En kort introduktion

till spelprogrammering

med *Python* och *pygame*

Håkan Lundberg, Cybergymnasiet
hakan.lundberg@nacka.cybergymnasiet.se

Innehåll

Inledning

Att göra enkla datorspel kan vara kul. Man får saker att röra sig på skärmen och man kan tävla mot kompisar med de spel man gjort. I detta häfte kan du läsa om hur man skapar enkla spel med *Python* (www.python.org) och *pygame* (www.pygame.org).

OBS: Bara för att spelen är enkla betyder det inte att de är enkla att göra. Det behövs en hel del övning och uthållighet för att lyckas även med enkla spel!

Vad är *Python* och *pygame*?

Python är ett skript- och programmeringsspråk med klar och enkel syntax som bl.a. passar bra när man börjar med programmering. För att lära dig grunderna i *Python* se t.ex. www.kodknackaren.se eller http://wiki.math.se/wikis/dd100n_0701/index.php/Huvudsida. *Python* skapades av holländaren Guido van Rossum i slutet på 80-talet och är idag ett omtyckt språk av många programmerare.

pygame är ett bibliotek av programkod som underlättar när man vill skapa interaktiv grafik, vilket för det mesta är spel i vårt fall. I början på 00-talet fick den rutinerade C-programmeraren Pete Shinners upp ögonen för *Python* och *SDL*. *SDL* står för *Simple Direct Layer* och är ett kodbibliotek skrivet i C avsett för att göra spel i 2d. Shinners blev förtjust i både *Python* och *SDL* och fick idén att kombinera de båda och skapade då *pygame* som är en slags ”wrapper” runt *SDL* skriven i *Python*.

Som slogan för *pygame* användes “*Takes the C out of game programming*”. C är ett utmärkt språk för spelprogrammering, men det är rätt komplicerat och tar lång tid att lära sig jämför med t.ex. *Python*. Med *pygame* kan man relativt snabbt skapa spel som tack vare kopplingen till *SDL* har bra prestanda på alla plattformar. Både *Python* och *pygame* är ”open source” och har aktiva ”communities”.

Som du kommer att se är ”snygg grafik” sällan något som de som gör spel i *pygame* lägger särskild stor vikt vid. Men *pygame* passar utmärkt för att lära sig grunderna i spelprogrammering och det går att göra även snygga spel om man har talang för digitalt skapande eller kan samarbeta med någon som har det.

Installation och lämplig IDE

pygame finns både för Python version 2.6 och Python version 3.1. De flesta tutorials och spel i *pygame* är gjorda med Python 2.6, men fler och fler använder Python 3.1 (t.ex. boken www.inventwithpython.org).

Här tre länkar om du vill använda *pygame* och Python 2.6:

Python 2.6 – http://www.filehippo.com/download_python/4744/ - klicka på "Download This Version 13.52MB" i högra hörnet
pygame 1.9 – http://www.pygame.org/download.shtml - välj där “pygame-1.9.1.win32-py2.6.msi” om du har Windows

En trevlig IDE (Integrated Development Enviroment) för Python är WingIDE - http://archaeopteryx.com/downloads under "Wing IDE 101"

OBS: När man jobbar med *pygame* i Wing IDE är det en fördel att starta program genom att klicka på *Debug*-knappen, då är det lätt att avsluta dem genom att klicka på *Stop*-knappen.

Speciella ord

Några ord som förklaras utförligt senare i häftet men som används på några ställen även i början:

- **Objekt** – har variabler som håller data och metoder för att utföra saker
  Ett objekt representerar ofta något konkret, t.ex. ett fönster eller ett spelobjekt.
- **Metod** – som en funktion, fast den finns i ett objekt
  Man anropar en metod genom att skriva *objektnamn.metodnamn* – t.ex. *spelare1.hoppa()*.
  En funktion anropas genom att bara skriva namnet på funktionen t.ex. *len(lista)*, eller genom att skriva *modulnamn.funktionsnamn* t.ex. *random.randint(1, 10)*.
- **Klass** – en slags mall som används för att skapa objekt
- **Tuplett** – en fast lista, d.v.s. en lista vars innehåll man inte kan ändra
  För att skapa en tuplett används vanliga parenteser, t.ex. *vit = (255,255,255)*. För att skapa en lista används hakparenteser, t.ex. *lista = [3, 4, 8, 9]*. För att komma åt ett element används hakparenteser både för tupletter och listor, t.ex. *b = vit[2]* och *forsta = lista[0]*.

De engelska orden är: *object, method, class, tuple*

Koordinatsystem

I datorvärlden så är origo (0, 0) högst upp till vänster. Vill man att ett spelobjekt ska röra sig neråt måste man alltså öka dess y-värde och inte minska det som man skulle ha gjort om det vore ett vanligt koordinatsystem.

```
spelare1.y += 10        # Variabeln y ökas med 10, alltså flyttas spelare1 nedåt
spelare2.y -= 10        # Variabeln y minskas med 10, alltså flyttas spelare2 uppåt
```

Skapa ett fönster med bakgrundsfärg

Nedan följer ett litet program i *Python/pygame* som skapar ett fönster med en bakgrundsfärg. Läs koden och kommentarerna noga. Testa gärna att köra programmet samt att ändra bakgrundsfärgen.

```
import pygame           # Importerar pygame-modulen
pygame.init()           # Initierar pygame-modulen
w = 640                 # Bredden på fönstret
h = 480                 # Höjden på fönstret

# Skapar ett objekt som representerar fönstret
screen = pygame.display.set_mode((w, h))  # Bredden och höjden anges som en tuplett

# Skapar en yta (ett yt-objekt av klassen Surface) som är lika stor som fönstret
bakgrund = pygame.Surface((w,h))

# Fyller bakgrundsytan med en röd färg
bakgrund.fill((255,0,0))        # Färgen anges som en tuplett med rgb-värden (red, green, blue)

# "Blittar" / Kopierar bakgrundsytan till screen-objektet
screen.blit(bakgrund, (0,0)) # Bakgrunden läggs på screen-objektet på positionen (0,0)

 # "Flippar displayen" / Uppdaterar hårdvaran för fönstret så att screen-objektet visas på skärmen
pygame.display.flip()
```

raw_input() *# Väntar på input. Används för att fönstret inte ska stängas direkt.*

”Double buffering”

När man jobbar med rörlig grafik gäller det att på ett smart sätt använda datorns resurser så att grafiken blir så bra och ”jämn” som möjligt. För att uppnå det använder alla spelbibliotek något som kallas för *”double buffering”*. Det innebär att man jobbar med en ”dubbelbuffer” som finns i datorns minne och som är en slags spegling av det som syns på skärmen. När man i sitt program ritar och flyttar olika figurer och objekt så gör man det först på ”buffern” som finns i minnet eftersom det går mycket fortare att uppdatera minne än att uppdatera grafikhårdvara.

En annan anledning till att använda ”dubbelbuffering” är att man inte vill att användaren ska se utritningarna av de olika spelobjekten var för sig, utan man vill visa de nya positionerna för spelobjekten först när alla är färdigritade. Alltså, när uppdateringen av ”dubbelbuffern” i minnet är helt klar så anropar man kod som gör så att innehållet i ”dubbelbuffern” visas på skärmen.

I *pygame*-program används ofta variabeln *screen* för att representera fönstret, eller om man ska vara noga ”dubbelbuffern” för *pygame*-fönstret.

screen = pygame.display.set_mode((bredd, hojd)) *# Skapar ett objekt som representerar fönstret*

Funktionen *pygame.display.set_mode* skapar ett speciellt objekt som representerar *pygame*-fönstret på skärmen och som fungerar som ”dubbelbuffer”.

För att göra ”dubbelbuffern” synlig på skärmen anropar man funktionen *pygame.display.flip*. Det är alltså först vid anropet av *flip* som det som finns på *screen*-objektet kommer att visas på skärmen.

pygame.display.flip() # Gör så att det som finns på *screen*-objektet visas på skärmen.

”Blitta en yta och flippa displayen”

Genom att använda metoden *blit* kan man modifiera ”dubbelbuffern”, d.v.s. *screen*. Metoden *blit* används för att kopiera en yta till en annan, t.ex. en bakgrundsyta till *screen*-ytan.

yta1.blit(yta2, pos) # En metod för att kopiera en yta till en annan yta
Metoden *blit* tar två parametrar:
- *yta2* är den yta som kommer att kopieras till *yta1*
- *pos* är en *tuplett* som anger var *yta2*:s översta vänstra hörn ska placeras på *yta1*

Ordet *blit* är en akronym för *”block image transfer”*.

Här ett exempel som skapar en 50x50-yta och ”blittar”/kopierar den två gånger till *screen*-objektet på olika positioner:

```
import pygame                          # Importerat pygame-modulen
pygame.init()                          # Initierar pygame-modulen
w = 640                                # Bredden på fönstret
h = 480                                # Höjden på fönstret
screen = pygame.display.set_mode((w, h)) # Skapar ett objekt som representerar fönstret
bakgrund = pygame.Surface((w,h))       # Skapar en yta som är lika stor som fönstret
bakgrund.fill((255,0,0))               # Fyller bakgrundsytan med en röd färg
box = pygame.Surface((50,50))          # Skapar en yta som är 50 x 50
box.fill((0,255,0))                    # Fyller ytan med en grön färg
screen.blit(bakgrund, (0,0))           # Kopierar bakgrundsytan till screen-objektet
```

```
screen.blit(box, (0,0))          # Kopierar box-ytan till screen-objektet
screen.blit(box, (200,200))      # Kopierar box-ytan till screen-objektet

pygame.display.flip()    # Uppdaterar hårdvaran för fönstret så att screen-objektet blir synligt
raw_input()              # Väntar på input så att inte fönstret stängs direkt
```

Det går även bra att blitta/kopiera box-ytan på bakgrundsytan och sen blitta/kopiera den till *screen*-objektet. Det ger samma resultat i detta fall. Genom att blitta/kopiera till bakgrunden gör man den kopierade ytan till en del av bakgrunden.

```
bakgrund.blit(box, (0,0))        # Kopierar en yta till bakgrundsytan
bakgrund.blit(box, (200,200))    # Kopierar en yta till bakgrundsytan
screen.blit(bakgrund, (0,0))     # Kopierar bakgrundsytan till screen-objektet
```

Sätta en pixel

En skärm byggs upp av små punkter som kallas pixlar. I *pygame* sätter man en pixel genom att använda metoden *set_at*, t.ex. *screen.set_at(pos, color)* där positionen och färgen anges med tupletter. Se exemplet nedan som sätter ut tre pixlar på olika positioner.

```
screen = pygame.display.set_mode((200, 200))    # Skapar ett screen-objekt (en fönsteryta)
screen.set_at((10,10), (255,0,0))               # Sätter en röd pixel på punkten (10,10)
screen.set_at((20,20), (0,255,0))               # Sätter en grön pixel på punkten (20,20)
screen.set_at((30,30), (0,0,255))               # Sätter en blå pixel på punkten (30,30)
pygame.display.flip()                           # Visar screen-objektet på skärmen
```

Med metoden *get_at((x,y))* kan man ta reda på färgen för den pixel som finns på positionen (x,y), den returnerar en tuplett med tre värden (r, g, b). Ett exempel på hur metoden *get_at* kan användas:

```
if screen.get_at(x, y) != (0, 0, 0):    # Om pixeln på (x, y) inte är svart
```

Ytor är viktiga

Som du kanske förstått så är ytor viktiga i *pygame*. Man skapar en yta genom att göra ett objekt av klassen *Surface*. Som parameter skickar man med en tuplett som anger storleken på ytan. För att fylla en yta anropar man metoden *fill*, t.ex. *yta.fill((r,g,b))* där *(r,g,b)* är en tuplett som anger färgen.

En annan vanlig metod för ytor är *convert* som konverterar en yta till ett pixel-format som passar *pygame* (och *SDL*).

```
ex_yta = pygame.Surface((30,30)) # Skapar en 30x30-yta
ex_yta.convert() # Konverterar ytan till ett lämpligt pixel-format
```

Om man vill kan man skapa och konvertera formatet på en rad:

```
ex_yta = pygame.Surface((30,30)).convert() # Skapar en 30x30-yta och konverterar dess format
```

För att göra en del av en yta genomskinlig använder man metoden *set_colorkey((r,g,b))*.

```
yta = pygame.Surface((20, 20)).convert()          # Skapar en 20x20-yta
yta.fill((130, 130, 30))                          # En färg (vilken som hels) …
yta.set_colorkey((130, 130, 30))                  # som här görs genomskinlig
pygame.draw.circle(yta, (255, 0, 0), (10, 10), 10)    # Ritar en röd cirkel med radien 10 på ytan
# Man kommer alltså att se den röda cirkeln men inte hela 20x20-ytan.
```

Ofta behöver man kopiera en yta till en annan, det gör man med (som vi såg ovan) metoden *blit*.

screen.blit(bakgrund, (0,0)) *# Kopierar en bakgrundsyta till screen-objektet*

Man kan skapa en yta utifrån en bild med funktionen *pygame.image.load("filnamn.png")*.

bildyta = pygame.image.load("min_bild.png") *# Skapar en yta från en bild*

Ytterligare ett sätt att skapa en yta är med metoden *render*, se nedan under "Visa text".

Även funktionen *set_mode* skapar en yta (ett yt-objekt), men den ytan är speciell eftersom den representerar fönsterytan genom att vara en s.k. dubbelbuffer för *pygame*-fönstret.

screen = pygame.display.set_mode((w,h)) *# Skapar ett objekt (en yta) som representerar fönstret*
…
pygame.display.flip() # Gör så att det som finns på *screen*-objektet visas på skärmen.

Både funktionen *set_mode* och funktionen *flip* finns i modulen *display* som just är till för att hantera "displayen", d.v.s. i det här sammanhanget *pygame*-fönstret där grafik, spelobjekt m.m. visas.

Rita figurer

Med hjälp av funktioner i modulen *pygame.draw* kan man rita olika figurer på en yta, t.ex. *screen*-objektet eller bakgrundsytan. Här följer några exempel:

 # Ritar en röd linje från punkten (5,100) till punkten (100,100) på bakgrundsytan
pygame.draw.line(bakgrund, (255,0,0), (5,100), (100,100))

 # Ritar en grön 100x100-kvadrat med vänstra översta hörnet på (20, 5)
 # Sista parametern anger tjockleken på linjerna, om 0 så blir figuren fylld
pygame.draw.rect(bakgrund, (0,255,0), (20, 5, 100, 100), 0)

 # Ritar en blå cirkel med centrum (300,300), radien 200 och linjetjockleken 5
pygame.draw.circle(bakgrund, (0,0,255), (300,300), 200, 5)

Se fler figur-exempel i filen *drawDemo.py*.

Visa text

För att skriva ut text i *pygame* krävs tre steg.

Steg 1: Skapa ett font-objekt genom att använda t.ex. klassen *SysFont*
minFont = pygame.font.SysFont("Comic Sans MS", 20) *# Fontnamn och textstorlek som parametrar*

Steg 2: Med hjälp av font-objektet och dess *render*-metod kan man göra text till ett yt-objekt
minTextyta = minFont.render("Ngn text", 1, (255,0,0)) *# 1 anger att "anti-aliasing" ska användas*

Steg 3: Blitta/Kopiera textytan till t.ex. *screen*-objektet
screen.blit(minTextyta, (200,200))

Ordet "*rendera*" betyder ungefär "skapa en bild/yta".
I stället för "*anti-aliasing*" kan man använda ordet "kantutjämning" på svenska.

Spela upp ljud

Även för enkla spel betyder ljudeffekter en hel del. I *pygame* finns stöd för att spela upp ljud i formaten *wav* och *ogg*.

Steg 1: Initiera mixer-modulen
pygame.mixer.init()

Steg 2: Skapa ett ljudobjekt av klassen *Sound* med en ljudfil som parameter
ljudTest = pygame.mixer.Sound("ljud.wav")

Steg 3: Spela upp ljudet med metoden *play*
ljudTest.play() *# Spelar upp ljudet en gång*

Genom att ange en parameter till *play* kan man spela upp ljudet flera gånger:
ljudTest.play(4) *# Spelar upp ljudet 4 gånger*
ljudTest.play(-1) *# Spelar upp ljudet igen och igen*
…
ljudTest.stop() *# Stoppar ljuduppspelningen*

Se programmet *soundTest.py* för ett exempel på ljuduppspelning.

På sidan http://www.grsites.com/archive/sounds/ finns olika ljud att ladda ner gratis.
På http://remix.kwed.org/ låtar som passar i datorspel.

Spelloopen

Precis som "double buffering" så är även spelloopen något som är gemensamt för i stort sett alla spel, stora som små, oberoende av plattform och utvecklingsmiljö/verktyg/språk.

I pseudo-kod kan man beskriva en generell struktur för spel på följande sätt:

Initiera moduler och grafik
Skapa variabler, objekt mm som behövs i spelloopen
Starta spelloopen:
  Kontrollera tiden (så att spelet inte går för fort)
  Ta hand om händelser (från t.ex. mus och tangentbord)
  Uppdatera spelobjekt (dess positioner, riktningar mm)
  Rita spelobjekten ("ritar" i dubbelbuffer)
  Kolla om "game over" (i så fall ska spelloopen avbrytas)
  Uppdatera spelfönstret (visar spelobjekten på sina nya positioner på skärmen)
 # Slut på spelloopen
Visar slutpoäng mm
Avslutar programmet

Idén är alltså att man har en loop som fortsätter så länge spelet är aktivt. För varje varv i loopen tar man hand om eventuella händelser från mus, tangentbord, joystick eller liknande; uppdaterar positionerna m.m. för de olika spelobjekten; kollar om spelet ska avslutas (i så fall ska spelloopen avbrytas); ritar spelobjekten och uppdaterar skärmen.

Att få ytor att röra sig

För att få spelobjekt, t.ex. ytor, att röra sig på skärmen kan man göra på följande sätt:

1. Skapa en bakgrundsyta och en annan yta, t.ex. en box-yta, som ska röra sig
2. Starta en spelloop
    a. Ändra positionen för box-ytan

b. Blitta/Kopiera bakgrundsytan till *screen*-objektet (allt som tidigare fanns på *screen*-objektet täcks över och blir osynligt)
c. Blitta/Kopiera box-ytan till *screen*-objektet på dess nya position
d. ”Flippa displayen” (gör *screen*-objektet, d.v.s. dubbelbuffern synligt på skärmen)

På det sättet skapas en illusion av rörelse, alltså det ser ut som om box-ytan rör sig på skärmen. Men egentligen så flyttar man bara dess position, täcker över tidigare utritning och ritar ut box-ytan igen – i en hastighet som gör att ögat uppfattar det som en rörelse på skärmen.

Här ett program som ”flyttar” en box-yta på skärmen:

```
import pygame
pygame.init()
screen = pygame.display.set_mode((640, 480))

# Skapar en gul bakgrundsyta
bakgrund = pygame.Surface(screen.get_size())
bakgrund = background.convert()
bakgrund.fill((255, 255, 0))

box = pygame.Surface((25, 25))          # Skapar en 25x25-yta
box = box.convert()                     # Konverterar till ett lämpligt pixel-format
box.fill((255, 0, 0))                   # Sätter färgen till röd
box_x = 0; box_y = 200                  # Sätter startpositionen för boxen

clock = pygame.time.Clock()             # Skapar ett clock-objekt (mer om det nedan)
while True:
  clock.tick(30)                        # Sätter maxhastigheten till 30 ”frames per second”

  box_x += 5                            # Ökar variabeln som lagrar boxens x-värde
  if box_x > screen.get_width():        # Om x-värdet är utanför fönstrets högra sida,
    box_x = 0                           # sätt x-värdet till 0, d.v.s. positionen på vänstra sidan

  # Blittar bakgrunden till screen-objektet, dvs ”täcker över” tidigare blittningar/utritningar
  screen.blit(bakgrund, (0, 0))         # Prova att ta bort denna rad!
  screen.blit(box, (box_x, box_y))      # Blittar boxen på dess nya position
  pygame.display.flip()                 # Visar screen-objektet på skärmen
```

”Sätta frame-rejten”

För att spelet (spelloopen) ska gå ungefär lika fort oberoende av hur (o)snabb dator man har sätter man en maxhastighet för spelloopen. Hastigheten kallas för ”frame rate” och mäts i ”*frames per second*” (fps). Det handlar alltså om hur många gånger loopen ska utföras på en sekund, eller med andra ord hur många bildramar (frames) man ska visa under en sekund. För enkla spel kan det räcka med att ha 30 fps, men de stora och avancerade spelen har ofta en ramhastighet på över 100 fps.

I *pygame* sätter man ramhastigheten med hjälp av ett *clock*-objekt och metoden *tick*. Mer exakt så anger *tick*-metoden en maxhastighet för spelet (spelloopen).

```
…
clock = pygame.time.Clock() # Skapar ett clock-objekt
while True: # Spelloopen startar
  # Sätter en övre gräns för ramhastigheten.
```

```
   clock.tick(30) # Om det går fortare än 30 fps så väntar tick-metoden en liten stund
…
```

Testa gärna att använda olika ramhastigheter i dina *pygame*-program.

Händelsehantering

För att få reda på vad som har hänt sen sist, d.v.s. vad användaren gjort under det förra varvet i spelloopen, kan man anropa funktionen *get* i modulen *pygame.event*. Då får man en lista av *event*-objekt som innehåller information om de händelser som inträffat, t.ex. att musen rört sig eller att en tangent trycks ned. Har det inte hänt något så returnerar *get* en tom lista. Studera och testa följande kod:

```
…
clock = pygame.time.Clock()                # Skapar ett clock-objekt
loopa = True                               # Så länge 'loopa' är True ska spelloopen fortsätta
while loopa:
   clock.tick(30)                          # Om det går fortare än 30 fps så väntar tick-metoden
   for event in pygame.event.get():        # Loopar igenom listan av event-objekt
      print event                          # Skriver ut info om ett händelse objekt
      if event.type == pygame.QUIT:        # Om användaren klickat på x i fönstret,
         loopa = False                     # i så fall så ska spelloopen avslutas
      if event.type == pygame.KEYDOWN:     # Om användaren tryckt på en tangent
         if event.key == pygame.K_a:       # Om tangenten är 'a'
            print "a"                      # I så fall skriver vi ut 'a' i konsollen
   pygame.display.flip()                   # "Flippar displayen" / Uppdaterar hårdvaran för fönstret
# Slut på spelloopen
```

Man kan även kolla om en tangent har släppts upp med hjälp av *if event.type == pygame.KEYUP:* och sen använda *event.key* för att ta reda på vilken tangent det gäller (på samma sätt som i koden ovan).

Att styra en kvadrat

Tidigare har vi sett hur man får en box-yta att röra på sig över skärmen. Just har vi sett hur man känner av tangenttryckningar. Nu ska vi kombinera dessa saker för att göra ett program där användaren kan styra en kvadrat med piltangenterna.

För att hålla reda på i vilken riktning som kvadraten ska röra sig inför vi två variabler: *box_dx* och *box_dy*. I *pygame*-program är det vanligt att man använder *dx* och *dy* för att ange förändringen i rörelse för varje varv i spelloopen. *dx* anger förändringen i x-led och *dy* i y-led (d:et står för delta, d.v.s ändring). Man kan även se namnen *vx* och *vy* där v:et står för ”velocity” (hastighet).

```
…                                  # Bakgrunden och box-ytan skapas som tidigare, se ovan
box_x = 0; box_y = 200
box_dx = 0; box_dy = 0

clock = pygame.time.Clock()
loopa = True

while loopa:
   clock.tick(60)
   for event in pygame.event.get():        # Loopar igenom listan av event-objekt
      if event.type == pygame.QUIT:        # Om användaren valt att stänga fönstret
         loopa = False                     # I så fall så ska spelloopen avslutas
```

```
    if event.type == pygame.KEYDOWN:      # Om användaren tryckt på en tangent
        if event.key == pygame.K_UP:      # Upppil-tangenten
            box_dy -= 1                   # Minskar box_dy med ett (Om box_dy < 0 går boxen uppåt)
        if event.key == pygame.K_DOWN:    # Nerpil-tangenten
            box_dy += 1                   # Ökar box_dy med ett (Om box_dy > 0 går boxen nedåt)
        if event.key == pygame.K_LEFT:    # Vänsterpil-tangenten
            box_dx -= 1                   # Minskar box_dx med ett (Om box_dx < 0 går boxen till vänster)
        if event.key == pygame.K_RIGHT:   # Högerpil-tangenten
            box_dx += 1                   # Ökar box_dx med ett (Om box_dx > 0 går boxen till höger)

    box_x += box_dx                       # Ändrar x-positionen med box_dx
    box_y += box_dy                       # Ändrar y-positionen med box_dy

    screen.blit(background, (0, 0))       # Täcker över tidigare utritningar
    screen.blit(box, (box_x, box_y))      # Ritar ut boxen på dess nya position
    pygame.display.flip()                 # Visar screen-objektet på skärmen
```

Kanthantering

Det finns olika sätt att ta hand om spelobjekt som befinner sig vid en ut av fönsterkanterna. Man kan:

- låta spelobjektet **försvinna** från fönstret (och från spelet) – genom att inte göra någonting
- låta spelobjektet **vara kvar** vid kanten – genom att sätta hastigheten till noll för spelobjektet
- låta spelobjektet **studsa** – genom att ändra riktningen på hastigheten för objektet
- låta spelobjektet komma in från **andra sidan** – genom att ändra positionen på spelobjektet från att ha varit vid ena kanten till att vara vid den motstående kanten
- flytta på bakgrunden, s.k. ***scrolling*** så att man får känslan av att röra sig i en stor ”värld”

Se gärna koden i mappen ”kanthantering” och testa den för att få en mer konkret förståelse.

Objektorienterad programmering - Klasser och objekt

Ett sätt att programmera spel (och även andra typer av program) är med det som kallas för **objektorienterad programmering** - vilket ofta förkortas OOP. Det bygger på att programkoden delas upp i delar som kallas för klasser. Utifrån de klasserna skapar man objekt som kan interagera med varandra genom att anropa varandras metoder.

En klass är som en slags mall. Ett objekt är en konkretisering av en klass och representerar ofta något som finns i verkligheten. Om vi t.ex. har en klass/mall som heter *Person* så kan vi utifrån den skapa objekt som representerar konkreta personer med olika namn, adress, längd, ålder mm.

En klass består av *attribut* och *metoder*. Attribut är variabler som håller information om ett objekt som skapas utifrån klassen. Metoder är som funktioner som beskriver vad man kan göra med objektet.

Det bästa sättet att komma in i det objektorienterade tänket är att titta på exempel och sen att själv försöka skriva klasser och skapa objekt. Nedan ett exempel på en klass med två attribut:

```
class Person:
    def __init__(self, namn, epost):      # Denna metod anropas när ett objekt skapas
        self.namn = namn                  # self.namn är ett attribut / instansvariabel
        self.epostadress = epost          # self.epostadress är ett attribut / instansvariabel
# ---------- Slut på klassen Person ----------
```

Den metod som anropas när ett objekt skapas av en klass kallas för *konstruktor* eller *konstruerare*. I *Python* är det metoden *__init__(self)* som fungerar som konstruktor.

Variabeln *self* refererar till det aktuella objektet av klassen. (Man behöver inte använda just namnet *self* för den variabel som refererar till det aktuella objektet, men det är tradition att göra så i *Python*. I *Java* och *C#* heter motsvarande variabel *this*.) Variabeln *self* används inom en klassdeklaration, men inte utanför (där använder man istället variabelnamnet för objektet).

Man skapar objekt genom att skriva ett variabelnamn, tilldelningstecken (=), namnet på klassen, en vänsterparentes, eventuella parametrar och en högerparentes.

```
person1 = Person("Ana Pana", "ana@lernu.net")        # Skapar ett objekt av klassen Person
person2 = Person("Zam", "zam@lernu.net")             # Skapar ett till objekt av klassen Person
```

För att komma åt ett attribut i ett objekt använder man så kallad punktnotation:

```
person1.namn              # objektnamn.attributnamn
```

I stället för ordet *objekt* kan man använda ordet *instans*. Istället för attribut kan man säga *instansvariabler* som alltså betyder att de är variabler i ett objekt. Instansvariabler har unika värden för varje objekt av klassen. Som vi såg ovan kommer åt en instansvariabel genom punktnotation *objektnamn.instansvariabel*:

```
print person1.namn, person1.epostadress   # Skriver ut instansvariablerna för objektet person1
print person2.namn, person2.epostadress   # Skriver ut instansvariablerna för objektet person2
```

Det finns även *klassvariabler* som är variabler i en klass som delas av objekten för den klassen, de har alltså samma värde för alla objekt av den klassen.

```
class Bomb:
   farg = (0,0,0)              # Svart färg. Denna variabel delas av alla objekt
   def __init__(self):
      self.x = random.randint(10,100) # Slumpar ett värde för instansvariabeln self.x
      self.y = random.randint(10,100) # Slumpar ett värde för instansvariabeln self.y

   def explodera(tid):
      # kod som visar en explosion på positionen self.x, self.y efter ”tid” sekunder
# ---------- Slut på klassen Bomb ----------

bomb1 = Bomb()              # Skapar ett bomb-objekt
bomb2 = Bomb()              # Skapar ett till bomb-objekt

print Bomb.farg             # Skriver ut klassvariabeln color
print bomb1.x, bomb1.y      # Skriver ut instansvariablerna för objektet bomb1
print bomb2.x, bomb2.y      # Skriver ut instansvariablerna för objektet bomb2

bomb1.explodera(10)         # Anropar metoden explodera för objektet bomb1
```

Principerna för klasser och objekt är de samma även i andra objektorienterade programspråk som t.ex. *C++*, *C#* och *Java*, men själva syntaxen skiljer sig en del i vissa sammanhang.

Grunderna i objektorienterad programmering är egentligen inte speciellt svåra, men det brukar ta en del tid innan alla begreppen sjunker in och man förstår hur man kan skapa och använda klasser och objekt. Här en kort sammanfattning av orden vi just gått igenom:

- **Klass**: som en mall för att skapa objekt
- **Objekt**: har data (attribut/instansvariabler) och metoder
- **Attribut**: håller data för ett objekt
- **Metod**: som en funktion som finns i ett objekt
- **Konstruktor**: den metod som anropas då ett objekt skapas, i *Python* heter den *__init__(self)*
- **Instans**: ett annat ord för objekt
- **Instansvariabel**: ett annat ord för attribut
- **Klassvariabel**: en variabel som har samma värde för alla objekt av den aktuella klassen
- ***self***: en speciell variabel som refererar till det aktuella objektet i en klassdeklaration
- **Punktnotation**: Sättet man kommer åt attribut och metoder i ett objekt.

Det passar bra att lära sig grunderna i OOP i samband med spelprogrammering i och med att de flesta av objekten då visas på skärmen som delar i spelet och de blir på så sätt mer konkreta och lättare att förstå än om man bara jobbar med icke-synliga objekt.

För att ett programspråk ska kunna användas för OOP krävs att språket stödjer två saker, nämligen de som kallas *inkapsling* och *arv*. (Mer om det senare.)

OOP – Exempel på klasser och objekt i *pygame*

Vi har tidigare sett flera exempel på hur man använder en klass för att skapa objekt i *pygame* och även på hur man anropar metoder för de objekten:

```
yta = pygame.Surface((50,50))          # Skapar ett yt-objekt från klassen Surface
yta.fill((0,0,255))                    # Anropar fill-metoden för att fylla ytan med en färg

clock = pygame.time.Clock() # Skapar ett clock-objekt från klassen Clock
clock.tick(30)              # Anropar tick-metoden för att sätta maxhastigheten till 30 fps

screen.set_at((10,10), (255,0,0))      # Anropar set_at-metoden för att sätta en pixel
screen.blit((0,0), yta)     # Anropar blit-metoden för att kopiera en yta till screen-objektet
```

Klassen *Surface* ligger i modulen *pygame*, därför skriver man *pygame.Surface((50,50))*. Klassen *Clock* ligger i modulen *pygame.time*, därför skriver man *pygame.time.Clock()*.

När man programmerar objekt-orienterat använder man ofta redan färdiga klasser (som ligger i olika moduler) för att skapa objekt. Man kan även skriva sina egna klasser och använda dem för att skapa objekt. Det vanligaste är att man gör både och, d.v.s. använder färdiga klasser och skriver sina egna.

OOP - Inkapsling

Med inkapsling menas att man kan ha variabler (som kallas attribut eller instansvariabler) och en slags funktioner (som kallas metoder) avgränsade från andra delar av programkoden. Attributen håller data om ett objekt och metoderna beskriver vad man kan göra med objekten.

```
class Punkt:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def rita(self, yta):
        yta.set_at(self.x, self.y)          # Ritar ut punkten
# ---- Slut på klassen Punkt -----
```

Attributen *x* och *y* och metoden *rita* sägs vara inkapslade.

OOP - Arv

När man skapar en ny klass är det ibland smidigt att utgå ifrån en redan befintlig klass och återanvända kod ifrån den. Det kan man göra genom att låta en klass ärva från en annan klass och på det sättet få alla dess attribut och metoder på köpet. Här följer ett väldigt enkelt exempel för att visa hur man ärver från en klass i *Python*:

```
class Punkt3D (Punkt):                  # Ärver från klassen Punkt
   def __init__(self, x, y, z):
      Punkt.__init__(self, x, y)        # Anropar basklassens konstruktor
      self.z = z
# ---- Slut på klassen Punkt3D -----

p = Punkt3D(2, 4, 6)                    # Skapar ett objekt av klassen Punkt3D
print p.x, p.y, p.z                     # Skriver ut instansvariablerna för objektet p
```

Den klass man ärver ifrån kallas för *basklass* eller *superklass*. Den klass som ärver från en annan klass kallas för *subklass*.

(Arv används även för att skapa hierarkier av klasser. Men det ska vi inte gå in på här.)

OOP – Relationer mellan objekt

För att ett objekt ska kunna använda metoder i ett annat objekt krävs att det finns någon slags koppling/relation mellan objekten. Den vanligaste relationen mellan objekt brukar kallas för *”känner till”*. I det fallet har ett objekt ett attribut som refererar till ett annat objekt. T.ex. kan det se ut på följande sätt:

```
class Skott:
   # Attribut och metoder för att visa ett skott (typ kanonkula) på skärmen
   …
# ---- Slut på klassen Skott -----

class Rymdskepp:

   def __init__(self, skott):     # En referens till ett skott-objekt skickas med till konstruktorn
      self.skott = skott          # Och sparas i en instansvariabel
      # En massa andra attribut

   def skjut(self):
      self.skott.x = self.x       # Gör skottet synligt
      self.skott.y = self.y
      self.skott.hastighet = 10 # Ger skottet en hastighet
      self.skott.vinkel = self.vinkel          # Och en vinkel

   # En massa andra metoder

# ---- Slut på klassen Rymdskepp-----

skott = Skott()                  # Skapar ett skott-objekt
skepp = Rymndskepp(skott)  # Skickar med en referens till skott-objektet till konstruktorn
```

Det var ett enkelt exempel på en *”känner till”*-relation, d.v.s. skepp-objektet känner till skott-objektet och kan därför använda det.

Arv mellan två objekt kan man kalla för en *”är en”*-relation. T.ex. om man han en klass *Person* och sen skapar en klass *Elev* som ärver från klassen *Person* så betyder det att *Elev* ”är en” *Person*.

(Man kan skriva tjocka böcker om OOP. Här har vi gått igenom det mest grundläggande inom OOP så att vi kan jobba med så kallade ”*sprites*”, vilka underlättar arbetet med att skapa spel i *pygame* en hel del.)

Spelobjekt (*Sprites*)

I de flesta spelbibliotek finns stöd för att visa och förflytta spelobjekt på skärmen. Ett spelobjekt kallas av tradition för en *sprite*. Definitionen för en *sprite* varierar beroende på programspråk och spelbibliotek, men ofta menar man en 2d-bild som har en viss position och storlek som man kan flytta, rotera, skala mm. I många spelbibliotek finns en speciell *sprite*-klass som man kan låta klasser för spelobjekt ärva ifrån.

I *pygame* finns klassen *Sprite*, i modulen *pygame.sprite*, som man ärver ifrån när man vill skapa klasser för spelobjekt. Följande attribut är speciellt viktiga för sådana *sprite*-objekt:

- ***self.image*** - en yta som representerar spelobjektet grafiskt
- ***self.rect*** - en rektangel som ligger runt ytan
  OBS: Genom att ändra positionen för *self.rect* flyttar man spriten/spelobjektet på skärmen.

Här följer några exempel på hur man kan använda attributet *self.rect* i ett spelobjekt:

```
self.rect.centerx += 5          # Flyttar spelobjektet 5 pixlar i x-led
self.rect.centerx, self.rect.centery = (40, 200)  # Sätter spelobjektet på positionen (40, 200)
if self.rect.top > 0: …         # Kollar om spelobjektet är på väg ut vid övre kanten
```

Följande attribut i *rect* är användbara och bra att känna till:

top, left, bottom, right
topleft, bottomleft, topright, bottomright
midtop, midleft, midbottom, midright
center, centerx, centery
size, width, height
w,h

Mer info om *rect* finns på sidan http://www.pygame.org/docs/ref/rect.html.

OBS: För att kunna flytta en yta med hjälp av *rect*-attributen krävs det att ytan och rect:en är attribut i ett *sprite*-objekt, d.v.s. så som det är beskrivet i denna del.

Alla spelobjekt/sprites är kvadratiska i *pygame*, även de som inte har en kvadratisk form. Det måste man vara uppmärksam på när man vill kontrollera om två spelobjekt har kolliderat. Dock finns det en metod i rect som heter *inflate(…)* med vilken man kan minska kollisions-rektangeln.

Följande metoder är extra viktiga och obligatoriska för *sprite*-objekt:

- ***__init__(self)*** – konstruktor som anropas när spelobjektet skapas. I den anger man vilka attribut som ska finnas, bl.a. *self.image* och *self.rect*, och ger dem värden. Första raden i konstruktorn måste vara ett anrop till *__init__*-metoden för basklassen, nämligen

pygame.sprite.Sprite.

- ***update(self)*** – här finns själva logiken för spelobjektet, t.ex. att uppdatera dess position och känna av om det är på väg utanför skärmen. Denna metod anropas för det mesta en gång för varje varv i spelloopen.

Här följer ett exempel på hur en klass för ett spelobjekt kan se ut:

```
class Boll(pygame.sprite.Sprite):
   def __init__(self, screen):
      pygame.sprite.Sprite.__init__(self)                 # Anropar konstruktron för Sprite-klassen
      self.screen = screen
      self.image = pygame.Surface((16, 16)).convert()     # Skapar en 16x16-yta
      self.image.fill((0, 0, 0))                          # Samma färg som bakgrunden
      pygame.draw.circle(self.image, (0,255,255), (8,8), 8) # Ritar en cirkel på 16x16-ytan
      self.rect = self.image.get_rect()
      self.starta()

   def starta(self):
      self.rect.center = (320, 240)             # Start-positionen för bollen
      self.dx = random.choice([-2,-1,1,2])      # Slumpar en hastighet i x-led
      self.dy = random.choice([-2,-1,1,2])      # Slumpar en hastighet i y-led

   def update(self):
      self.rect.centerx += self.dx
      self.rect.centery += self.dy
      if self.rect.bottom >= self.screen.get_height():          # Nedre kanten
         self.dy *= -1              # Byter riktning på bollen i y-led
      if self.rect.top  <= 0:                                   # Övre kanten
         self.dy *= -1              # Byter riktning på bollen i y-led

# ----------------- Slut på klassen Boll -------------------
```

För att få ett spelobjekt av den klassen, d.v.s. att skapa en instans av klassen *Boll*, skriver man:

```
boll = Boll()    # Ett anrop till metoden __init__ görs och objektet skapas
```

Som vanligt kan man komma åt attribut och metoder i objektet genom så kallad punktnotation.

```
if boll.rect.right > screen.get_width():        # Om bollen nått den högra sidan
   boll.dx = boll.dx * -1          # Byter riktning på bollen i x-led
```

För att *sprite*-objekt verkligen ska komma till sin rätta i *pygame* måste man lägga dem i en *sprite*-grupp (se nedan). Men man kan använda *sprite*-objekt även utan *sprite*-grupper, i spelloopen har man i så fall följande:

```
   …
   screen.blit(bakgrund, (0,0))              # Täcker över tidigare utritningar
   boll.update()                             # Uppdaterar bollens position
   screen.blit(boll.image, boll.rect.top)    # Ritar ut bollen igen på screen-ytan/objektet
   pygame.display.flip()                     # Visar screen-objektet på skärmen
```

Sprite-grupper

För att underlätta hanteringen av *sprite*-objekt använder man s.k. *sprite*-grupper. Det går till så att man först skapar sina *sprite*-objekt, t.ex. *obj1* och *obj2* och sen skriver:

grupp = pygame.sprite.Group(obj1, obj2) *# Skapar ett sprite-grupp-objekt med två sprite-objekt*

I spelloopen måste man sen anropa tre metoder i *grupp*-objektet för att dess *sprite*-objekt ska visas på skärmen:

while loopa:

…
grupp.clear(screen, backgrund) *# Ritar över spelobjekten från förra varvet med bakgrundsfärgen*
grupp.update() *# Gör så att update()-metoderna för de sprite-objekt som finns i gruppen anropas*
grupp.draw(screen) *# Ritar ut spelobjekten, d.v.s. ”blittar” dess image-ytor på screen-objektet*
pygame.display.flip() # Gör dubbelbuffern, d.v.s. screen-objektet, synligt på skärmen

Det som är mest intressant för oss är metoden *grupp.update()* för den anropar de *update*-metoder vi själva skrivit i de klasser som ärver från *Sprite*-klassen. I *update*-metoderna flyttar vi positionen för ett spelobjekt så att det när det sen ritas ut med hjälp av *grupp.draw(screen)* hamnar på en ny position. På så sätt skapas en illusion av rörelse för alla *sprite*-objekt. För varje varv görs alltså bl.a. följande med spelobjekten: *suddar bort - clear, beräknar nya positioner - update, ritar ut igen - draw.*

En stor fördel med *sprite*-objekt och *sprite*-grupper är att kollisionshanteringen blir enkel!

Kollisionshantering

I många spel är själva syftet att det ska ske (eller inte ske) kollisioner mellan spelobjekt. Använder man *sprite*-objekt och *sprite*-grupper har man stor hjälp av de inbyggda metoderna för kollisionshantering:

- ***obj1.rect.colliderect(obj2.rect)*** – För att undersöka om två spelobjekt kolliderat.

- ***pygame.sprite.spritecollide(obj, grupp, kill)*** – För att undersöka om ett *sprite*-objekt kolliderat med något ut av *sprite*-objekten i den angivna gruppen. Parametern *kill* har värdet *True* eller *False*. Om den är satt till *True* så tas ev. kolliderade objekt bort från gruppen.

- ***pygame.sprite.groupcollide(grupp1, grupp2, kill1, kill2)*** - För att undersöka om något av *sprite*-objekten i två olika grupper kolliderat.

Man kan så klart även skriva sina egna metoder eller funktioner för att utföra kollisionshantering, men man spar en hel del tid om man kan använda de inbyggda metoderna (beskrivna ovan).

Mer om händelsehantering

De vanligaste sätten att ta hand om händelser är att:

1. ta hand om alla händelser i spelloopen med en *for*-loop som går igenom listan av *event*-objekt och sen ha en massa *if*-satser som kollar vilken händelse som inträffat;

2. bara ta hand om QUIT-händelsen i spelloopen och låta *update*-metoden i *sprite*-objekten ta hand om övriga händelser.

För att ta hand om tangenthändelser kan man antingen använda *pygame.event.get()* för att få en lista av *event*-objekt och sen kolla om *event.type == pygame.KEYDOWN* (eller KEYUP) och använda

event.key för att ta redan på vilken tangent det gäller. Ett annat sätt är att använda funktionen *pygame.key.get_pressed()* som returnerar en tuplett med *True* för tangenter som tryckts ner och *False* för de andra.

```
keys = pygame.key.get_pressed()          # Ger en tuplett med True/False för varje tangent
if keys[pygame.K_LEFT]:                  # Kollar om vänster-pil-tangent har värdet True
   self.dx -= 1                          # I så fall betyder det att den är nedtryckt
```

När det gäller att ta redan på koordinaterna för muspekaren är det funktionen *pygame.mouse.get_pos()* som gäller, den returnerar en tuplett med x och y för muspekaren.

x, y = pygame.mouse.get_pos() *# Ger x och y för muspekarens position*

Vanliga moduler och klasser i *pygame*

Här följer korta beskrivningar av några vanliga moduler och klasser i *pygame*.

Moduler:

- *pygame.display* – Har funktioner för att hantera displayen (d.v.s. *pygame*-fönstret).
  - Ex: *set_mode((w, h))* och *flip()*
- *pygame.draw* – Har funktioner för att rita figurer på en yta.
  - Ex: *line(…), circle(…), rect(…)*
- *pygame.event* – Har funktioner för att ta hand om händelser.
  - Ex: *get(), pump()*
- *pygame.font* – Har bl.a. klasser för att skapa fontobjekt.
  - Ex: *SysFont(…), Font(….)*
- *pygame.image* – Har funktioner för att öppna och spara bildfiler.
  - Ex: *load(filnamn), save(yta, filnamn)*
- *pygame.key* – Har funktioner och konstanter för att behandla tangent-händelser
  - Ex: *get_pressed(), K_UP, K_a*
- *pygame.mouse* – Har funktioner för att hantera musen.
  - Ex: *get_pos(), set_visible(True)*
- *pygame.sprite* – Har klasser och funktioner för att hantera spelobjekt.
  - Ex: *Sprite, spritecollide(…), groupcollide(…)*
- *pygame.transform* – Har funktioner för att bl.a. rotera och skala ytor.
  - Ex: *rotate(…), scale(…)*

Klasser:

- *pygame.Surface* – För att skapa ytor, d.v.s. yt-objekt.
- *pygame.sprite.Sprite* – Ärver man ifrån när man skriver en klass för spelobjekt.
  - Extra viktiga attribut: *self.image* och *self.rect*
  - Extra viktiga metoder: *__init__(self)* och *update(self)*
- *pygame.sprite.Group* – För att skapa grupper för spelobjekt.
  - Viktiga metoder: *clear(screen, bakgrund), update(), draw(screen)*
- *pygame.Rect* – För att skapa rect-objekt.
  - Viktiga attribut: *top, left, bottom, right, center, centerx, centery mfl*

  - Viktiga metoder: *collliderect(…), move(…), inflate(…)*
- *pygame.time.Clock* – För att kunna sätta ramhastigheten med metoden *tick(fps)*.

Användbara moduler och klasser (som inte ingår i *pygame*)

Om ingen länk ges nedan till modulen eller klassen så finns den i katalogen ”användbara”.

Label – En *sprite*-klass för att ha en text som uppdateras varje varv i spelloopen.

Shell – En klass för att skjuta och visa ett skott.

dumbmenu – En modul för att göra en enkel startmeny, t.ex. ”Starta med 1 spelare”, ”Starta med 2 spelare”, ”Förklaringar”, ”Sluta”. Se http://www.pygame.org/project-dumbmenu-1425-.html.
(Det finns även andra meny-moduler/klasser att välja på, se http://www.pygame.org/tags/menu.)

Ett spelexempel - *Pong*

Här följer koden för ett enkelt spel. Läsa, testa o ändra gärna i koden (då lär man sig en hel del).

```
import pygame, random, sys
pygame.init()

class Boll(pygame.sprite.Sprite): # Används för att skapa en pong-boll

    def __init__(self, screen):                 # Konstruktorn som anropas när ett objekt skapas
        pygame.sprite.Sprite.__init__(self)     # Anropas konstruktorn för Sprite-klassen
        self.screen = screen                    # En referens till screen-objektet för att veta dess storlek
        self.image = pygame.Surface((16, 16)).convert()                 # En 16x16-yta
        self.image.fill((0, 0, 0))              # Fyller den med svart så att den blir osynlig
        pygame.draw.circle(self.image, (0, 255, 255), (8, 8), 8)        # Ritar en vit cirkel på ytan
        self.rect = self.image.get_rect()       # En rektangel runt bollen
        self.starta()                           # Sätter start posision och hasatighet för bollen

    def starta(self):                           # Anropas när bollen ska sättas i spel
        self.rect.center = (self.screen.get_width()/2, self.screen.get_height()/2) # Börjar på mitten
        self.dx = random.choice([-2,-1,1,2])    # Slumpar en hastighet i x-led
        self.dy = random.choice([-2,-1,1,2])    # Slumpar en hastighet i y-led

    def update(self):                           # Anropas en gång för varje varv i spelloopen
        self.rect.centerx += self.dx            # Flyttar bollens posision i x-led
        self.rect.centery += self.dy            # Flyttar bollens posision i x-led
        if self.rect.bottom >= self.screen.get_height():        # Om bollen gått i taket
            self.dy *= -1                       # Ändra y:s riktning
        if self.rect.top  <= 0:                 # Om bollen gått i golvet
            self.dy *= -1                       # Ändra y:s riktning

# ---------- Slut på klassen Boll ----------
```

```
# En klass för att skapa spelare, dvs "paddlar"
class Spelare(pygame.sprite.Sprite):

    # Klassvariabler som delas mellan objekt av denna klass
    STILL = 0
    UPP = 1
    NER = 2

    def __init__(self, screen, startx, starty):    # Anropas när objekt skapas av klassen Spelare
        pygame.sprite.Sprite.__init__(self)
        self.screen = screen
        self.image = pygame.Surface((5, 40)).convert()
        self.image.fill((0, 255, 0))
        self.rect = self.image.get_rect()
        self.rect.center = (startx, starty)
        self.poang = 0
        self.hastighet = 3
        self.riktning = Spelare.STILL

    def update(self):                              # Anropas en gång för varja varv i spelloopen
        if self.riktning == Spelare.UPP:
            self.rect.centery -= self.hastighet
        elif self.riktning == Spelare.NER:
            self.rect.centery += self.hastighet

#------- Slut på klassen Spelare -------

# En klass för att skriva ut text, i vårt vall poängen
class Label(pygame.sprite.Sprite):

    def __init__(self, x, y):                      # Anropas när objekt skapas av klassen Label
        pygame.sprite.Sprite.__init__(self)
        self.font = pygame.font.SysFont("None", 30)
        self.text = "0 - 0"                        # Ändras i spelloopen när ngn får poäng
        self.center = (x, y)

    def update(self):                              # Anropas en gång för varja varv i spelloopen
        self.image = self.font.render(self.text, 1, (0, 0, 233))
        self.rect = self.image.get_rect()
        self.rect.center = self.center

#------- Slut på klassen Label --------

# Här börjar själva huvudprogrammet
w, h = 640, 480                                    # Storleken på fönstret
screen = pygame.display.set_mode((w, h))   # Skapar ett screen-objekt som representerar skärmen
pygame.display.set_caption("Ett enkelt Pong-spel")          # Sätter en titel i fönsterramen
```

```
bakgrund = pygame.Surface(screen.get_size()).convert()  # Skapar en bakgrundsyta
bakgrund.fill((0, 0, 0))                                # Fyller den med svart färg
screen.blit(bakgrund, (0, 0))          # Kopierar den svarta bakgrunden till screen-ytan

AVSTAND = 30                           # Avståndet mellan en "paddel" och väggen bakom
spelare_v = Spelare(screen, AVSTAND, h/2)               # Skapar spelaren till vänster
spelare_h = Spelare(screen, w - AVSTAND, h/2)           # Skapar spelaren till höger
boll = Boll(screen)                                     # Skapar ett boll-objekt
poang = Label(w/2, 20)                 # Skapar ett label-objekt för att skriva ut poängen

allaSpelobjekt = pygame.sprite.Group(boll, spelare_v, spelare_h, poang)  # Spelobjekten i en grupp

to_late = False # Sätts till True om bollen gått forbi en spelare (vill ända se bollen gå in i väggen)

clock = pygame.time.Clock()  # Skapar ett clock-objekt

while True:                            # Startar spelloopen
  clock.tick(100)                      # Sätter maxhastigheten (kan behöva ändras beroende på dator)
  for event in pygame.event.get():     # Hämtar o loopar igenom en lista av händelser
    if event.type == pygame.QUIT:      # Om man klickat på krysset i fönstret
      sys.exit(0)                      # Avslutar programmet
    if event.type == pygame.KEYDOWN:   # Om ngn tangent tryckts ned
      if event.key == pygame.K_a:
        spelare_v.riktning = Spelare.UPP   # Sätter vänsterspelarens riktning till UPP
      elif event.key == pygame.K_z:
        spelare_v.riktning = Spelare.NER   # Sätter vänsterspelarens riktning till NER
      if event.key == pygame.K_k:
        spelare_h.riktning = Spelare.UPP   # Sätter högerspelarens riktning till UPP
      elif event.key == pygame.K_m:
        spelare_h.riktning = Spelare.NER   # Sätter högerspelarens riktning till NER
    if event.type == pygame.KEYUP:     # Om ngn tanget släpps upp
      if event.key == pygame.K_a or event.key == pygame.K_z:
        spelare_v.riktning = Spelare.STILL  # Sätter vänsterspelarens riktning till STILL
      if event.key == pygame.K_k or event.key == pygame.K_m:
        spelare_h.riktning = Spelare.STILL  # Sätter högerspelarens riktning till STILL
  #------- Slut på for-event-loopen -------

  # Kolla studs mot en paddel mm
  if not to_late:                      # Om bollen inte redan gått förbi en paddel
    if boll.rect.colliderect(spelare_v.rect) or boll.rect.colliderect(spelare_h.rect):
      boll.dx *= -1                    # Om bollen träffat en paddel så byt riktningen i x-led
    if boll.rect.right > screen.get_width() - AVSTAND: # Om bollen gått förbi högerspelaren
      spelare_v.poang += 1             # Poäng till vänsterspelaren
      poang.text = str(spelare_v.poang) + " - " + str(spelare_h.poang) # Ny text att skriva ut
      to_late = True                   # Nu kan man inte längre "fånga" bollen
    elif boll.rect.left < AVSTAND:     # Om bollen gått förbi vänsterspelaren
      spelare_h.poang += 1
```

```
        poang.text = str(spelare_v.poang) + " - " + str(spelare_h.poang)
        to_late = True
    else:                                   # Bollen har gått föbi en paddel
      if boll.rect.left > screen.get_width() or boll.rect.right < 0:
        boll.starta()                       # Starta om när bollen gått in i väggen
        to_late = False         # Ny omång, alltså är det möjligt att "fånga" bollen igen

    # Suddar bort, beräknar nya positioner och ritar ut igen
    allaSpelobjekt.clear(screen, bakgrund)  # Ritar över spelobjekten med bakgrunden
    allaSpelobjekt.update()                 # Uppdaterar positionerna för spelobjekten
    allaSpelobjekt.draw(screen)             # Blittar alla spelobjekt på screen-objektet
    pygame.display.flip()                   # Visar screen-objektet på skärmen

#------- Slut på spelloopen -------
```

Att göra animeringar

I stället för att ha en statisk bild eller yta som man t.ex. kan röra med piltangenterna vill man ibland ha en animerad bild som man kan styra. Det kan man ordna genom att ha ett sprite-objekt med flera olika bilder som visas efter varandra i snabb följd. Man kan även ha olika typer av bilder beroende på vad spelobjektet är i för tillstånd, t.ex. om det ”går” använder man vissa bilder och för ”hopp” har man andra bilder.

Att programmera ett animerat spelobjekt är en sak, att hitta eller göra passande bilder för animeringen är en helt annan sak. Om man själv inte är kapabel att skapa sådana bilder kan man gå till t.ex. sidan http://www.flyingyogi.com/fun/spritelib.html där det finns en hel del att hämta.

Se filerna *cow….py* för exempel på hur man kan skriva programkod för att skapa animeringar.

Att rotera spelobjekt

Ibland vill man bara använda en bild som man ser ovan ifrån och kunna rotera den beroende på i vilken riktning spelobjektet rör sig. I *pygame* finns en modul *transform* som bl.a. har en funktion för att rotera en bild. När man roterar en bild flera gånger så tappar den i kvalité. Det är därför viktigt att ha en ”master-bild” som man alltid utgår ifrån när man roterar. En annan sak att tänka på är att bildens centrum kan förflyttas vid en rotation. Så för att behålla samma centrum får man först spara undan det i en variabel som man sen använder efter rotationen. T.ex. kan det se ut på följande sätt:

```
oldCenter = self.rect.center
self.image = pygame.transform.rotate(self.imageMaster, self.dir)
self.rect = self.image.get_rect()
self.rect.center = oldCenter
```

Rektangeln som ligger runt bilden kan bli större vid en rotation, se koden i *drawBounds.py*.

I filen *move8dir.py* finns ett bra exempel med kommentarer på ett roterbart spelobjekt.

Att scrolla en bakgrund

Genom att scrolla bakgrunden får man en känsla av att röra sig i en stor spelvärld. Den effekten kan man få genom att ha en stor bakgrundsbild som an flyttar lite på för varje varv i spelloopen. När man kommer till slutet av bakgrundsbilden låter man den börja från början igen.

Om scrollningen ska ske i x-led ska bakgrundsbilden vara längre än fönstret, t.ex. tre gånger så lång. Om rörelsen ska vara åt höger låter man den del av bakgrundsbilden som inte syns ligga till vänster om fönstret som utgångsläge. Sen flyttar man fram bakgrundsbilden några pixlar åt gången åt höger. När positionen för vänsterkanten på bakgrundsbilden är på väg att bli 0 så sätter man tillbaka bilden i utgångsläget igen och börjar om. Se klassen *ScrollX* i filen *scrolltest.py*.

På liknande sätt gör man om man vill att bakgrunden ska scrolla i y-led. Då gäller det att bakgrundsbilden är betydligt högre än fönstret och att man för varje varv i spelloopen flyttar bakgrundsbilden i y-led. Se klassen *ScrollY* i filen *scrolltest.py*.

Att ha olika speltillstånd

Ett spel består ofta av olika tillstånd, t.ex. startsidan med en meny, själva spelet, ett paus-tillstånd samt en slutsida som presenterar en bästa-poäng-lista. De olika tillstånden kan man dela upp i funktioner i sin programkod. För att göra en enkel meny kan man t.ex. använda modulen *dumbmenu* (se ovan).

Källor

www.pygame.org - Där finns beskrivningar av moduler, klasser, metoder, funktioner m.m. i *pygame*.

Game Programming av Andy Harris - http://aharrisbooks.net/pythonGame/