Inhaltsverzeichnis

Aufgabenstellung

Beschreibung


```
# Ein einfaches Spiel mit 3 Elementen
# Das Spielfeld: Hintergrund mit Punkteanzeige
# Der Spieler: animierte Figur, die springen kann
# Die Hindernisse: tauchen unregelmässig auf und müssen übersprungen werden
# Ziel: Möglichst viele Hindernisse nicht berühren
```

Die Elemente

Der Spieler

Der Spieler besteht aus einer animierten Figur, die springen kann.

Die Figur steht immer an derselben Position auf der linken Seite. Durch eine Animation sieht es so aus, als würde sie rennen.

Der Spieler hat aber die Möglichkeit zu springen. Wenn er ganz unten steht, springt er sehr schnell auf eine bestimmte Höhe. Ausgelöst wird das durch einen Druck auf die UP-Taste (Pfeil nach oben). Sobald die Taste losgelassen wird, kehrt die Figur sofort auf den Boden zurück. Etwas Magie ist auch dabei: durch dauerhaftes Drücken der Taste, kann der Flug verlängert werden.

Die Hindernisse

Die Hindernisse bestehen aus unregelmäsig am rechten Rand auftauchenden Kakteen. Sie laufen von rechts nach links durch und dürfen vom Spieler nicht berührt werden. Durch die Animation des Spielers sieht es so aus, als würde der Spieler rennen und eine Kamera würde dafür sorgen, dass er immer im Bild bleibt.

Wenn ein Hindernis den linken Rand erreicht, ohne dass es berührt wurde, verschwindet es und dem Spieler wird ein Punkt gutgeschrieben.

Das Spiel endet, wenn der Spieler ein Hindernis berührt.

Die Spielkontrolle

Der wichtigste Teil ist hier das Spielfeld. Es wird als Fenster auf dem Computerbildschirm dargestellt, der Titel kann durch das Programm bestimmt werden.

Rechts oben wird der aktuelle Punktestand angezeigt.
Links oben werden Statusinformationen ausgegeben.
Das eigentliche Geschehen läuft unten ab, wir sehen einen Spieler und mehrere Hindernisse.

Zusätzlich wird das Spiel noch mit Musik und Geräuschen etwas attraktiver gestaltet.

Installation und Hilfsmaterial

Thonny

Es wird die Entwicklungsumgebung Thonny verwendet.

https://thonny.org

Unter ***Tools*** / ***Options*** / ***Interpreter*** muss die Option
`The same interpreter which runs Thonny (default)`
ausgewählt sein.

Unter ***Tools*** / ***Manage plug-ins...*** muss ***`pgzero`*** installiert werden. Pygame Zero basiert auf Pygame, vereinfacht aber einige Dinge.

Damit steht die notwendige Python - Umgebung zur Verfügung.

Grafikdateien

Im Tutorial https://aposteriori.trinket.io/game-development-with-Pygame-zero ist ein Link auf diverse frei verfügbare Grafikdateien zu finden.

Wir verwenden https://www.aposteriori.com.sg/wp-content/uploads/2020/02/image_pack.zip

Der Spieler besteht aus den Dateien ***run__000.png*** bis ***run__009.png,*** die im Verzeichnis ***ninja*** zu finden sind. Für die Hindernisse wird ***cactus.png*** aus dem Verzeichnis ***items*** verwendet.

Musik und Sound

Weitere Ressourcen findet man auf https://opengameart.org.
Die Musikdatei ***Venus.wav*** findet man in https://opengameart.org/content/nes-shooter-music-5-tracks-3-jingles.

Die Sounddatei ***GameOver.wav*** findet man unter https://opengameart.org/content/game-over-soundold-school.

Diese Dateien müssen umbenannt werden. Sie dürfen keine Grossbuchstaben oder Sonderzeichen enthalten. Wir verwenden daher ***venus.wav*** und ***gameover.wav***.

Zusätzliche Module

Thonny installiert mit pgzero alle notwendigen Module. Wir benötigen aber noch ein zusätzlichen Hilfsmodul: https://www.aposteriori.com.sg/wp-content/uploads/2021/01/pgzhelper.zip

pgzhelper.py muss direkt in das Projektverzeichnis kopiert werden.

Lektion 1 - Vorbereitung und ein Spielfeld

Dateien und diese Anleitung im Github - Repository
https://github.com/hobbyelektroniker/Python_Pygame-Zero

Für die Videos gibt es eine eigene Playlist:
https://www.youtube.com/playlist?list=PL4dxj1rGc3b0jfUgaehIsVxNDQq9W-r2o

Die Dateistruktur

Die Dateistruktur wird von Pygame Zero vorgegeben. Alle Dateien liegen in einem Projektverzeichnis (z. Bsp. ***Game***).

```
Game
  game.py
  pgzhelper.py
  sounds
   gameover.wav
  music
    venus.wav
  images
    cactus.png
    run__00.png
    run__01.png
    run__02.png
    run__03.png
    run__04.png
    run__05.png
    run__06.png
    run__07.png
    run__08.png
    run__09.png
```

Grundstruktur des Programmes

Auch diese ist durch Pygame Zero vorgegeben. Wir nennen unser Hauptprogramm ***game.py***.

```
import pgzrun

# Das Spielfeld
WIDTH=800
HEIGHT=600
TITLE = "Jump and Run"

def update():
    pass

def draw():
    pass

pgzrun.go()
```

Zuerst wird das Modul ***pgzrun*** importiert. Dieses stellt ein Spielfeld zur Verfügung. Wir müssen nur noch sagen, wie gross es sein soll. Dazu wird die ***Breite*** und ***Höhe*** angegeben. Zusätzlich kann noch ein ***Titel*** festgelegt werden.

Die Funktion des Spiels wird durch die Funktion ***update()*** festgelegt. Alle Elemente müssen mit ***draw()*** gezeichnet werden. Diese beiden Funktionen sind im Moment noch leer. Es wird unsere Aufgabe sein, den entsprechenden Code zu schreiben.

Ein Python - Programm beendet sich immer selbst, wenn es am Ende ankommt. Unser Spiel soll aber immer weiterlaufen. Darum ist der letzte Befehl im Spiel immer ***pgzrun.go()***. Dieser Befehl führt eine Endlosschleife aus, die das Spiel am Laufen hält.

Es ist klar, dass dieses Programmgerüst noch nicht viel macht. Es stellt einfach ein schwarzes Fenster in der Grösse 800 x 600 mit dem Titel ***Jump and Run*** dar. Also müssen wir noch einen Schritt weitergehen und wenigstens ein richtiges Spielfeld darstellen.

Das Spielfeld

Ganz am Anfang haben wir gesagt, dass das Spielfeld Bestandteil des Elements Spielkontrolle ist. Wir versuchen also eine Spielkontrolle zu programmieren, die in der Lage ist, das Spielfeld darzustellen. Wir gehen jetzt von der Vorlage aus dem Anhang aus.
Das Spielfeld soll etwa so aussehen:

Damit der Computer dieses Spielfeld aufbauen kann, müssen wir ihm einen Bauplan zur Verfügung stellen.

Das Fenster selbst haben wir ja bereits und der Titel ist auch schon vorhanden. Es bleibt also noch der Inhalt des Fensters.

Solche Baupläne nennt man in der Programmierung ***Klassen***. Die Klasse für unsere Spielkontrolle Klasse nennen wir ganz unbescheiden ***Spiel***.

```
class Spiel:
    def __init__(self):
        self.aktiv = False
        self.finished = False    # finished == beendet (Game Over)
        self.punkte = 0
        # Figuren erzeugen
        self.spieler = None      # Noch kein Spieler vorhanden
        self.hindernisse = []    # Eine leere Liste für die Hindernisse
```

Eine solche Klasse bekommt einen Namen (***Spiel***). Unter ***def __init__(self)*** kann festgelegt werden, welche Eigenschaften die Klasse hat. Mit dem Wort ***self*** lässt sich auf diese Eigenschaften zugreifen.

Die Klasse muss wissen, ob ein Spiel momentan gerade läuft oder ob ein Game Over aufgetreten ist.
```
self.aktiv = False
self.finished = False    # finished == beendet (Game Over)
```

Wir wollen Punkte verwalten und erzeugen daher eine Eigenschaft ***punkte***. Diese wird für den Anfang auf 0 gesetzt.
```
self.punkte = 0
```

Wir haben einen Spieler. Dieser wird in der Eigenschaft ***spieler*** verwaltet. Wir haben ihn noch nicht programmiert, deshalb setzen wir die Eigenschaft auf ***None*** (nicht vorhanden).
```
self.spieler = None
```

Ausserdem kann es mehrere Hindernisse geben. Diese sind ebenfalls noch nicht programmiert. Wir können aber bereits eine leere Liste anlegen, in die später die Hindernisse eingefügt werden.
```
self.hindernisse = []    # Eine leere Liste für die Hindernisse
```

Wenn ein Spiel gestartet wird, werden die Punkte auf 0 gesetzt und die Liste mit den Hindernissen geleert. Dann kann das Spiel auf aktiv gesetzt werden.

```
def start(self):
    self.Punkte = 0
    self.hindernisse.clear()
    self.aktiv = True
    self.finished = False
```

Bei Spielstopp werden die beiden Eigenschaften aktiv und finished entsprechend gesetzt.

```
def stopp(self):
    self.aktiv = False
    self.finished = True
```

Unsere Klasse muss aber noch mehr können. Sie muss zum Beispiel in der Lage sein, das Spielfeld, den Spieler und die Hindernisse zu zeichnen. Wir müssen also eine Methode angeben, die diese Elemente zeichnet. Diese Zeichenfunktion nennen wir ***draw()***. Es ist sehr hilfreich, dass Pygame Zero uns bereits eine Zeichenfläche namens ***screen*** zu Verfügung stellt.

```
def draw(self):
    self.zeichne_spielfeld()
    self.zeichne_spieler()
    self.zeichne_hindernisse()
```

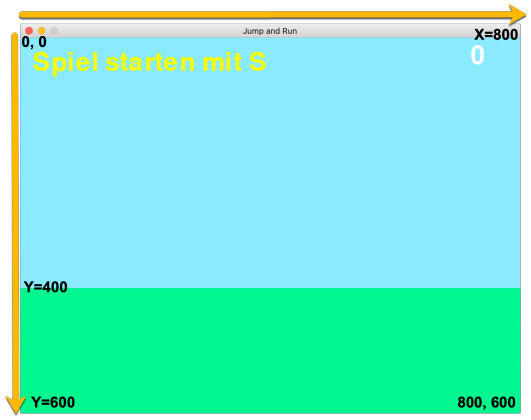
Auch hier haben wir wieder dieses ***self***. Es sorgt dafür, dass wir auf die Eigenschaften und Methoden der Klasse zugreifen können. Hier sagen wir jetzt, dass wir das Spielfeld, den Spieler und die Hindernisse zeichnen möcht. Wir sagen aber noch nicht, wie das geschehen soll. Deshalb brauchen wir drei weitere Methoden.

zeichne_spieler() und ***zeichne_hindernisse()*** geben nicht viel Arbeit. Da wir noch keinen Spieler und keine Hindernisse haben, erzeugen wir leere Methoden. Da eine solche Funktion nie ganz leer sein darf, fügen wir ***pass*** ein. Damit sagen wir, dass hier momentan keine Funktion ausgeführt wird, wir diese aber später nachliefern werden.

```
def zeichne_spieler(self):
    pass

def zeichne_hindernisse(self):
    pass
```

Um das Spielfeld zu zeichnen, müssen wir uns zuerst über das Koordinatensystem klar werden.



Das Spielfeld ist aufgeteilt in einen Himmel und einen Boden. Y-Werte unter 400 liegen im Bereich des Himmels, Werte über 400 liegen auf dem grünen Boden. Damit wir uns das nicht merken müssen, definieren wir eine Konstante ***HIMMEL***.

```
WIDTH=800
HEIGHT=600
TITLE = "Jump and Run"
HIMMEL = 400
```

Mit diesem Wissen können wir den Himmel zeichnen.

```
def zeichne_spielfeld(self):
    farbe_himmel = (163, 232, 254)  # blau
```

```
    screen.draw.filled_rect(Rect(0, 0, WIDTH, HIMMEL), farbe_himmel)
```

Die Farbe des Himmels wird im RGB (rot, grün, blau) Format angegeben. Um die richtigen Farbcodes zu finden, gibt es einige gute Webseiten. Hier ein Beispiel:

https://www.rapidtables.com/web/color/RGB_Color.html

Der Himmel erstreckt sich über ein Rechteck (***Rect***) ausgehend von Punkt (0,0), mit der Breite 800 und der Höhe 400. Wir geben das mit Hilfe der Konstanten ***WIDTH*** und ***HIMMEL*** an.

Dasselbe machen wir mit dem Boden.

```
farbe_boden = (88, 242, 152)  # grün
screen.draw.filled_rect(Rect(0, HIMMEL, WIDTH, WIDTH - HIMMEL), farbe_boden)
```

Die Punkte sollen rechts oben dargestellt werden. Ich platziere sie 80 Pixel vom rechten Rand weg.

```
farbe_schrift = (255, 255, 255)  # weiss
screen.draw.text(str(self.punkte), (WIDTH - 80,  10), fontsize=60, color=farbe_schrift)
```

Jetzt bleibt nur noch die Beschriftung für Game Over und den Spielstart.

```
if self.finished:
  screen.draw.text("GAME OVER", (20, 20), fontsize=80, color=(255, 0, 0))
  screen.draw.text("Neues Spiel mit S", (20, 80), fontsize=60, color=(255, 255, 0))
elif not self.aktiv:
  screen.draw.text("Spiel starten mit S", (20, 20), fontsize=60, color=(255, 255, 0))
```

Für den Moment belassen wir die update() - Methode so wie sie in der Vorlage steht. Später wird sie dann die wichtigsten Teile der Spielelogik enthalten.

```
def update(self):
    # Start-Taste nur abfragen, wenn das Spiel nicht aktiv ist
    if not self.aktiv and keyboard.s:
        # neues Spiel starten
        self.start()

    # Wenn das Spiel nicht aktiv ist, hier abbrechen
    if not self.aktiv: return
```

Damit haben wir einen funktionierenden Bauplan erstellt. Die Spielkontrolle muss nur noch gebaut werden. Das ist ganz einfach:

```
spiel = Spiel()
```

Damit erzeugen wir aus dem Bauplan ***Spiel*** ein Objekt ***spiel***.

Im Hauptprogramm finden wir die Funktionen update() und draw(). Bisher wurde hier mit ***pass*** nichts gemacht. Die Funktion soll aber das Spiel ausführen und zeichnen. Daher rufen wir jetzt spiel.update() und ***spiel.draw()*** auf.

```
def update():
    spiel.update()

def draw():
    spiel.draw()
```

Und schon sieht unser Spielfeld wie erwartet aus.

Lektion 2 - Der Spieler

Was wäre ein Spiel ohne Spieler.

Das ist unser Held. Er läuft und läuft und läuft... und darf dabei kein Hindernis berühren. Damit er das tun kann, müssen wir ihn zeichen, animieren und ihm das Springen beibringen.

Um das Projekt übersichtich zu halten, erstellen wir für den Spieler eine eigene Date: ***spieler.py***
Auch hier erstellen wir eine Klasse. In diesem Fall hat Pygame Zero aber sehr schön vorgearbeitet. Es wird eine Klasse ***Actor*** (Schauspieler) zur Verfügung gestellt, die bereits die wichtigsten Eigenschaften und Fähigkeiten hat. Wir übernehmen das für unsere eigene Klasse. Die Vorlage aus dem Anhang hilft uns dabei.

Die Klasse Spieler

```
class Spieler(Actor):
    # Es muss eine Startposition angegeben werden
    def __init__(self, x, y):
        super().__init__('run__000')
        self.aktiv = False

        # Die Ausgangsposition wird gespeichert
        self.baseX = x
        self.baseY = y
        # Die aktuelle Position
        self.x = x
        self.y = y
```

In einem ersten Schritt stellen wir einfach einen unbewegten Spieler dar. Wir übernehmen alles, was ***Actor*** kann. Das wird durch `class Spieler(Actor):` festgelegt.

Beim Erstellen des Spielers möchten wir angeben, wo der Spieler erscheinen soll. Deshalb geben wir beim Erstellen die x- und y-Koordinate mit: `def __init__(self, x, y):`

Da wir ***Actor*** übernehmen, müssen wir auch die __init__ Methode von Actor aufrufen. Das geschieht mit `super().__init__('run__000')`

super() verweist immer auf die Basisklasse, die in unserem Fall ***Actor*** ist. Actor benötigt ein Bild, wir verwenden ***run__000***. Die dazugehörige Datei ***run__000.png*** haben wir bereits in den Ordner ***images*** kopiert.

Wir speichern die Anfangsposition in ***baseX*** und ***baseY***, die aktuelle Position (***x*** und ***y***) wird auf die Basisposition gesetzt.

```
self.baseX = x
self.baseY = y
self.x = x
self.y = y
```

Den Spieler in das Spiel einbinden

Jetzt erzeugen wir einen Spieler auf unserem Spielfeld. Das geschieht wieder im Hauptprogramm ***game.py***.

Zuerst wird das neue Modul eingebunden.

```
import pgzrun
from spieler import Spieler
```

Das Modul ***spieler*** soll uns also die Klasse ***Spieler*** zur Verfügung stellen.
Das benutzen wir auch gleich in der Klasse ***Spiel***. Bisher war die Eigenschaft ***spieler*** ja auf None gesetzt. Jetzt erzeugen wir stattdessen einen Spieler.

```
self.finished = False   # finished == beendet (Game Over)
self.spieler = Spieler(100, HIMMEL)
self.hindernisse = []   # Eine leere Liste für die Hindernisse
```

Der Spieler wird mit einem Abstand von 100 Pixel vom linken Rand her gezeichnet. Die Mitte des Körpers soll an der Grenze zwischen Boden und Himmel liegen.

Der Spieler muss noch gezeichnet werden. Dafür haben wir bereits die Funktion ***zeichne_spieler()*** vorgesehen. Wir rufen dort einfach ***self.spieler.draw()*** auf. Diese ***draw()*** - Funktion müssen wir nicht selbst programmieren. Sie wird durch ***Actor*** unserem Spieler zur Verfügung gestellt.

```
def zeichne_spieler(self):
    self.spieler.draw()
```

Der Spieler muss in start() noch gestartet werden.

```
def start(self):
    ...
    self.spieler.start()
```

Wenn wir jetzt das Programm ausführen, wird der Spieler dargestellt.
Jetzt möchten wir ihm aber das Springen beibringen.

Der Spieler soll springen

Dazu erstellen wir in der Klasse Spieler die Methode ***sprung(self)***.

```
def sprung(self):
    self.y = self.baseY - 350 # Sprung auf maximale Höhe
```

Mit diesem Befehl springt der Spieler 350 Pixel hoch. Danach muss er aber wieder herunterkommen. Das soll automatisch geschehen. Dazu muss man wissen, dass Spiele in einzelnen Schritte ausgeführt werden. Bei jedem Schritt wird im Hauptprogramm die Funktion ***update()*** aufgerufen. Von dort aus kann die Update - Methode des Spielekontrollers aufgerufen werden. Diese ruft dann die Update-Methode aller beteiligten Elemente auf. Wir müssen also die Update-Methoden der Klasse ***Spiel*** und der Klasse ***Spieler*** schreiben.

Hier in der Klasse ***Spieler***.

```
def update(self):
    # Wenn wir nicht am Boden sind, dann fallen wir
    if self.y < self.baseY:
        self.y += 2
```

Im Hauptprogramm (***game.py***) müssen wir noch dafür sorgen, dass der Sprung ausgelöst und die Update-Methode des Spielers aufgerufen wird.

In der Klasse ***Spiel***:

```
def update(self):
    ...
    if keyboard.up:
        self.spieler.sprung()

    self.spieler.update()
```

Mit ***keyboard.up*** testen wir, ob die UP-Taste gedrückt wurde. Wenn ja, dann lösen wir einen Sprung aus. Ausserdem muss bei jedem Schritt die Update-Methode des Spielers aufgerufen werden.

Der Spieler kann jetzt springen. Trotzdem werden noch nicht alle Anforderungen erfüllt.

Der Spieler hat aber die Möglichkeit zu springen. Wenn er ganz unten steht, springt er sehr schnell auf eine bestimmte Höhe. Ausgelöst wird das durch einen Druck auf die UP-Taste (Pfeil nach oben).
Das ist erfüllt.

Sobald die Taste losgelassen wird, kehrt die Figur sofort auf den Boden zurück.
Das funktioniert nicht. Wir kehren langsam auf den Boden zurück. Das muss schneller gehen.

Etwas Magie ist auch dabei: durch dauerhaftes Drücken der Taste, kann der Flug verlängert werden.
Auch noch nicht erfüllt. Wir müssen den Fall verlangsamen.

Beide fehlenden Anforderungen lassen sich durch eine kleine Änderung in der Spiele-Klasse erfüllen. Zuerst setzen wir die normale Fallgeschwindigkeit auf 20. Dadurch fällt der Spieler sehr schnell.

```
def update(self):
    # Wenn wir nicht am Boden sind, dann fallen wir
    if self.y < self.baseY:
        self.y += 20 # Fallgeschwindigkeit
```

Wenn die UP-Taste gedrückt bleibt, erhalten wir laufend Sprung-Befehle. So können wir den Sprung verlangsamen, falls der Spieler noch nicht ganz unten ist.

```
def sprung(self):
    if self.y >= self.baseY: # ist ganz unten
        self.y = self.baseY - 350 # Sprung auf maximale Höhe
    else:
        self.y -= 19 # Muss kleiner als die Fallgeschwindigkeit sein
```

Die Animation

Der Spieler soll auch rennen können. Diesen Eindruck erreichen wir mit einer Animation. Dazu laden wir in der Spielerklasse mehrere Bilder.

```
def __init__(self, x, y):
    ...

    # Es wird eine Liste mit den Bildern für die Animation angelegt
    images = []
    for i in range(10):
        images.append("run__{:03d}".format(i))
    self.images = images
```

Die eigentliche Animation findet in ***update()*** statt.

```
def update(self):
    if not self.aktiv: return
    # Wenn wir nicht am Boden sind, dann fallen wir
    if self.y < self.baseY:
        self.y += 20
    self.next_image()
```

Lektion 3 - Die Hindernisse

Die Hindernisse erscheinen in zufälligen Abständen. Für das einzelne Hindernis wird die Klasse ***Hindernis*** in der Datei ***hindernis.py*** erstellt.

Die Klasse Hindernis

Ein Hindernis wird vom Hauptprogramm nach zufälligen Kriterien erzeugt und in die Liste eingefügt.
Es wird rechts dargestellt, sobald es in die Liste eingefügt wird.
Dann läuft es automatisch von rechts nach links und verschwindet beim Erreichen des linken Randes wieder.
Falls der Spieler mit dem Hindernis kollidiert, fällt es um und das Spiel ist beendet.

Die Klasse Hindernis ist verhältnismässig einfach:

```
class Hindernis(Actor):
    # Es muss eine Startposition angegeben werden
    def __init__(self, x, y):
        super().__init__('cactus')
        self.aktiv = False
        self.scale = 2 # vergrössern

        # Das Element wird platziert
        self.angle = 0
        self.x = x
        self.y = y

        self.start() # Das Hindernis wird automatisch gestartet

    def start(self):
        self.aktiv = True

    def stopp(self):
        self.aktiv = False

    def kollision(self):
        self.angle = -90 # Hindernis fällt um

    def update(self):
        if not self.aktiv: return
        self.x -= 2
```

Für das Hindernis wird das Bild ***cactus.png*** geladen. Mit ***self.scale = 2*** wird das Element vergrössert.
Nebst der Position wird auch noch ein Winkel verwaltet. Bei der Erstellung ist der Winkel 0, der Kaktus steht aufrecht. Nach einer Kollision liegt der Kaktus am Boden (self.angle = -90).

Dafür muss die Methode ***kollision()*** aufgerufen werden.
In ***update()*** bewegt sich der Kaktus einfach zwei Pixel nach links.

Diese Klasse hat sehr wenig eigene Funktionalität. Die eigentliche Arbeit wird durch die Klasse ***Spiel*** verrichtet.

Das Erzeugen von Hindernissen

Wir sind jetzt wieder in der Datei ***game.py***.

Die Klasse ***Hindernis*** muss importiert werden. Ausserdem benötigen wir ***random*** für den Zufallsgenerator.

```
import pgzrun
from spieler import Spieler
from hindernis import Hindernis
import random
```

Damit die Hindernisse in wechselnden Abständen erscheinen können, müssen wir die Zeit verwalten, bei der das nächste Hindernis erzeugt werden muss. Wir benötigen hier nicht die aktuelle Zeit, wir können auch einfach die Schritte zählen.

```
class Spiel:
    def __init__(self):
        ...
        self.hindernisse = []   # Eine leere Liste für die Hindernisse
        self.hinderniszeit = 0
```

Ob ein Hindernis erzeugt werden soll, wird in ***update()*** festgelegt. Danach muss noch ***update()*** aller Hindernisse aufgerufen werden.

```
def update(self):
    if keyboard.up:
        self.spieler.sprung()

    self.spieler.update()

    self.hinderniszeit -= 1
    if self.hinderniszeit <= 0:# neues Hindernis erzeugen
        hindernis = Hindernis(WIDTH-100,HIMMEL)
        self.hindernisse.append(hindernis)
        self.hinderniszeit = random.randint(50, 300)

    for hindernis in self.hindernisse:
        hindernis.update()
```

Wenn ein neues Hindernis erzeugt wird, wir ***hinderniszeit*** auf einen zufälligen Wert zwischen 50 und 300 gesetzt. Bei jedem Schritt wird diese Zeit um 1 verkleinert. Sobald sie kleiner als 0 wird, wird ein neues Hindernis erzeugt und der Liste hinzugefügt.

Das Hindernis muss jetzt noch dargestellt werden. Wir stellen alle Hindernisse dar, die sich in der Liste befinden.

```
def zeichne_hindernisse(self):
    for hindernis in self.hindernisse:
        hindernis.draw()
```

Hindernisse wieder verschwinden lassen

Die Hindernisse sind zwar nicht mehr sichtbar, wenn sie über den linken Rand hinauskommen. Sie sind aber in der Liste noch vorhanden und diese wird dadurch immer grösser.

Wir müssen also testen, ob sich ein Hindernis noch im sichtbaren Bereich befindet. Wenn nicht, soll es aus der Liste entfernt werden. Bei dieser Gelegenheit können wir gleich noch den Punktestand um eins erhöhen.

Das geschieht ebenfalls in der ***update()*** - Methode von ***Spiel***.

```
for hindernis in self.hindernisse:
    if hindernis.x < 0:
        self.hindernisse.remove(hindernis)
        self.punkte += 1
    hindernis.update()
```

Kollision mit dem Spieler

Für jedes Hindernis in der Liste testen wir, ob der Spieler mit ihm kollidiert ist. Wenn ja, dann ist das Spiel beendet.

Auch dafür passen wir die ***update()*** - Methode von ***Spiel*** an.

```
for hindernis in self.hindernisse:
    if hindernis.colliderect(self.spieler):
        hindernis.kollision()
        self.spieler.kollision()
        self.stopp()
    if hindernis.x < 0:
        self.hindernisse.remove(hindernis)
        self.punkte += 1
    hindernis.update()
```

Das Spiel wird jetzt beendet und der Kaktus fällt um. Nun soll der Spieler ebenfalls umfallen. Dazu können wir die ***kollision()*** - Methode von Spieler verwenden.

```
def kollision(self):
    self.y = self.baseY
    self.angle = -90 # Spieler fällt um
    self.stopp()
```

In start() müssen wir sicherstellen, dass der Spieler aufrecht steht.

```
def start(self):
    self.x = self.baseX
    self.y = self.baseY
    self.angle = 0
    self.aktiv = True
```

Lektion 4 - Musik und Soundeffekte

Hintergrundmusik

Sobald das Spiel gestartet wird, soll eine Hintergrundmusik abgespielt werden. Im Verzeichnis ***music*** befindet sich bereits die Datei ***venus.wav***. Diese benutzen wir für die Hintergrundmusik.

Mit ***music.play('venus.wav')*** kann das Abspielen gestartet werden.
Wir fügen die Zeile in der Klasse ***Spiel*** ein.

```
def start(self):
    self.Punkte = 0
    self.hindernisse.clear()
    self.spieler.start()
    music.play('venus.wav')
    self.aktiv = True
    self.finished = False
```

Mit ***music.stop()*** herrscht wieder Ruhe.

```
def stopp(self):
    music.stop()
    self.aktiv = False
    self.finished = True
```

Soundeffekte

Game Over soll einen Soundeffekt auslösen. Die Sounddatei ***gameover.wav*** befindet sich bereits im Verzeichnis ***sounds***.

```
for hindernis in self.hindernisse:
    if hindernis.colliderect(self.spieler):
        hindernis.kollision()
        self.spieler.kollision()
        self.stopp()
        sounds.gameover.play()
```

Anhang

Figur

Ist immer von Actor abgeleitet
Als Grundlage wird die erweiterte ***Actor*** - Klasse aus dem Modul pgzhelper verwendet. Deshalb muss dieses importiert werden.
```
import pgzrun
from pgzhelper import *
```

Der Konstruktor der Basisklasse ***Actor*** muss aufgerufen werden. Dabei wird ein Bild angegeben.
```
super().__init__('run__000')
```

Besitzt eine aktiv - Eigenschaft
Diese Eigenschaft wird im Konstruktor immer auf False gesetzt.
```
self.aktiv = False
```

Besitzt die Methoden start() und stopp()
start() und stop() müssen mindestens ***aktiv*** True oder False setzen.
```
def start(self):
    self.aktiv = True

def stopp(self):
    self.aktiv = False
```

Sichtbaren Figuren wird die Startpositionen mitgegeben (x- und y-Koordinaten)
```
def __init__(self, x, y):
    super().__init__('run__000')
    self.aktiv = False

    # Die aktuelle Position
    self.x = x
    self.y = y
```

Sichtbare Figuren haben eine kollision() - Methode
Hier wird der Zustand nach einer Kollision festgelegt. Üblicherweise wird mindestens stopp() aufgerufen.
```
def kollision(self):
   self.stopp()
```

Implementiert eine update() - Methode
Update wird nur ausgeführt, wenn aktiv auf True steht.
```
def update(self):
    if not self.aktiv: return
```

Dateivorlage für eine Figur

```
import pgzrun
from pgzhelper import *

class Figur(Actor):
    def __init__(self, x, y):
        super().__init__('BildDatei')
        self.aktiv = False
        # Die aktuelle Position
        self.x = x
        self.y = y

    def start(self):
        self.aktiv = True

    def stopp(self):
        self.aktiv = False

    def kollision(self):
        self.stopp()

    def update(self):
        if not self.aktiv: return
```

Eine Figur kann weitere Methoden und Eigenschaften besitzen.

Spiel

Diese Klasse stellt das Spielfeld dar und kontrolliert das Spiel.

Besitzt eine aktiv - Eigenschaft

Diese Eigenschaft wird im Konstruktor immer auf False gesetzt.

```
self.aktiv = False
```

Besitzt die Methoden start() und stopp()

start() und stop() müssen mindestens ***aktiv*** True oder False setzen.

```
def start(self):
    self.aktiv = True

def stopp(self):
    self.aktiv = False
```

Im Konstruktor werden alle Figuren erzeugt

```
def __init__(self):
    self.aktiv = False
    # Figuren erzeugen
    # Beispiel: self.figur = Figur(10,20)
```

Besitzt eine zeichne_spielfeld() - Methode

Alle Elemente mit Ausnahme der Figuren werden hier gezeichnet.

```
def zeichne_spielfeld(self):
    # Beispiel: farbe = (163, 232, 254)
    # Beispiel: screen.draw.filled_rect(Rect(0, 0, 500, 600), farbe)
```

Besitzt je eine zeichne_ - Methode pro Figur

```
def zeichne_figur(self):
    self.figur.draw()
```

Besitzt eine draw() - Methode

Diese Methode ruft alle zeichne_ - Methoden auf.

```
def draw(self):
    self.zeichne_spielfeld()
    self.zeichne_figur()
```

Besitzt eine update() - Methode

Darin ist die ganze Spiele - Logik untergebracht.
Zuerst muss die Möglichkeit geschaffen werden, ein Spiel zu starten.
Falls das Spiel nicht aktiv ist, wird die Methode verlassen. Andernfalls geht es mit der Spielelogik weiter.

```
def update(self):
    # Start-Taste nur abfragen, wenn das Spiel nicht aktiv ist
    if not self.aktiv and keyboard.s:
        # neues Spiel starten
        self.start()

    # Wenn das Spiel nicht aktiv ist, hier abbrechen
    if not self.aktiv: return
```

Eine Dateivorlage für das Hauptprogramm

```
import pgzrun
# Beispiel: from figur import Figur

# Das Spielfeld
WIDTH=800
HEIGHT=600
TITLE = "Ein Spiel"

class Spiel:
    def __init__(self):
        self.aktiv = False
        # Figuren erzeugen
        # Beispiel: self.figur = Figur(10,20)

    def start(self):
        self.aktiv = True

    def stopp(self):
        self.aktiv = False

    def zeichne_spielfeld(self):
        pass
        # Beispiel: farbe = (163, 232, 254)
        # Beispiel: screen.draw.filled_rect(Rect(0, 0, WIDTH, HEIGHT), farbe)

    def zeichne_figur(self):
        pass
        # Beispiel: self.figur.draw()

    def draw(self):
        self.zeichne_spielfeld()
        self.zeichne_figur()

    def update(self):
        # Start-Taste nur abfragen, wenn das Spiel nicht aktiv ist
        if not self.aktiv and keyboard.s:
            # neues Spiel starten
            self.start()

        # Wenn das Spiel nicht aktiv ist, hier abbrechen
        if not self.aktiv: return

spiel = Spiel()

def update():
    spiel.update()

def draw():
    spiel.draw()

pgzrun.go()
```