Space Invaders mit Python und der Turtle

Space Invaders ist ein Klassiker in der Geschichte der Computerspiele und erschien erstmals 1978 für Spielekonsolen. In japanischen Spielhallen war es so populär, daß nach kurzer Zeit die 100-Yen-Münzen im ganzen Land knapp wurden.

Die 1979 erschienene Umsetzung des Spiels auf die bis dahin eher schleppend laufende Spielkonsole Atari 2600 brachte den Durchbruch der Videospiele im Heimmarkt.

In diesem Beitrag möchte ich eine (eigenwillige) Version dieses Klassikers mit Python und dem Turtle-Modul programmieren. Ich beginne mit einem schlichten Template und arbeite mich dann Stage für Stage durch bis zum fertigen Spiel.

Stage 1: Ein Template für ein Turtle-Spieleprogramm

Das Template ist von ergreifender Schlichtheit und macht nichts anderes, als ein 700x700 Pixel große Fenster mit einem schwarzen Hintergrund zu öffnen:

```
1  # Space Invaders Stage 1: Template für Turtle-Programme
2
3  import turtle as t
4
5  WIDTH = 700
6  HEIGHT = 700
```

```
# Hier kommen die Klassendefinitionen hin


# Initialisierung

wn = t.Screen()
wn.bgcolor("#000000")
wn.setup(width = WIDTH, height = HEIGHT)
wn.title("Space Invaders - Stage 1")

# Bildschirm-Refresh ausschalten
wn.tracer(0)

def exitGame():
    global keepGoing
    keepGoing = False

# Auf Tastaturereignisse lauschen
t.listen()
t.onkey(exitGame, "Escape") # Escape beendet das Spiel

keepGoing = True
while keepGoing:
    wn.update()  # Bildschirm-Refresh einschalten und den gesam
```

Es importiert erst einmal das Turtle-Modul und legt die Konstanten für die Breite und Höhe des Fensters fest. Dann wird das Fenster initialisiert und mit einem Titel versehen.

Eine Besonderheit ist der Befehl `wn.tracer(0)`. Denn im »Normalbetrieb« versucht jede Turtle kontinuierlich, jeden Graphikbefehl sofort abzuarbeiten und zu zeichnen. Das wird mit diesem Befehl ausgeschaltet, solange wie er gültig ist, wird mit allen Zeichnungen nur ein *Buffer* gefüllt.

Erst der Befehl `wn.update()` kopiert den Inhalt dieses Buffers auf den

Bildschirm. Dies passiert in der Hauptschleife des Programms, so daß nur einmal je Durchlauf der Inhalt des Buffers auf den Bildschirm kopiert wird. Dies beschleunigt die Graphikausgabe enorm.

Diese Version von *Space Invaders* soll über die Tastatur gesteuert werden, dafür muß ein *Listener* eingerichtet werden.

Zur Zeit lauscht er nur auf die *Escape*-Taste, mit der das Spiel beendet wird.

Stage 2: Die Spielewelt zeichnen

Leider ist das Weite und Höhe des Fensters im Turtle-Modul »brutto«, also inklusiv der Ränder, die auch noch je nach Betriebssystem unterschiedlich breit sein können. Daher sollte die Spielewelt deutlich innerhalb des Fensters liegen. Ich habe mich für eine 600x600 Pixel große Welt entschieden, die ich im Fenster zentriere. Dazu habe ich erst einmal zwei weitere Konstanten definiert:

```
1  WW = 600
2  WH = 600
```

Natürlich wollte ich *Space Invaders* mit allen Vorzügen der Objektorienterung implementieren, und so habe ich erst einmal die Klasse `GameWorld` definiert:

```
1  class GameWorld(t.Turtle):
2
3      def __init__(self):
4          t.Turtle.__init__(self)
5          self.penup()
6          self.hideturtle()
7          self.speed(0)
8          self.color("white")
9          self.pensize(3)
10         self.keepGoing = True
11
```

```
    def draw_border(self):
        self.penup()
        self.goto(-WW/2, -WH/2)
        self.pendown()
        for i in range(4):
            self.forward(WW)
            self.left(90)

    def exit_game(self):
        self.keepGoing = False
```

Sie ist für nichts anderes zuständig, als einmal um die Spielewelt einen weißen Rahmen zu ziehen und zum anderen habe ich die Funktion `exit_game()` zu einer Methode dieser Klasse gemacht. Damit verhindere ich, daß ich `keepGoing` als *global* definieren muß und halte den Namensraum sauber. Somit kann ich die in *Stage 1* definierte globale Funktion `exitGame()` eliminieren.

Das bedingt allerdings folgende Änderung im *Listener*:

```
t.listen()
t.onkey(world.exit_game, "Escape") # Escape beendet das Spiel
```

Darüber wird die Welt initialisiert und die Methode zum Zeichnen des Randes aufgerufen:

```
# Objekte initialisieren
world = GameWorld()
world.draw_border()
```

Alles andere bleibt gleich. Wenn wir nun das Programm aufrufen, wird die Welt initialisiert und mit einem weißen Rand versehen. Und nach wie vor beendet die *Escape*-Taste das Programm, auch wenn der Listener nun eine Methode und keine Funktion mehr aufruft

keine Funktion mehr aufruft.

Stage 3: Die Klasse Sprite und den Spieler hinzufügen

Ursprünglich war ein Sprite (englisch für *Geistwesen* oder *Kobold*) ein Graphikobjekt, das von der Graphikhardware über das Hintergrundbild beziehungsweise den restlichen Inhalt der Bildschirmanzeige eingeblendet wird. Die Positionierung wurde dabei komplett von der Graphikhardware erledigt.

Heute ist die echte Sprite-Technik überholt, vor allem, da Computer inzwischen schnell genug sind, ohne Probleme tausende spriteartige Objekte auf dem Bildschirm darzustellen und zugleich den Hintergrund in ursprünglicher Form wiederherzustellen. Auch der dafür nötige Speicherplatz ist weniger wichtig geworden. Dennoch hat sich der Begriff *Sprite* auch verallgemeinernd für Objekte gehalten, die nun per Software (statt Graphikhardware) über den Hintergrund eingeblendet werden. Solche *Software-Sprites* sind heute die Grundlage fast jeden Computerspiels und daher ist es sinnvoll, ihnen eine eigene Klasse zu widmen:

```
1  class Sprite(t.Turtle):
2
3      def __init__(self, tshape, tcolor):
4          t.Turtle.__init__(self)
5          self.penup()
6          self.speed(0)
7          self.shape(tshape)
8          self.color(tcolor)
9          self.speed = 1
```

Auch Klasse `Sprite` erbt alle Eigenschaften der Klasse `Turtle`. Sie ist momentan noch ein wenig schmalbrüstig, aber schon mächtig genug, daß von ihr der Spieler abgeleitet werden kann (schließlich erbt sie alle Methoden der

`Turtle`):

```
class Actor(Sprite):

    def __init__(self, tshape, tcolor):
        Sprite.__init__(self, tshape, tcolor)
        self.color = tcolor
        self.speed = 10
        self.x = 0
        self.y = -280
        self.setheading(90)
        self.goto(self.x, self.y)

    def go_left(self):
        self.x -= self.speed
        if self.x <= -WW/2 + 20:
            self.x = -WW/2 + 20
        self.setx(self.x)

    def go_right(self):
        self.x += self.speed
        if self.x >= WW/2 - 20:
            self.x = WW/2 - 20
        self.setx(self.x)
```

Da ich den Spieler mit Hilfe der Pfeiltasten nach rechts oder links bewegen möchte, er aber sonst am unteren Fensterrand bleibt (sich also nicht vertikal bewegen soll), habe ich die Methoden `go_left()` und `go_right()` implementiert. Sie enthalten jeweils eine Ränderabfrage, so daß der Spieler nicht versehentlich die Spielewelt verlassen kann.

Ansonsten ist er einfach ein purpurfarbenes Dreieck, das um 90 Grad gedreht nach Norden zeigt:

```
player = Actor("triangle", "purple")
```

Natürlich mußte auch noch der *Listener* um die beiden Pfeiltasten erweitert werden:

```
t.onkey(player.go_right, "Right")
t.onkey(world.exit_game, "Escape") # Escape beendet das Spiel
```

Wir haben nun eine Spielfigur, die wir mit den Pfeiltasten nach rechts und links bewegen können. Als nächstes sollten wir den Spieler einen Gegner spendieren.

Stage 4: Die Klasse Enemy für den Gegner

Um einem Gegner zu implementieren, wird erst einmal eine Klasse `Invader` benötigt, die (Überraschung!) ebenfalls von `Sprite` abgeleitet wird:

```
class Invader(Sprite):

    def __init__(self, tshape, tcolor):
        Sprite.__init__(self, tshape, tcolor)
        self.color = tcolor
        self.speed = 2
        self.x = -200
        self.y = 250
        self.goto(self.x, self.y)

    def move(self):
        self.x += self.speed
        if self.x >= WW/2 - 20 or self.x <= -WW/2 + 20:
            self.y -= 40
            self.sety(self.y)
            self.speed *= -1
        self.setx(self.x)
```

Der Gegener ist – wie die Initialisierung zeigt – einfach eine grüne Scheibe:

Der Gegner ist – wie die Initialisierung zeigt – einfach eine grüne Scheibe:

```
enemy = Invader("circle", "green")
```

Da sich der Gegener autonom mit der Methode `move()` bewegt, müssen wir diese in der Hauptschleife aufrufen:

```
while world.keepGoing:
    wn.update()  # Bildschirm-Refresh einschalten und den gesam
    enemy.move()
```

Auch die Methode `move()` fragt die Ränder ab. Wird einer der Ränder erreicht, wird die y-Koordinate um 40 Pixel nach unten verschoben. Und eine Richtungsänderung wird einfach mit

```
            self.speed *= -1
```

erreicht, da ja bekannt die Multiplikation mit `-1` das Vorzeichen wechselt.