OBLIGATORISK OPPGAVE 3

INF-1400-Objekt-Orientert programmering

27. mars 2019

Martin Soria Røvang
Universitetet i Tromsø

Inneholder 10 sider, inkludert forside.

Innhold

1 Introduksjon

I denne oppgaven ble en klone 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 fuel for å motså gravitasjonen. Spillerne vil miste poeng når romskipet blir skutt ned eller kræsjer, og man får poeng når 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

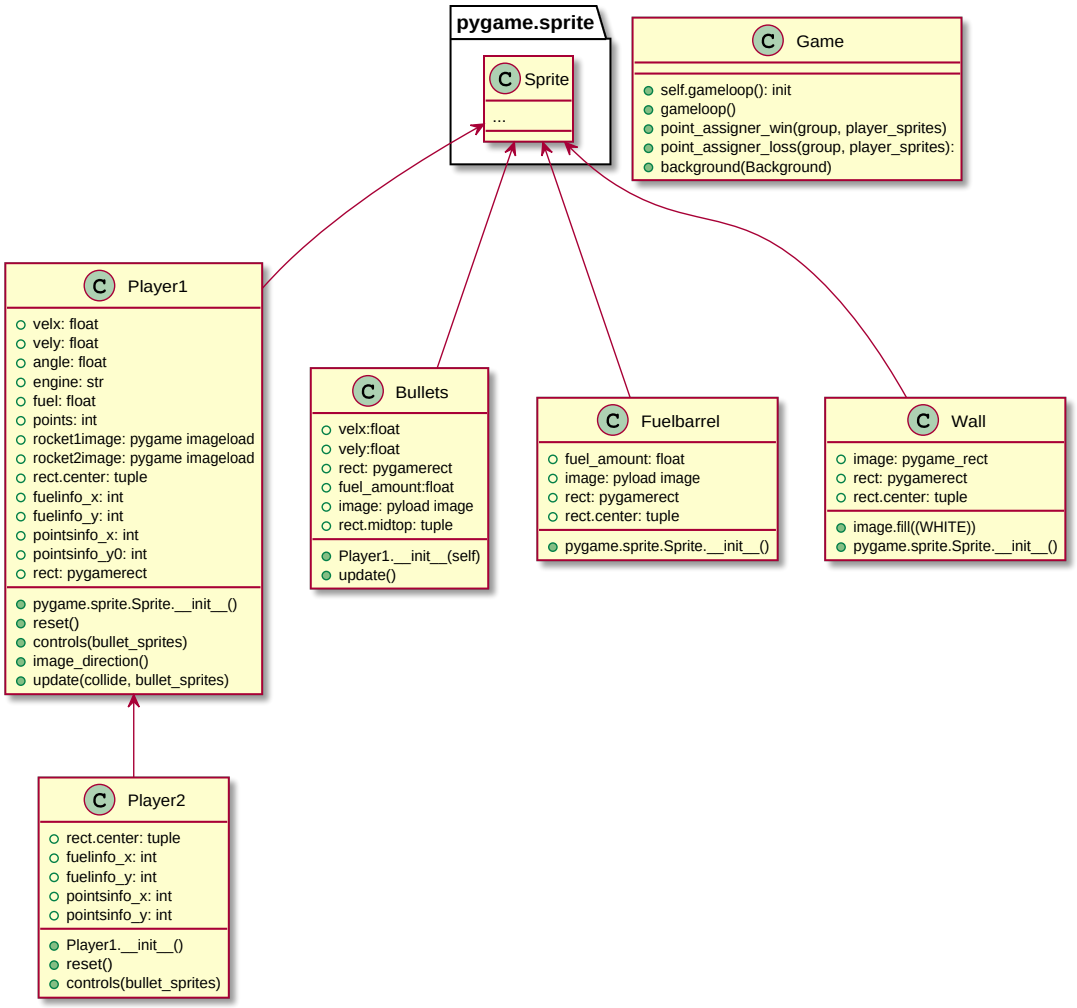
Klassen *Game* er klassen som er *hjernen* i programmet. Denne kobler sammen alle de andre klassene og funksjonene slik at vi får en logikk i spillet. Denne klassen vil ta for seg poeng fordeling til spillerne ved å endre *self.points* attributten til spillerne. Self.points brukes som variabelnavn i 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

Spiller 1 og spiller 2 klassene er delt i to, spiller 2 arver fra spiller 1 fordi de er nesten helt like, men bruker andre knapper for å styre romskipet og annen 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 firkantet, dette vil medføre til litt rar kollisjon da pygame bruker rectangulær 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 osv er i bruk. Ved bruk av *eval* funksjonen kan man lage en variabel for knappene



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

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

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.

Hastigheten til romskipet blir endret når man bruker motoren, hvis motoren er av vil gravitasjonen

dra skipet i y-retning, og hvis motoren er på vil gravitasjonen 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).

$$\boldsymbol{v}(x,y) = \begin{cases} (0, \frac{1}{2}g^2)\Delta t, \textit{ engine off} \\ (cos(\theta), -sin(\theta))\alpha, \textit{ engine on} \end{cases} \tag{1}$$

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 $\Delta t = 60 * sekunder$. Når motoren er på har det blitt valgt å fjerne akselerasjonen (tidsavhengigheten) for å gjøre det lettere å spille.

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 dictionarien vil bli brukt til å tildele eller fjerne poeng til spillerne. Kulene ser som vist i figur(3).

Figur 3: 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 4: 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. Disse vil fylle romskipene helt opp, og vil derfor forhindre romskipet fra å kræsje i veggen pågrunn av gravitasjon. Her ble *time* modul fra python sine innebygde moduler brukt for å finne tiden. *time.time()* vil gi antall sekunder siden denne funksjonen ble kalt[1], ved å derfor differnansen mellom to slike kall kan man få tiden det har tatt (slutt-start). Dette ble brukt ved å første 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(5) kan man se et fat med drivstoff ute på spillskjermen.

Figur 5: 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).

[1] On 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

4 Implementasjon

Koden er skrevet i Python versjon 3.7[2]
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)

6 Evaluering

Programmet kjørte ganske tregt, og derfor kunne det ha vært bedre optimalisert. Tar man en titt på figur(6) 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.

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 "sandwhich" 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(7).

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.

[2]https://www.python.org/
[3]https://www.pygame.org/wiki/GettingStarted
[4]http://www.numpy.org/

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 6: 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*

```
1          @timer
2          def somefunction():
3              .....
4
```

Figur 7: Hvordan dekoratoren ble brukt i koden.

7 Konklusjon

I dette prosjektet ble en klone av spillet *Mayhem* laget med hensyn på objekt-orientert programmering og 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 mye tredjeparts moduler/funskjoner(pygame, sprites etc).

8 Part II

8.1 1

En klasse er en slags blåprint for å lage objekter (fabrikk som lager biler). Objekter er bilene og fabrikkene er klassene.

```
class fabrikk:
        def __init__(self, wheelsize, color, motor):
            wheel.self = wheelsize
            color.self = color
            motor.self = motor
        def functions_that_does_stuff(self):
            ...

Bil1 = (50, 'green', 'RollsRoyce100X')

```

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

8.2 2

9 Appendix

10 Referanser