

# Dag 4

...

Starten på KivyBird!

# Denne gangen skal vi starte på flappy bird appen!

Husk å legge denne koden i bunnen av python filen!

*main.py*

Vi begynner først med bakgrunnen, og får den animert.

```
if __name__ == "__main__":  
    KivyBirdApp().run()
```

Deretter begynner vi på resten av spritesa, så ser vi hvor langt vi kommer!

En generell konvensjon er å legge alle *import* statements i toppen av filen, for lesbarhetens skyld - så husk det mens vi går gjennom koden.

# Grunnleggende Widgeten

Dette er den grunnleggende widgeten.

Vi snakket forrige gang om hvordan klasser kan arve fra hverandre.

Derfor lager vi nå en grunnleggende klasse, som gir alle andre muligheten til å hente ut teksturen vår.

Vi gjør også at teksturen kan kalles fra andre ting som en attributt kalt `tx_*` - i bakgrunnens tilfelle ville det være `tx_background`

*main.py*

```
+from kivy.core.image import Image  
+from kivy.uix.widget import Widget
```

```
+class BaseWidget(Widget):  
+    def load_tileable(self, name):  
+        t = Image('images/{}.png'.format(name)).texture  
+        t.wrap = 'repeat'  
+        setattr(self, 'tx_{}'.format(name), t)
```

# Bakgrunnen

Dette er klassen for alt som har med bakgrunnen å gjøre.

Først skaper vi klassen, og gjør klart en *ObjectProperty* - en objektegenskap - som skal inneholde den faktiske *teksturen*

Videre lager vi en hjelpemetode *set\_background\_size* som hjelper oss med å sette teksturen til riktig størrelse for skjermen.

Vi lager også metoden *on\_size* for å endre størrelsen hver gang skjermen endrer seg.

*main.py*

```
+from kivy.properties import ObjectProperty
+class Background(BaseWidget):
+    tx_background = ObjectProperty(None)
+    def __init__(self, **kwargs):
+        super(Background, self).__init__(**kwargs)
+        self.load_tileable('background')
+    def set_background_size(self, tx):
+        tx.uvsize = (self.width / tx.width, -1)
+    def on_size(self, *args):
+        self.set_background_size(self.tx_background)
```

# App klassen

I app klassen skriver vi vår egen *on\_start* metode som setter bakgrunnen, og sier at den skal oppdatere seg selv ca 60 ganger i sekundet.

*self.root.ids.background* refererer her til kv  
filen - som vi skal lage om 2 strakser.

*main.py*

```
+from kivy.app import App
```

```
+from kivy.clock import Clock
```

```
+class KivyBirdApp(App):
```

```
+    def on_start(self):
```

```
+        self.background = self.root.ids.background
```

```
+        Clock.schedule_interval(self.update, 1.0/60.0)
```

```
+    def update(self, nap):
```

```
+        self.background.update(nap)
```

# KV filen!

Her setter vi et *FloatLayout* som rot-widgeten.

Dette betyr at det første som blir satt på skjermen er dette layoutet, og alt som kommer sammen med det.

*Background* har vi definert i *main.py* og her instansierer vi den og gir den en id.

*canvas* er lærrettet i bakgrunnen - noe vi kan tegne på og gi teksturer. Her, *tx\_background*

*kivybird.kv*

```
+FloatLayout:
```

```
+    Background:
```

```
+        id: background
```

```
+    canvas:
```

```
+        Rectangle:
```

```
+            pos: self.pos
```

```
+            size: (self.width, self.height)
```

```
+            texture: self.tx_background
```

# Update metoden!

Her kommer *update* metoden, som vi kaller 60 ganger i minuttet. Update metoden kaller en ny metode, *set\_backgroun\_uv* med navnet på teksturen, og verdien den skal bruke på den nye posisjonen.

*main.py*

```
class Background(BaseWidget):  
    ...  
+     def update(self, nap):  
+         self.set_background_uv('tx_background', 2 * nap)  
  
+     def set_background_uv(self, name, val):  
+         t = getattr(self, name)  
+         t.uvpos = ((t.uvpos[0] + val) % self.width, t.uvpos[1])  
+         self.property(name).dispatch(self)
```

# Resten av importsa!

For enkelthetens skyld importerer vi nå bare resten av importsa vi skal bruke i dag:

Når du er ferdig skal det se slik ut:

*main.py*

```
from kivy.app import App
```

```
from kivy.clock import Clock
```

```
from kivy.core.image import Image
```

```
from kivy.uix.widget import Widget
```

```
-from kivy.properties import ObjectProperty
```

```
+from kivy.properties import ObjectProperty, ListProperty, NumericProperty
```

```
+from kivy.properties import AliasProperty
```

```
+from kivy.core.window import Window, Keyboard
```

```
+from kivy.uix.image import Image as ImageWidget
```

```
+import random
```



# Pipes! - i kv filen

*kivybird.kv*

```
<Pipe>:
```

```
    canvas:
```

```
        Rectangle:
```

```
            pos: (self.x + 4, self.FLOOR)
```

```
            size: (56, self.lower_len)
```

```
            texture: self.tx_pipe
```

```
            tex_coords: self.lower_coords
```

```
        Rectangle:
```

```
            pos: (self.x, self.FLOOR + self.lower_len)
```

```
            size: (64, self.PTOP_HEIGHT)
```

```
            texture: self.tx_ptop
```

```
        Rectangle:
```

```
            pos: (self.x + 4, self.upper_y)
```

```
            size: (56, self.upper_len)
```

```
            texture: self.tx_pipe
```

```
            tex_coords: self.upper_coords
```

```
        Rectangle:
```

```
            pos: (self.x, self.upper_y - self.
```

```
PTOP_HEIGHT)
```

```
            size: (64, self.PTOP_HEIGHT)
```

```
            texture: self.tx_ptop
```

```
            size_hint: (None, 1)
```

```
            width: 64
```

# Pipes! Attributter

FLOOR her er “bakkenivå” for pipesa  
PTOP\_HEIGHT er størrelsen på “toppen”  
av pipesa  
PIPE\_GAP er mellomrommet mellom  
pipesa.

*main.py*

```
+class Pipe(BaseWidget):  
+    FLOOR = 96  
+    PTOP_HEIGHT = 26  
+    PIPE_GAP = 150  
+    tx_pipe = ObjectProperty(None)  
+    tx_ptop = ObjectProperty(None)  
+    ratio = NumericProperty(0.5)  
+    lower_len = NumericProperty(0)  
+    lower_coords = ListProperty((0, 0, 1, 0, 1, 1, 0, 1))  
+    upper_len = NumericProperty(0)  
+    upper_coords = ListProperty((0, 0, 1, 0, 1, 1, 0, 1))  
+    upper_y = AliasProperty(  
+        lambda self: self.height - self.upper_len,  
+        None, bind=['height', 'upper_len'])
```

# Pipes! - teksturene

Her legger vi til samme startmetoden vi bruker i bakgrunnen, for å laste inn teksturen vår for pipen og pipe toppen.

*main.py*

```
class Pipe(BaseWidget):
```

```
...
```

```
+ def __init__(self, **kwargs):
```

```
+     super(Pipe, self).__init__(**kwargs)
```

```
+     for name in ('pipe', 'ptop'):
```

```
+         self.load_tileable(name)
```

# Pipes - Koordinater (Vanskelig tema!)

Her setter vi koordinatene til pipesa.  
Dette er et ganske komplisert tema,  
så vi hopper over å gå i detalj her.

Om det ønskes kan jeg forberede en  
ting til neste gang som forklarer  
dette.

*main.py*

```
class Pipe(BaseWidget):  
    ...  
    def set_coords(self, coords, len):  
        len /= 16  
        coords[5:] = (len, 0, len)  
    def on_size(self, *args):  
        pipes_length = self.height - (  
            Pipe.FLOOR + Pipe.PIPES_GAP + 2 * Pipe.PTOP_HEIGHT)  
        self.lower_len = self.ratio * pipes_length  
        self.upper_len = pipes_length - self.lower_len  
        self.set_coords(self.lower_coords, self.lower_len)  
        self.set_coords(self.upper_coords, self.upper_len)  
        self.bind(ratio=self.on_size)
```

# Spawne pipes!

Her legger vi til pipes i hovedklassen

Her lager vi en metode for å lagre pipesa vi henter ut, slik at vi kan fjerne de senere - samtidig som vi henter inn nye.

*main.py*

```
class KivyBirdApp(App):  
+     pipes = []  
  
def on_start(self):  
+     self.spacing = 0.5 * self.root.width  
...  
  
def spawn_pipes(self):  
+     for p in self.pipes:  
+         self.root.remove_widget(p)  
+     self.pipes = []  
+     for i in range(4):  
+         p = Pipe(x=self.root.width + (self.spacing * i))  
+         p.ratio = random.uniform(0.25, 0.75)  
+         self.root.add_widget(p)  
+         self.pipes.append(p)
```

# Bevege på pipes!

Her oppdaterer vi *update* metoden, slik at den også oppdaterer og henter pipes.

*main.py*

```
class KivyBirdApp(App):
```

```
...
```

```
def update(self, nap):
```

```
    self.background.update(nap)
```

```
+     for p in self.pipes:
```

```
+         p.x -= 96 * nap
```

```
+         if p.x <= -64:
```

```
+             p.x += 4 * self.spacing
```

```
+             p.ratio = random.uniform(0.25, 0.75)
```

# “Fuglen”!

Akkurat som bakgrunnen,  
henter vi inn fuglen.

Her skal fuglen kun være på et  
satt punkt i starten.

Siden vi ikke skal *tile* fuglen, henter  
vi den bare som et vanlig bilde.

*kivybird.kv*

```
FloatLayout:
```

```
...
```

```
+ Bird:
```

```
+ id: bird
```

```
+ pos_hint: {'center_x': 0.3333, 'center_y': 0.6}
```

```
+ size: (50, 50)
```

```
+ size_hint: (None, None)
```

```
+ source: 'images/flappynormal.png'
```

# Fuglen!

For å bare hente fuglen inn skal man bare trenge å gjøre dette:

*main.py*

```
+class Bird(ImgWidget):
```

```
+    pass
```



# Brukerinput!

Her binder vi touch inputet og mellomrom til en metode, som vi kaller *on\_key\_down* -

Denne kaller `self.user_action` om man trykket på mellomrom

Det som skjer da er at man starter spillet for nå.

```
class KivyBirdApp(App):  
+   playing = False  
def on_start(self):  
...  
+   window.bind(on_key_down=self.on_key_down)  
+   self.background.on_touch_down = self.user_action  
+   def on_key_down(self, window, key, *args):  
+       if key == Keyboard.keycodes['spacebar']:  
+           self.user_action()  
+       def user_action(self, *args):  
+           if not self.playing:  
+               self.spawn_pipes()  
+               self.playing = True
```

# Å fly er bare å forsinke et fall!

Her legger vi til tyngdekraften på fuglen vår.

Merk at *ACCEL\_FALL* kan endres etter vanskelighetsgrad.

```
class Bird(ImageWidget):  
-    pass  
+    ACCEL_FALL = 0.25  
+    speed = NumericProperty(0)  
+    def gravity_on(self, height):  
+        self.pos_hint.pop('center_y', None)  
+        self.center_y = 0.6 * height  
+    def update(self, nap):  
+        self.speed -= Bird.ACCEL_FALL  
+        self.y += self.speed
```

# Oppdatere hovedklassen

Her legger vi til at fuglen skal begynne å falle når man trykker på skjermen.

Legger også til oppdateringen på fuglen selv.

```
class KivyBirdApp(App):  
    ...  
    def user_action(self, *args):  
        if not self.playing:  
+            self.bird.gravity_on(self.root.height)  
            self.spawn_pipes()  
            self.playing = True  
    def update(self, nap):  
        self.background.update(nap)  
        if not self.playing:  
            return  
+            self.bird.update(nap)
```

# Flyving!

Her legger vi til hvor raskt fuglen skal fly opp gjennom *ACCEL\_JUMP*

og lager metoden vi kaller for å gjøre det!

```
class Bird(ImageWidget):  
    ...  
    + ACCEL_JUMP = 5  
    ...  
    + def bump(self):  
    +     self.speed = Bird.ACCEL_JUMP
```

# Rotere fuglen etter vinkelen! (Vanskelig stoff!)

Dette er en egenskap for å regne ut vinkelen fuglen skal holde relativt til farten, sammen med det som kommer i KV filen

```
class Bird(ImageWidget):  
    ...  
    + angle = AliasProperty(  
    +     lambda self: 5 * self.speed,  
    +     None, bind=['speed'])  
    ...
```

# Rotere fuglen, KV delen

Igjen, dette er vanskelig stoff  
så vi går ikke inn i full detalj her.

```
+<Bird>:  
+   canvas.before:  
+       PushMatrix  
+       Rotate:  
+           angle: root.angle  
+           axis: (0, 0, 1)  
+           origin: root.center  
  
+   canvas.after:  
+       PopMatrix
```

# Kollisjon!

Her sjekker vi om fuglen har nådd taket eller bunnen.

Vi sjekker også om man har kollidert med pipen, gjennom en standard kivy funksjon kalt *collide\_widget*

```
class KivyBirdApp(App):  
    ...  
    + def test_game_over(self):  
    +     screen_height = self.root.height  
    +     if self.bird.y < 90 or self.bird.y > screen_height - 50:  
    +         return True  
    +     for p in self.pipes:  
    +         if not p.collide_widget(self.bird):  
    +             continue  
    +         if (self.bird.y < p.lower_len + 116 or  
    +             self.bird.y > screen_height - (p.upper_len + 75)):  
    +             return True  
    +     return False
```

# Sjekk om game over

Her sjekker vi om game over er True  
i så fall, sett playing til False.

```
def update(self, nap):  
...  
+     if self.test_game_over():  
+         self.playing = False
```