8.2	2	10
8.3	3	10
8.4	4	10
8.5	5	11
9	Appendix	12
10	Referanser	12

1 Introduksjon

I denne oppgaven ble en klonen av spillet *Mayhem* laget. Dette er et multiplayer-spill som består av to romskip som skal skyte på hverandre og passe på at de har nok drivstoff til å motstå gravitasjonen. Spillerne vil miste poeng hvis romskipet blir skutt ned eller kræsjer, og man får poeng hvis de skyter ned motstanderen sitt romskip.

2 Teknisk bakgrunn

- *Arv*: I denne oppgaven arves det fra pygame sin *Sprites* og mellom egne definerte klasser.
- Initialisering direkte fra en annen klasse, f.eks *Player1.__init__(self)* i *Player2* sin *__init__*.
- *Wrappers*: Wrapper er en funksjon som tar inn en funksjon for så gjøre noe rundt den funksjonen man tar inn, som f.eks finne ut hvor lang tid funksjonen brukte. Denne wrapperen kan brukes med en dekorator *@* rundt funksjonen man vil ta tiden av.

3 Design

3.1 Game

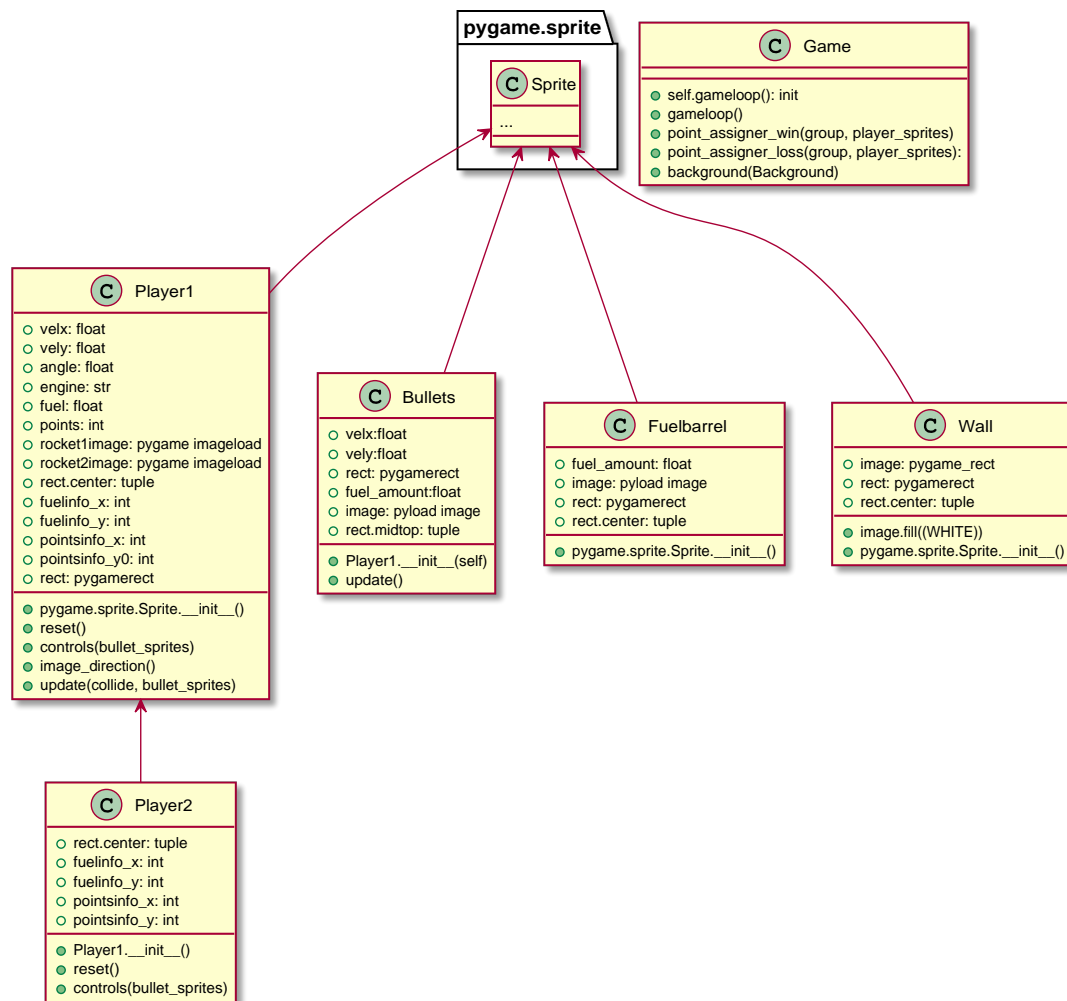
Klassen *Game* er klassen som er *hjernen* i programmet. Denne kobler sammen alle de andre klassene og funksjonene slik at vi får en logikk i spillet. Denne klassen vil ta for seg poeng fordeling ved å endre *self.points* attributten til spillerne. *Self.points* brukes som variabelnavn i pygame sin tekst til skjerm funksjon slik at man til enhver tid kan se poengene til spillerne. Det er i denne klassen *gameloop* metoden er, denne metoden vil kjøre løkken som oppdaterer objektene (spillerne, kuler osv.) og tegner dem ut på skjermen. I denne prosessen brukes pygame sin *spritegroup.update* og *.draw* funksjoner. Disse oppdaterer og tegner alle objektene som har blitt lagt inn i spritegruppen (liste).

3.2 Player1 og Player2

Vi har to klasser for spillerne, *Player1* og *Player2*. Spiller 2 arver fra spiller 1 fordi de er nesten helt like, men bruker andre knapper for å styre romskipet, og har forskjellige plasseringer av poeng/drivstoff informasjon. I denne klassen initialiseres bildene som romskipet skal ha med pygame sin *load* metode *pygame.image.load*. Dette er to forskjellige bilder som viser romskipet med motor av og på, disse er vist i figur(2). Som man kan se på bildene så er ikke romskipet rektangulært, dette vil medføre til litt rar kollisjon da vi bruker rektangulær kollisjons-boks (hitbox).

Spillerklassen har en metode som resetter startverdiene og legger til et minuspoeng, denne blir brukt hver gang et romskip blir skutt ned eller om man kræsjer.

Kontrollering av romskipene blir gjort fra *controls* metoden ved hjelp av pygames sin *get_pressed* metode, den henter ut knappene som blir trykket på. Dette ble så brukt til å sjekke om knappene som er valgt til å skyte, fly og akselerere blir brukt. Ved bruk av *eval* funksjonen kan man lage en variabel



Figur 1: UML diagram av prosjektet, nesten alle klassene arver fra pygame sin spriteklasse for å kunne bruke *spritegroups*, dette vil gjøre det enklere å avgjøre kollisjoner.



Figur 2: Modell av romskipene med motor av og på.

for knappene som skal brukes, slik at dette kunne endres i *config.py* filen.

Rotasjon av skipet ble gjort ved bruk av pygame sin *pygame.transform.rotate* metode, den tar inn en vinkelverdi som blir gitt fra *controls* metoden.

Hvis motoren er av vil gravitasjonen dra skipet i y-retning, og hvis motoren er på vil gravitasjonen

$$\mathbf{a}(x, y) = \begin{cases} (0, g), & \text{engine off} \\ (0, 0)\alpha, & \text{engine on} \end{cases} \quad (2)$$

Figur 3: Akselerasjonen til romskipene

skru av og en fart vil bli lagt til i x- og y-retning basert på hvilken vinkel romskipet er i. Dette blir styrt med en sinus- og cosinus-funksjon, med vinkelen som argument. Dette vil skje kun hvis man har nok drivstoff så her er det også en *if* test. Hastigheten er gitt matematisk i ligning(1).

$$\mathbf{v}(x, y) = \begin{cases} (0, g)\Delta t, & \text{engine off} \\ (\cos(\theta), -\sin(\theta))\alpha, & \text{engine on} \end{cases} \quad (1)$$

Her er g gravitasjonen, α er en konstant (hvor stor hastighet), θ er vinkelen på romskipet og Δt er FPS, slik at hvis man har 60 fps vil dette skje 60 ganger i sekunder $\Delta t = 60 * \text{sekunder}$. Når motoren er på har det blitt valgt å fjerne akselerasjonen (tidsavhengigheten) for å gjøre det lettere å styre romskipet, dette ble gjort ved å sette hastigheten lik og ikke summe på som vi gjør med gravitasjonen.

Player2 klassen er så og si den samme, men med annen andre knapper for å kontrollere romskipet og annen posisjon for drivstoff og poeng informasjon, der player1 har informasjonen i venstre topphjørne og player2 har det i høyre topphjørne.

3.3 Bullets

Kulene skal komme ut i samme vinkel som romskipet, derfor har vinkelen til romskipet blitt brukt som argument i en cosinus- og sinus-funksjon i hastigheten til kulene. Når en spiller skyter ut kulene legges det en kule sprite i en spritegroup. Dette blir brukt sammen med `sprite.groupcollide` funksjonen til pygame for å teste kollisjon mellom vegger og spillere. Denne funksjonen vil returnere en dictionary med sprites som kolliderer, og deretter fjerne de fra `sprite group`-listen, hvis man har lagt inn dette som argument. Denne return dictionaryen blir brukt til å tildele eller fjerne poeng til spillerne. Kulene ser som vist i figur(4).

**Figur 4:** Kuler som kommer ut av romskipet.

3.4 Walls

Veggene i spillet er også sprites som har blitt lagt inn i spritegroups slik at at man kan teste for kollisjon mellom vegg, spillere og kuler. Her skal ingen av veggene forsvinne etter en kollisjon, derfor legges dette inn som argument i `spritecollide`-funksjonen til pygame at både kuler og romskip skal bli fjernet, men ikke veggene.



Figur 5: En av veggene som er i spillet(den hvite streken).

3.5 Fuel

Drivstoff-fat oppstår i et tilfeldig område på spillskjermen hvert 40 sekund(dette kan endres i `config.py`). Disse vil fylle på drivstoff, og vil da hindre romskipet fra å kræsje i veggen på grunn av gravitasjon(hvis motoren blir brukt da selvfølgelig). Her ble `time` modul fra python sine innebygde moduler brukt for å finne tiden(for når et fat skulle bli lagt ut). `time.time()` vil gi antall sekunder siden denne funksjonen ble kalt på¹, ved å derfor ta differansen mellom to slike kall kan man få tiden det har tatt (slutt-start). Dette ble brukt ved å først kalle `time.time()`, og deretter ha en ny midlertidig variabel som har samme verdi som `time.time() + 40`. Herfra blir det testa om `time.time()` har større verdi en denne, når denne endelig får større verdi så legges det ut et drivstoff-fat på et tilfeldig sted og deretter legges det til dette tidsintervallet på den midlertidige variabelen også testes alt på nytt. I figur(6) kan man se et fat med drivstoff ute på spillskjermen.

¹On Windows, this function returns wall-clock seconds elapsed since the first call to this function, as a floating point number,<https://docs.python.org/3/library/time.html>



Figur 6: Drivstoff-fat kan plukkes opp av romskipene, denne vil fylle drivstoffet helt opp når man flyr på den. Fatene spawner/oppstår hvert 40 sekund (dette kan endres i konfigurasjonsfilen).

4 Implementasjon

Koden er skrevet i Python versjon 3.7²

OS: Windows 10

Systemtype: 64-bit OS, x64-basert prosessor

Skjermkort: NVIDIA Geforce 920MX

CPU: Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz

RAM: 4GB

Pygame ³ Version: 1.9.4

Numpy ⁴ Version: 1.14.5

5 Diskusjon

Bruk av pygame sin sprite-klasse gjorde det enklere å håndtere forskjellige spill relaterte problemer som kollisjon. Her var gruppe oppdatering og tegning/draw, og fjerning av objekter allerede implementert. Ett problem som kan oppstå her er at man kan glemme at alle disse funksjonene kan bli ganske krevende da det underliggende i funksjonene er for løkker, og annet som kan være svært systemkrevende. FPS(frames per second) var ikke så bra under testing av spillet så her kunne man prøvd å optimalisere mer. Når man skyter så lager man veldig mange kuler. Dette er fordi at det ikke har blitt lagt inn noen restriksjon på hvor mange kuler som skal komme ut når man bruker skyteknappen, og derfor vil programmet bare skyte av så mange kuler den klarer(ved 60 fps vil det være 60 kuler i sekundet, så her kunne man optimalisert mer med tanke på dette). Laget også en dekorator-mønster for å gjøre det enklere med feilsøking/optimalisering. Her ble det brukt den innebygde time modulen for å finne hvor lang tid det tok å kjøre en funksjon, denne tiden ble summet opp for hver gang denne ble kjørt slik at man finner den kumulative summen. Antall funksjonskall ble også tatt vare på. Hvis man så avslutta programmet ville man se den kumulative summen av tid og antall funksjonskall for de(n) gitte funksjonen(e).

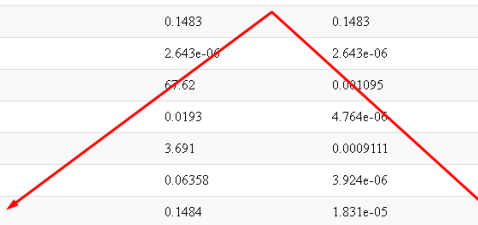
²<https://www.python.org/>

³<https://www.pygame.org/wiki/GettingStarted>

⁴<http://www.numpy.org/>

6 Evaluering

Programmet kjørte ganske tregt, og derfor kunne det ha vært bedre optimalisert. Tar man en titt på figur(7) kan man se at *Player.update* bruker en kumulativ sum på 2.1 sekunder og 8100 kall. Antall kall kan man egentlig ikke gjøre så mye med siden spillet er avhengig av en kjapp oppdatering av verdiene til spillerene, men her kunne man prøvd å optimalisere hvordan update funksjonen fungerer, og eventuell se om det er noen plasser man kunne ha lagt inn generatorer for å spare minne. Hvis man ser litt lengre opp ser man at bakgrunnen brukte veldig mye tid på kun 4051 kall. Dette ble optimalisert ved hjelp av eget diagnostikkverktøy forklart under.



216/1	0.002959	0.002959	74.71	74.71	~0(<built-in method builtins.exec>)
1	2.077e-05	2.077e-05	74.71	74.71	main.py:2(<module>)
1	0.1483	0.1483	74.09	74.09	main.py:39(gameloop)
1	2.643e-06	2.643e-06	74.09	74.09	main.py:13(__init__)
61760	67.62	0.001095	67.62	0.001095	~0(<method 'blit' of 'pygame.Surface' objects>)
4051	0.0193	4.764e-06	66.2	0.01634	main.py:17(background)
4051	3.691	0.0009111	3.691	0.0009111	~0(<built-in method pygame.display.update>)
16204	0.06358	3.924e-06	2.287	0.0001412	sprite.py:452(update)
8102	0.1484	1.831e-05	2.178	0.0002688	Players.py:86(update)
16204	0.04639	2.863e-06	1.072	6.619e-05	sprite.py:464(draw)
4051	0.08073	1.993e-05	0.6905	0.0001705	Players.py:44(controls)
279/2	0.002826	0.001413	0.6161	0.308	<frozen importlib._bootstrap>:978(_find_and_load)
279/2	0.001324	0.0006621	0.616	0.308	<frozen importlib._bootstrap>:948(_find_and_load_unlocked)
263/2	0.002032	0.001016	0.6149	0.3074	<frozen importlib._bootstrap>:663(_load_unlocked)
191/2	0.000939	0.0004695	0.6148	0.3074	<frozen importlib._bootstrap_external>:722(exec_module)

Figur 7: Profil av programmet, her kan man se de hvor lang tid de forskjellige funksjonene brukte og antall kall det var til dem. Her har det blitt brukt *cProfiler* sammen med visualiseringsbiblioteket *snakeviz*. Pilene peker på antall kall og kumulativ sum for *Player.update()* metoden.

Det ble også laget en egen profilerings-dekorator slik at man kunne teste en eller flere funksjoner. Dette ble gjort ved bruk av timer modulen, og ved å lage en "sandwich" med funksjonen som ble testa i midten av to `time.time()` funksjonskall. Med dette kunne jeg dekorere funksjoner jeg ville sjekke kumulativ sum av tid og antall kall på, slik som vist i figur(8).

```

1         @timer
2         def somefunction():
3             .....
4
5
6 >> Cumsum/Calls: {'point_assigner_win': [0.0004951953887939453, 377], '
      point_assigner_loss': [0.002472400665283203, 754]}
7

```

Figur 8: Hvordan dekoratoren ble brukt i koden, og resultat og utskrift gitt at to funksjoner var dekorert.

Et problem her var at alle funksjonene kjørte i "evig tid" fordi det er løkke som må kjøre så lenge man spiller, slik at her var det også implementert en metode som printer ut resultatet, denne må man legge inn sammen med *if* testen som tok av seg å lukke vinduet.

Dette verktøyet brukte jeg for å optimalisere "Background" funksjonen. Før optimalisering fikk man

resultatet vist i figur(9),

```
1      Cumsum/Calls: {'background': [19.079365253448486, 1163]}
2
```

Figur 9: Før optimalisering

og etter optimalisering fikk man resultatet vist i figur(10)

```
1      Cumsum/Calls: {'background': [1.6295523643493652, 1471]}
2
```

Figur 10: Etter optimalisering. Her har vi optimalisert betydelig, da vi har kumulativ tid på 1.62 sekunder med flere funksjons-kall enn før optimalisering.

Dette ble gjort ved å legge til `.convert()` etter at man lastet inn bilde med `pygame.load()`, dette vil formatere pikselene til bilde til å ha samme format som flaten.⁵

7 Konklusjon

I dette prosjektet ble en klonen av spillet *Mayhem* laget med hensyn på objekt-orientert programmering. Ved hjelp av arv fra pygame sin sprite-klasse kunne man enkelt oppdatere, tegne og sjekke for kollisjon mellom alle objekter man hadde laget. Den største utfordringen er å få spillet optimalisert nok til at det var behagelig å spille, noe som ikke har blitt gjort her og dermed kan man forvente lav fps(frame per second). cProfiler ble brukt til å se hvilke funksjoner som var krevende, men dette var noe vanskelig å bruke da det var mange tredjeparts-moduler/funksjoner(pygame, sprites etc), men noe optimaliseringsproblemet ble løst med eget diagnostikk-verktøy diskutert i rapporten.

⁵<http://www.pygame.org/docs/ref/surface.html#pygame.Surface.convert>

8 Part II

8.1 1

En klasse er en slags blåprint for å lage objekter, dette kan ses på som en fabrikk som lager biler, der objekter er bilene og fabrikkene er klassene.

```
1 class fabrikk:
2     def __init__(self, wheelsize, color, motor):
3         wheel.self = wheelsize
4         color.self = color
5         motor.self = motor
6     def functions_that_does_stuff(self):
7         ...
8
9 Bili = (50, 'green', 'RollsRoyce100X')
10
```

I denne kodesnippetten over er et eksempel på fabrikk/bil metaforet.

8.2 2

Arv(Inheritance) er at man kan "kopiere" en annen klasse, og derfra endre på akkurat de funksjonene/attributtene man ønsker, dette kan brukes hvis man for eksempel skal ha en ny klasse som er ganske lik en annen, men man må endre litt på hvordan en/flere funksjon(er) fungerer. Syntaksen for dette er vist i figur(11).

```
1 class Child(Parent):
2     ....
3
```

Figur 11: Child arver fra Parent

8.3 3

Is-a relasjon kommer fra det å arve fra en klasse, og *has-a* kommer fra det å arve fra flere klasser, slik at man arver funksjoner fra både Parent1 og Parent2. Dette blir analogt til at et barn har øyne fra far og nese fra mor. [(p.20-23)Dusty [Oktober 2018]]

8.4 4

Encapsulation er det å gjemme implementasjon bak grensesnittet i programmet ditt, i python kan man ikke blokkere noen ute av deler av koden(utenom å kompilere til .exe eller annet.) så derfor har det blitt laget en konvensjon der man bruker `_` før en funksjon/klasse/variabel som betyr at dette er *privat* og dermed ikke bør endres. Dette kan brukes hvis disse har systemkritiske funksjoner som kan ødelegge programmet hvis disse endres på.

8.5 5

Polymorfisme er det at man har noe som endrer seg basert på hva det skal gjøre. Hvis vi for eksempel har en klasse som er en gjenstand, og denne gjenstanden skal ha muligheten til å kun bevege seg diagonalt. Da vil man ha en metode som heter *move*. Videre vil man at en gjenstand også skal kunne bevege seg, men denne ganger kun i en retning. Herfra kan man arve fra den første klassen og endre på *move* funksjonen slik at objekter fra dette kun beveger seg i en retning. Her har man da polymorfisme fordi nå kan man f.eks kalle *move* på objektene uten å tenke over hva de er for noe også vil de bevege seg med forskjellige regler, dette er også kalt *duck typing*, fra "*if it walks like a duck or swims like a duck, it's a duck.*"[(p.22-23)Dusty [Oktober 2018]].

9 Appendix

10 Referanser

Phillips Dusty. *Python 3 Object-Oriented Programming*. Packt Publishing Ltd., third edition, Oktober 2018.