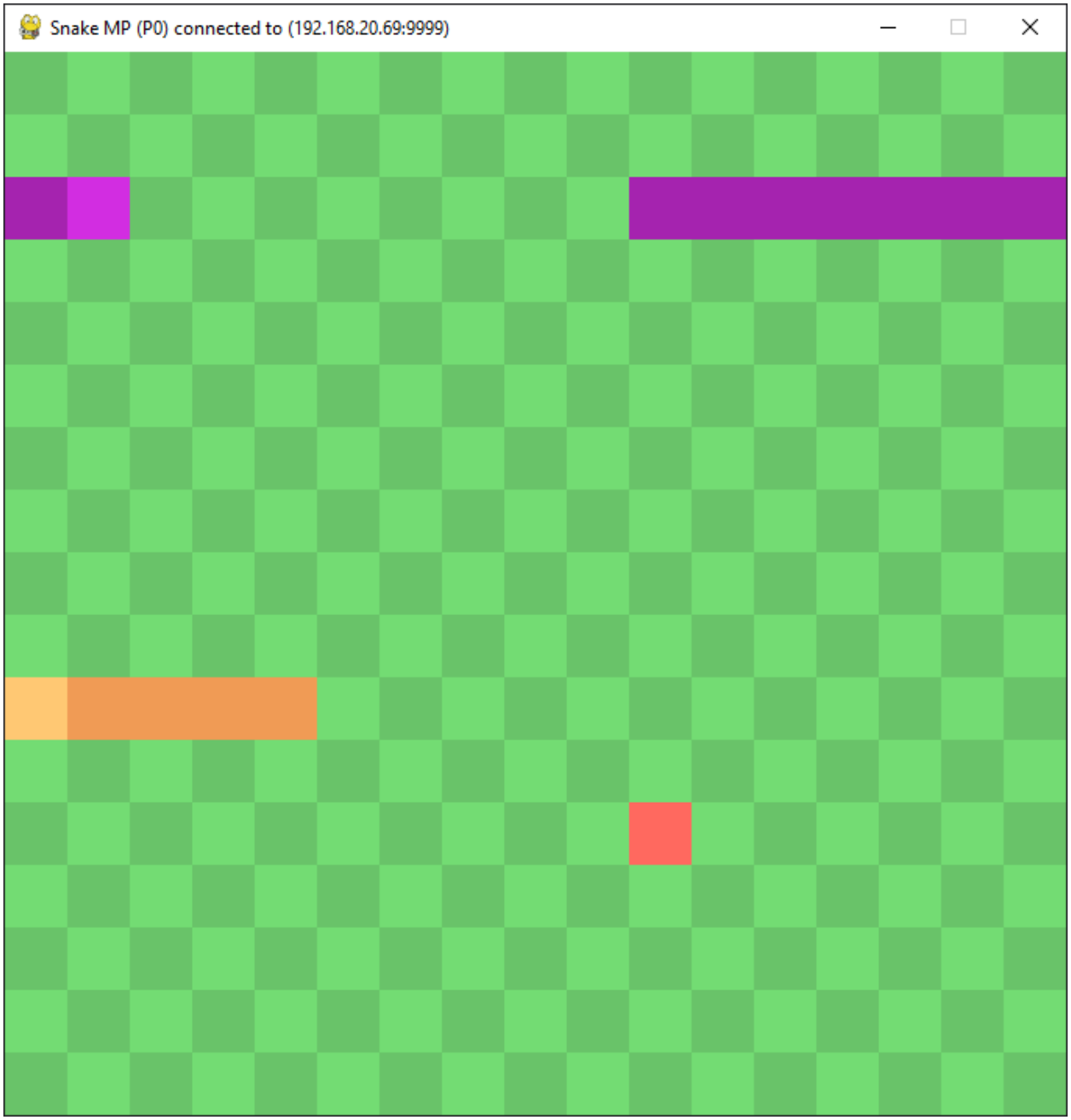
Snake Multiplayer



Vejledet af Lena Erbs

Indholdsfortegnelse

Indledning

Spillet Snake er et klassisk arkadespil, der blev populært i slutningen af 1970'erne og begyndelsen af 1980'erne. Det har altså været et spil i over 50 år. Det blev først introduceret på arkademaskiner og senere til andre platforme som mobiltelefoner (Nokiaen er nok det mest kendte eksempel) og til sidst computere. Målet i spillet er simpelt: styr slangen på skærmen og sørg for at den rammer æblerne. Når slangen spiser æblerne, bliver den længere, og spilleren skal undgå at kollidere med væggene eller sin egen hale. Der er forskellige fortolkninger af spillet. I nogle fortolkninger accelererer slangen, hvilket gør det sværere at styre slangen. Snake er kendt for sin enkle tilgang og evnen til at tilbyde underholdning uden at man skal være særlig god til spil. Det har gjort det til et tidløst og genkendeligt spil på tværs af generationer.
Jeg har så valgt at lave min egen fortolkning af denne klassiker af et spil. Mit twist er et online aspekt, så man kan spille mod sine venner i korte sessioner, da essensen af spillet netop er sjov og kort underholdning. Det vil skubbe det over i retningen af noget Super Smash Bros lignende.

Arbejdsproces

Under udviklingen af hele projektet brugte jeg vandfaldsmetoden.
Vandfaldsmetoden er en lineær og sekventiel tilgang, hvor man inddeler processen i afgrænsede faser, som udføres i sekvens. Først laver man kravspecifikationerne, derefter kommer man op med en plan for, hvordan softwaren skal blive opbygget på baggrund af tidligere nævnte krav.
Så implementerer man det ud fra designet, samt tester om det fungerer, som det skal.
Det sidste step i en virksomheds situation ville være integration og vedligeholdelse, men det er ikke relevant for mig, da jeg laver projektet for mig selv (i forbindelse med gymnasiet).
Derudover har selve opbygningen været meget lineær og iterativ, da spillet skulle være færdigt for, at multiplayer delen kunne implementeres, og multiplayer delen skulle være færdig, før jeg kunne teste applikationen.
Det har givet mig en meget tydelig opdeling af opgaver og funktionaliteter, som har skulle være på plads før jeg har kunnet gå videre:

1. Først - undersøg om det er muligt at lave et multiplayer spil med pygame og forskellige metoder til udviklingen af sådan en applikation.
2. Programmer spillet i pygame.
3. Implementer multiplayer-funktionaliteter.
4. Test-fiks-finpuds hvis der er tid.

Kravsspecifikationer

Første step i processen er opstillingen af krav, for at have en idé om hvad jeg arbejder hen mod. Kravene har jeg inddelt i tre kategorier, hhv. Design-, funktionelle- og performancekrav. Derudover er der også nogle opgavekrav.

Opgavekrav

- Lav et spil med pygame. (*Dette har jeg rykket til funktionelle krav*)

Funktionelle krav

1. Opgave Krav 1 (Lav et spil i pygame)
2. Man skal kunne spille spillet snake
3. Tilføj multiplayer til spillet
4. Lav en funktionel 1v1 spiltype

Designkrav

A. Sørg for at man kan se at det er spillet snake

Performance krav

I. Sørg for at spillet kører stabilt
II. Sørg for at synkroniser spillet mellem klienter

Programmets udvikling og opbygning

Anvendte biblioteker

Her er en kort opsummering af alle bibliotekerne, som jeg bruger, samt lidt af deres funktionalitet, som bliver yderligere uddybet i udviklingsafsnittet til de forskellige elementer af mit program.

Pygame

Pygame er et open-source bibliotek, der anvendes til udvikling af multimedieapplikationer, som f.eks. spil. Det gør det lettere at implementere almindelige funktioner i spiludvikling. Pygame

tilbyder en intuitiv tilgang til at håndtere lyd, grafik og input i spil. Jeg vil i mit projekt bruge det til at konstruere selve spillet. Det skal bruges til håndtering af alt, der skal vises til spilleren.

Socket

Socket biblioteket gør det muligt at lave netværksprogrammering, der tillader kommunikation mellem enheder over et netværk. Det implementerer sockets, som er endepunkter for at sende eller modtage data over et netværk ved hjælp af forskellige protokoller som TCP eller UDP.
TCP giver mulighed for god kommunikation, det kræver en god forbindelse. Men det er til for at være sikker på, at den data, der bliver sendt, bliver modtaget. Eksempler på TCP er FTP (File Transfer Protocol) og HTTP (Hypertext Transfer Protocol) eller web-adgang. UDP er mere bare send data og håb på det bedste, det bliver bl.a. brugt i forbindelse med GeForce Now, da det er vigtigt, at man kan modtage ekstremt meget data, hvor det er underordnet hvis man taber enkelte pakker. Biblioteket giver mulighed for at oprette, forbinde, sende og modtage data gennem netværksforbindelser. Det er afgørende for at skabe client-server-applikationer, peer-to-peer-netværk og andre netværksbaserede systemer. I mit projekt kommer jeg til at anvende TCP - server-client-programmering, til at løse mit krav om multiplayer-funktionalitet. TCP er den bedste løsning for mig, da jeg gerne vil være sikker på at alle data bliver modtaget.

Threading

Threading er et bibliotek i Python, som gør det muligt at køre processer/funktioner i parallel. Normalt når man eksekvere applikationer, så kører de på en tråd, men da jeg skal modtage og sende data samtidig, har jeg brug for at køre det på flere tråde (i parallel).

Time

Time er et bibliotek, som kan alt med tid. Jeg bruger det primært til at sikre at forskellige værdier er sat, samt at begrænse hvor ofte serveren sender data til klienterne.

Regex

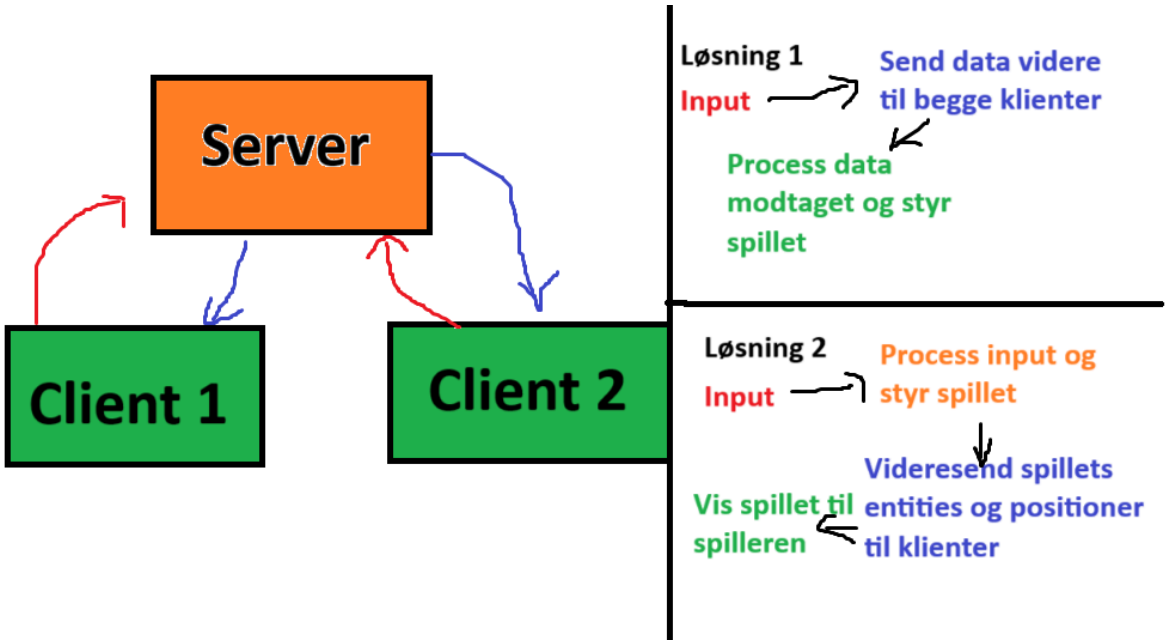
Regex er et bibliotek, som gør det muligt at bruge regulære udtryk.

Random

Random er et bibliotek, som gør det muligt at lave pseudo-*tilfældighed*. Det anvender jeg til at lave tilfældige positioner til æblet.

Udviklingsproces

Der var to overordnede veje, som jeg kunne gå i henhold til udviklingen af en løsning til snake multiplayer: Løsning 1 - jeg kan gøre således at klienterne har spillet, så spillet kører client-side. Det vil indebære, at serveren bare fungerer, som bindeled mellem klienterne. Den videresender altså bare de anmodninger og den data, som klienterne sender. Klienterne vil både fungere som inputs for spillerne, som brugerflade til spillet samt som selve spillet.
Løsning 2 - serveren fungerer som selve spillet, så spillet foregår server-side. I denne løsning fungerer klienterne bare som formidling af spillet til spillerne samt til at sende input fra spillerne til serveren. En mere visuel forklaring kan ses på *figur 1*.



Figur 1 - visualisering af kommunikation mellem hhv. Server og klient.

Der er fordele og ulemper ved begge løsninger, for at finde den optimale løsning her vil jeg skulle have testet begge, men det var der ikke tid til, så jeg bliver nødt til at vælge, hvilken løsning, som jeg mener, er bedst for min applikation.

Fordele og ulemper ved løsning 1: Der skal ikke sendes super meget data mellem serveren og klienterne, da alt spil-logikken foregår hos klienterne, ergo er det kun hver gang der sker et input, at der skal forårsages en kommunikation mellem serveren og klienterne. Det kommer dog med en ulempe, da der i denne løsning skal tages ekstra foranstaltninger for at holde spillet synkront mellem alle klienterne.

Det er en god løsning til spil, hvor man kun bevæger sig, når man laver input. I mit tilfælde skal slangen bevæge sig hele tiden, hvilket kan medføre, at hvis en klient fryser, så ryger klienten ud af synkronisering med de andre klienter.

Fordele og ulemper ved løsning 2: Klienterne kan ikke falde ud af synkronisering med hinanden, da de hele tiden modtager data fra serveren. Til gengæld kræver det, at jeg hvert render-tick skal sende utroligt meget data fra serveren, hvilket kan være enormt krævende og muligvis påvirke performance af applikationen.

Jeg ville gerne, hvis jeg havde tid, have prøvet begge løsninger for at se, hvilken der fungerede bedst, men på grund af det begrænsede omfang af opgaven, vælger jeg at fokusere på løsning 1, da jeg regner med at kunne lave en makeshift løsning på synkronisering.

Udvikling af selve spillet

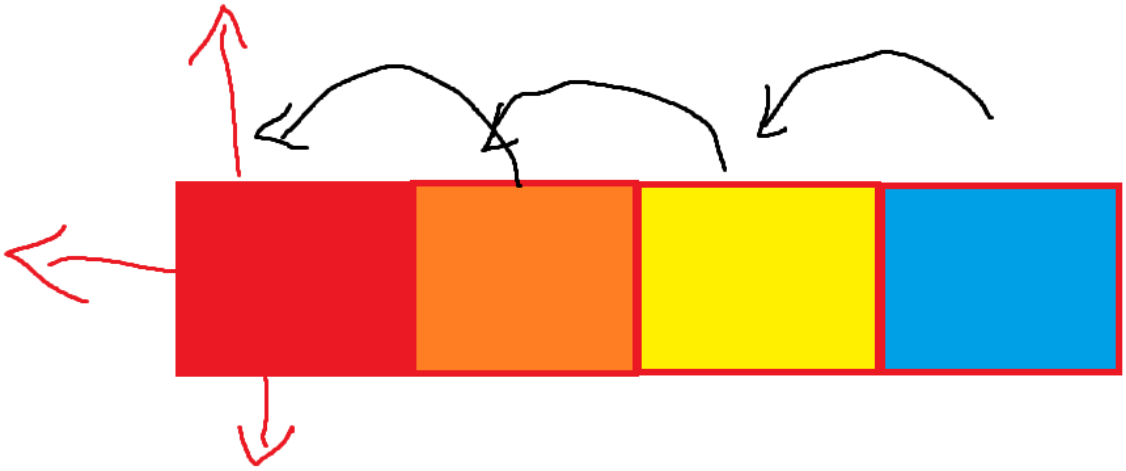
Spillet snake behøver næsten ingen introduktion.
Applikationen skal have to stadier: 'menu' og 'game'.
Alt afhængigt af stadiet skal jeg render forskellige objekter på skærmen og tage imod forskellige inputs. Det bliver løst ved at jeg laver et map med to funktioner som begge returnerer sandt hvis applikationen skal fortsætte med at køre. Deres nøgle er en streng som enten er `'MENU'` eller `'GAME'`. Det gør det muligt for mig hurtigt at ændre hvilket stadie applikationen skal være i.

```
202  LoopMap = {'MENU':menu_loop_logic,'GAME':game_loop_logic}
203  while True:
204      status = LoopMap[STATE_OF_APPLICATION]()
205      if not status:  #   Exit logic
206          print('Ending application...')
207          quit_application()
208          break
209      pygame.display.flip()
210      clock.tick(60)
```

Figur 2 - Application-loop

Kroppens bevægelse

I selve spillet er der hovedsageligt en ting der skal ske - krops-objekterne skal bevæge sig. De skal bevæge sig hen til positionen af den kropsdel foran deres nuværende position, se figur 3. Hovedet skal bevæge sig i en af tre retninger, som ikke er ind mod sig selv.



Figur 3 - slange bevægelse visualiseret.

For at nemmest muligt at opnå denne funktionalitet, lavede jeg klassen `'Body'`.
Body klassen fungerer lidt som en linked-liste, da den kender hhv. næste og forrige element i slangen. Det gør det super nemt at opdatere positionen, da jeg bare kan sætte det til den foran.

Body
+ player: Player + previous: Body + idx: int + x: int + y: int + next: Body
+ <<constructor>> __init__(x:int,y:int, prev:Body/Head,next:bool,player:Player) + debug() -> str + update() + render(screen, pg:pygame)

Figur 4 - Klassediagram af Body klassen.

```
105  def update(self):
106      if not self.alive:return
107      prev = self.tail
108      if self.hasEaten:
109          newTail = Body(prev.x,prev.y,prev,False,self)
110          prev.next = newTail;self.tail = newTail
111          newTail.update();self.hasEaten = False
112      while prev:prev.update();prev = prev.previous
```

```
51  def update(self):
52          self.x = self.previous.x;self.y = self.previous.y
```

Figur 5 - Implementering af opdatering af spillerens krops positioner.

Player klassen har en variabel `'Player.tail'`, som er sat til den bagerste kropsdel i slangen. Det gør det nemt at iterere igennem alle slangens kropsdele bagfra med en mens-løkke.
Den kører mens den midlertidige variabel `'prev'` er sat. Prev variablen bliver None, når man kommer til hovedet og derfor breaker løkken. Prev bliver for hver kropsdel sat til den tidligere kropsdel i slangen efter, positionen på den nuværende er opdateret.

Hovedets bevægelse

Hovedet er interessant, da det bliver sat til den nuværende retning af spilleren. Men spilleren må ikke bevæge sig ind i sig selv, for at undgå dette har jeg to variable hhv. `'new_movement'` og `'last_movement'`. De er sat til hhv. den sidste og nye bevægelse repræsenteret af en koordinat og en om den bevæger sig positivt eller negativt f.eks. `'x-'`.

```
12  #   Input map
13  input_movement = {K_w:'y-',K_s:'y+',K_d:'x+',K_a:'x-'}
```

```
150 if event.key in input_movement.keys():
151     new_movement = input_movement[event.key]
152     if not re.sub('[-+]','',last_movement) in new_movement:
153         send_data(new_movement)
154         pass
155     else:
156         new_movement = last_movement
```

Figur 6 - Input verificering.

Som vist på figur 6, så hvis sidste bevægelse er på samme koordinat/akse, som den nye, så sætter den den nye bevægelse tilbage til at være det samme som sidste bevægelse.
På opdaterings-ticken bliver sidste bevægelse opdateret til værdien af den nye bevægelse (som set på figur 7)

```
91  def update_logic():
92      global players, last_movement, new_movement, apple
93      for tag in players.keys():
94          player = players[tag]
95          player['lastmovement'] = player['newmovement']
96          last_movement = new_movement
97          player['player'].direction = player['newmovement']
98          player['player'].update()
99          if apple == player['player'].head:
100             player['player'].eat()
101             apple.new_position()
102             send_data(f'apple:{apple.x},{apple.y}')
```

Figur 7 - opdaterings logik.

Grundet pladsmangel, vil jeg hoppe videre til server-netværk implementeringen.

Server-klient implementering

Server implementering

For at snakke mellem serveren og spillet skal jeg først lave en socket server, det gør jeg med socket bibliotekets indbyggede class `'socket.socket(args)'`.
Derefter laver jeg en ny tråd til håndtering af hver klient, så jeg kan modtage informationer parallelt.

```python
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind((IP, PORT))
server_socket.listen(5)

while True:
    client_socket, addr = server_socket.accept()
    print(f'[*] Accepted connection from {addr[0]}:{addr[1]}')

    # Add the new client socket to the list
    client_sockets.append(client_socket)
    client_socket.send(f'you|P{client_sockets.index(client_socket)}'.encode('utf-8'))
    if len(client_sockets) == 2:
        time.sleep(0.5)
        client_socket.send('start|info'.encode('utf-8'))
        send_to_all('start|info'.encode('utf-8'), client_socket)
        update_handler = threading.Thread(target=update_loop, args=())
        update_handler.start()
    client_handler = threading.Thread(target=handle_client, args=(client_socket,))
    client_handler.start()
```

Figur 8 - Implementering af socket_server

Alt data skal indkodes som bytes, så det kan sendes over socket-strømmen.
Det gør jeg med Pythons indbyggede `'.encode(type)'` funktion.
Når to klienter har oprettet forbindelse, så sender serveren hver 10. sekund en opdatering til klienterne. Dette er for at sikre at de opdaterer på samme tidspunkt.

```python
try:
    while True:
        data = client_socket.recv(1024)
        if not data:
            break

        # Process the received data
        processed_data = f'P{client_sockets.index(client_socket)}|{data.decode("utf-8")}'
        sendable_data = processed_data.encode('utf-8')
        # Send the processed data back to the client(s)
        client_socket.send(sendable_data)

        # Send data to all connected clients
        print(f'Player {client_sockets.index(client_socket)}: {processed_data}')
        send_to_all(sendable_data, client_socket)

except Exception as e:
    print(f'Error: {e}')

finally:
    # Remove the client socket from the list when done
    client_sockets.remove(client_socket)
    client_socket.close()
```

Figur 9 - håndtering af modtaget data fra klienter.

I håndteringen tjekker serveren hele tiden om den har modtaget data, hvis ikke den har en stabil forbindelse til klienten, så fjerner den klienten fra listen af klienter, da forbindelsen ikke længere eksisterer. Derudover formatere og sender serveren den modtagne data ud til alle klienter. Dette er i forbindelse med inputs fra klienterne, som skal videreformidles til de andre klienter (den røde pil på figur 1 ift. Løsning 1).

Klient implementering

Ligesom med serveren, har klienten en modtag data funktion som kører parallelt med spil-tråden. Der kører jeg bare dataen igennem en fortolker-funktion, hvis opgave er at kalde de rigtige funktioner ud fra hvad for noget data man modtager se figur 10.

```
45  def parse_data(data):#      Parsing of received data
46      global this, tick, STATE_OF_APPLICATION, apple,render_tick
47      tag, cmd = data.split('|')
48      #   I miss switch case from Java :'(
49      if tag == 'you':
50          this = cmd
51          return
52      if tag == 'start':
53          tick = 0
54          STATE_OF_APPLICATION = 'GAME'
55          return
56      if cmd.startswith('apple'):
57          apple.x, apple.y = [int(coord) for coord in cmd.split(':')[1].split(',')]
58          return
59      if tag == 'update':
60          update_logic()
61          return
62      players[tag]['newmovement'] = data
```

Figur 10 - Fortolker-funktion

Noget af det vigtigste på klient-siden er at når jeg lukker applikationen, så skal den også dræbe tråden, som modtager data fra serveren, så serveren er klar over at den har mistet forbindelsen det sker i `'quit_application()'` metoden, hvor jeg kalder `'socket.close()'`, for at afslutte forbindelsen med serveren.

```
37  def quit_application():#    Close socket connection when the application is quit
38      global client_socket
39      client_socket.close()
```

Figur 11 - Afslut program

Test, videreudvikling og konklusion

Med alle de forskellige elementer er der mange ting, som kan gå galt. Det primære problem, når man arbejder med netværksapplikationer, er selvfølgelig synkronisering og stabil kommunikation. Update løkken i server-side er en funktion, som jeg har tilføjet senere hen, da jeg oplevede, at takket være de forskellige computer-specs på hhv. min bærbar og stationær resulterede i, at klienterne ville blive desynkroniseret. Efter flere test har jeg fundet ud af, at det stadig kan ske, men det sker færre gange. F.eks. hvis man trækker i applikationen på windows, så fryser applikationen, hvilket forårsager at man modtager mange update/move pakker på en gang, og man som konsekvens bliver desynkroniseret. Det kunne jeg evt. komme rundt om med et afsluttende tegn på hver packet, så jeg kunne iterere igennem de forsinkede pakker og eksekvere det manglende. En anden løsning kunne være at stoppe programmet når en klient fryser, det har selvfølgelig den ulempe, at det ville gøre det træls at spille for den klient, som ikke fryser. Derudover kunne det være interessant at eksperimentere med løsning 2 ift. at lade spillet køre server-side. Men det er videreudvikling, som der desværre ikke var tid til.
For at opsummere på de krav jeg stillede mig selv i starten af forløbet, kan jeg sige at:
Jeg lever op til 4-5 ud af de 7 krav. Da jeg ikke personligt vil mene, at jeg sikrer en synkronisering på tværs af klienter, men jeg har mulige løsninger til de to krav (krav I og II). Det at spillet ikke er synkront 100% af tiden gør mig også tilbøjelig til at sige at 1v1 spiltypen ikke var 100% funktionel (Krav 4).

Bilag