Einführung in die 2D-Spieleprogrammierung mit PyGame

Ralf Adams (TBS1, Bochum)

Version 0.1 vom 14. Dezember 2021

Inhaltsverzeichnis

1 Ziele **4**

2 Mein erstes Spiel **5**

2.1 Mein erstes Programm 5

2.2 Was war neu? 10

3 Grafikprimitive **11**

Abbildungsverzeichnis

2.1 Eine einfache grüne Spielfläche 7
2.2 Ressourcenverbrauch ohne Taktung 8
2.3 Ressourcenverbrauch mit Taktung 9

1 Ziele

Dieses Skript ist eine Einführung in die Programmierung zweidimensionaler Spiele mit Hilfe von PyGame. Als Programmiersprache liegt Python zu Grunde.

Im ersten Teil werden die wichtigsten Konzepte anhand einfacher Beispiele eingeführt. Im zweiten Teil wird ein Spielprojekt vollständig durchprogrammiert und damit der Einsatz der Techniken verdeutlicht.

Es bleibt offen, welche Entwicklungsumgebung verwendet wird; ich verwende Visual Code.

Für eine Rückmeldung bei groben Patzern wäre ich sehr dankbar: adams@tbs1.de.

2 Mein erstes Spiel

2.1 Mein erstes Programm

Quelltext 2.1: Mein erstes *Spiel*, Version 1.0

```
1  import pygame                                        # PyGame-Modul
2  import os
3
4  if __name__ == '__main__':
5      os.environ['SDL_VIDEO_WINDOW_POS'] = "10,␣50" # Fensterposition
6      pygame.init()                                    # Subsystem starten
7      pygame.display.set_caption('Mein␣erstes␣PyGame-Programm');# Fenstertitel
8
9      screen = pygame.display.set_mode((600, 400))  # Fenster erzeugen
10     running = True
11     while running:                                   # Hauptprogammschleife: start
12         for event in pygame.event.get():             # Ermitteln der Events
13             if event.type == pygame.QUIT:            # Fenster X angeklickt?
14                 running = False
15         screen.fill((0, 255, 0))                     # Spielfläche einfärben
16         pygame.display.flip()                        # Doublebuffer austauschen
17
18     pygame.quit()                                    # Subsystem beenden
```

Um PyGame verwenden zu können, muss das Modul `pygame` importiert werden (Zeile 1). Danach steht uns Konstanten, Funktionen und Klassen des Namensraums zur Verfügung.

In Zeile 5 wird die Umgebungsvariable gesetzt, die erstmal nichts mit PyGame zu tun hat. Vielmehr wird hier die Umgebungsvariable `SDL_VIDEO_WINDOW_POS` des Betriebssystems gesetzt. Diese steuert die linke obere Startposition meines Fensters bezogen auf den ganzen Bildschirm.

SDL_VIDEO_WINDOW_POS

PyGame ist nicht nur der Aufruf von Funktionen oder die Instantiierung von Klassen, sondern vielmehr wird ein ganzes Subsystem verwendet. Dieses Subsystem muss erst noch gestartet werden. Dabei klinkt sich PyGame in die relevanten Komponenten des Betriebssystems ein, damit diese im Spiel verwendet werden können. In Zeile 6 wird der ganze PyGame-Motor mit `init()` angeworfen. Man könnte auch nur die Komponenten starten, die gerade gebraucht werden wie beispielsweise die Soundunterstützung mit `pygame.mixer.init()`.

init()

Wir werden uns nur mit Spielen beschäftigen, die unmittelbar auf dem Desktop laufen. Oder anders herum: Wir werden keinen Game-Server implementieren. Daher brauchen unsere Spiele eine *Spielfläche*/ein Fenster innerhalb dessen sich alles abspielt. Die Funktion `pygame.display.set_mode()` liefert mir einen solche Spielfläche. Die Funkti-

set_mode()

on bekommt in Zeile 9 einen(!) Übergabeparameter – nämlich die Breite und die Höhe des Fensters als ein 2-Tupel. Unser Fenster ist also 600*px* breit und 400*px* hoch. Als Rückgabe bekomme ich ein `pygame.Surface`-Objekt, was ungefähr sowas wie ein Bitmap ist. Dem Fenster kann ich dann noch mit `pygame.display.set_caption()` eine Titelüberschrift verpassen (siehe Zeile 7). set_caption()

Das Spiel selbst – so wie auch alle zukünftigen Spiele – laufen innerhalb einer Hauptprogammschleife. Hier startet die Schleife in Zeile 11 und endet in Zeile 18. Innerhalb dieser Schleife werden zukünftig immer drei Dinge passieren:

1. Ereignisse auslesen und verarbeiten: Wie in Zeile 12f werden Maus-, Tastatur- oder Konsolenereignisse festgestellt und an die Spielelemente weitergegeben. In unserem Fall wird lediglich das Anklicken des X im Fenster oben rechts registriert.
2. Zustand der Spielelemente aktualisieren: Basierend auf den oben festgestellten Ereignissen und den Zuständen der Spielelemente, werden die neuen Zustände ermittelt (Spieler bewegt sich, Geschoss prallt auf, Punkte erhöhen sich etc.). In unserem Fall wird nur das Flag `running` der Hauptprogrammschleife auf `false` gesetzt.
3. Bitmaps der Spielelemente malen: Die Spielelemente haben eine neue Position oder ein neues Aussehen und müssen deshalb neu gemalt werden. In diesem Minimalbeispiel wird lediglich Zeile 15 der Hintergrund der Spielfläche eingefärbt und anschließend in Zeile 16 der Doublebuffer mit `pygame.display.flip()` ausgetauscht. flip()

Exkurs: Was ist der *Doublebuffer*? Dies ist ein zweiter Speicherbereich, der genauso groß ist wie der Bildschirmspeicher. Wird jetzt etwas auf die Spielfläche gezeichnet, passiert dies zunächst auf diesem zweiten Speicher. Erst wenn alle Spielelemente ihr neues Aussehen gemalt haben, wird mit einem Schlag der alte Bildschirmspeicher mit dem zweiten ausgetauscht. Bei bestimmten Hardware- oder Grafikkonfigurationen kann es passieren, dass der Bildschirmspeicher neu gemalt wird, obwohl das Spiel noch nicht alle neuen Zustände abgebildet hat. Dadurch können hässliche Artefakte entstehen. Durch das Doublebuffering wird dieser Effekt vermieden. Doublebuffer

Zurück zum Quelltext – es sind noch einige Elemente unerklärt. PyGame schleust durch den Aufruf von `pygame.init()` einen Horchposten in das Betriebssystem. Und zwar horcht PyGame die *Message-Queue* ab. Dort werde vom Betriebssystem alle Meldungen eingesammelt, die durch Ereignisse ausgelöst werden. Dies können USB-Anschlussmeldungen, SSD-Fehlermeldungen, Mausaktionen, Programmstarts bzw. -abstürze usw. sein. PyGame fischt nun aus der Message-Queue mit Hilfe von `pygame.event.get()` alle Events, die das Spiel betreffen könnten heraus. Mit Hilfe einer for-Schleife iteriere ich nun die Ereignisse und picke die für mich interessanten heraus. event.get()

Dabei überprüfe ich erstmal, was für ein Ereignistyp (`event.type`) mir da angeboten wird. Derzeit ist für mich nur der Typ `pygame.QUIT` wichtig. Dieser Typ wird ausgelöst, wenn das Betriebssystem eine *Beenden*-Nachricht an die Anwendung sendet. Falls ist nun eine solche Nachricht empfange, setzte ich das Flag `running` auf `False`, so dass die Hauptprogrammschleife beendet wird. event.type pygame.QUIT

Falls ich dieses Signal nicht empfange, läuft die Hauptprogrammschleife fröhlich weiter und füllt in Zeile 15 die gesamte Spielfläche mit `screen.fill()` mit einer Farbe – hier grün – ein. Bitte beachten Sie, dass ähnlich wie in Zeile 9 die Funktion einen Übergabeparameter – nämlich ein 3-Tupel – erwartet. Dieses 3-Tupel kodiert die Farbe durch RGB-Angaben zwischen 0 und 255.

RGB

Verbleibt noch Zeile 16: Dort wird die Funktion `pygame.quit()` aufgerufen. Diese Funktion ist quasi das Gegenteil von `pygame.init()` in Zeile 6. Alle reservierten Ressourcen werden wieder freigegeben und die PyGame-Horchposten werden wieder aus dem System entfernt. Rufen Sie diese Funktion unbedingt immer am Ende Ihrer Anwendung auf; beenden Sie nicht einfach das Spiel. Der Unterschied entspricht dem einfachen Herauslaufen aus der Wohnung und dem ordnungsgemäßen Lichtausmachen und Türabschließen beim Verlassen der Wohnung.

quit()

Wenn Sie jetzt die Anwendung starten, bekommen Sie eine schmucke grüne Spielfläche zu sehen. Beenden können Sie diese durch das Anklicken des X im Fensterrahmen oben rechts.

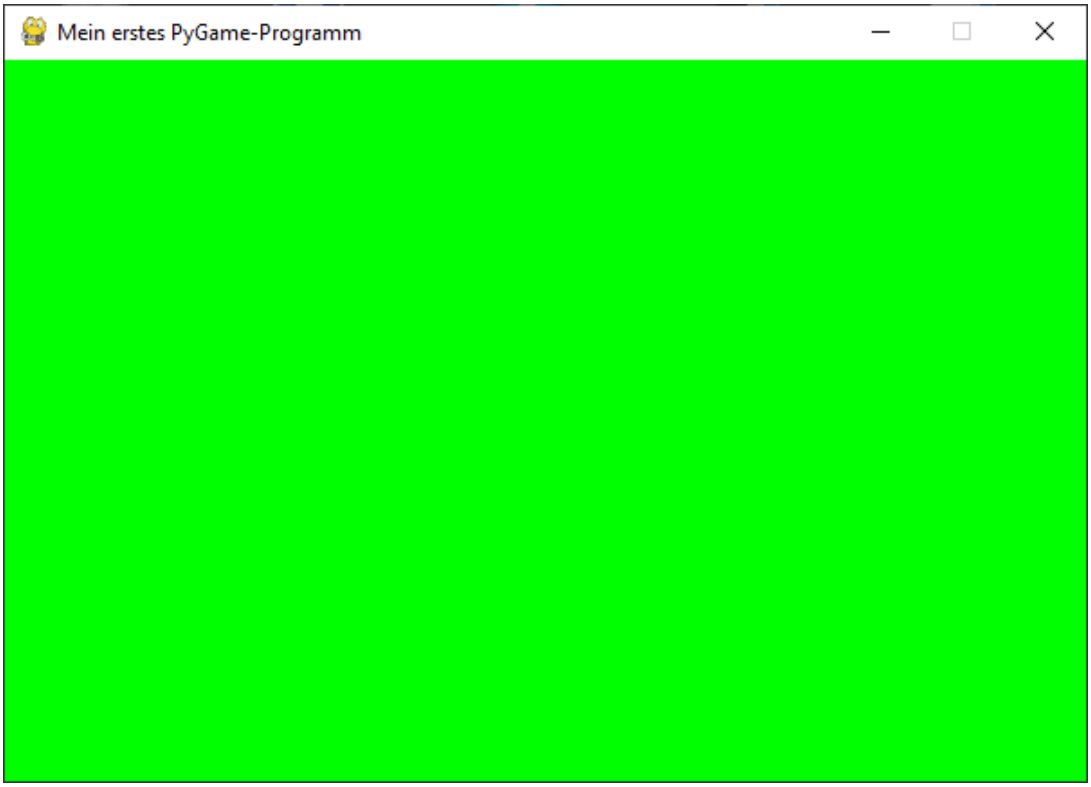

Abbildung 2.1: Eine einfache grüne Spielfläche

Wenn wir uns das Spiel mal im Task-Manager anschauen (siehe Abbildung 2.2 auf der nächsten Seite), könnten wir leicht überrascht sein: Es werden rund 30% der CPU-Zeit für dieses *IchMacheJaEigentlichGarNichts*-Spiel verbraucht.

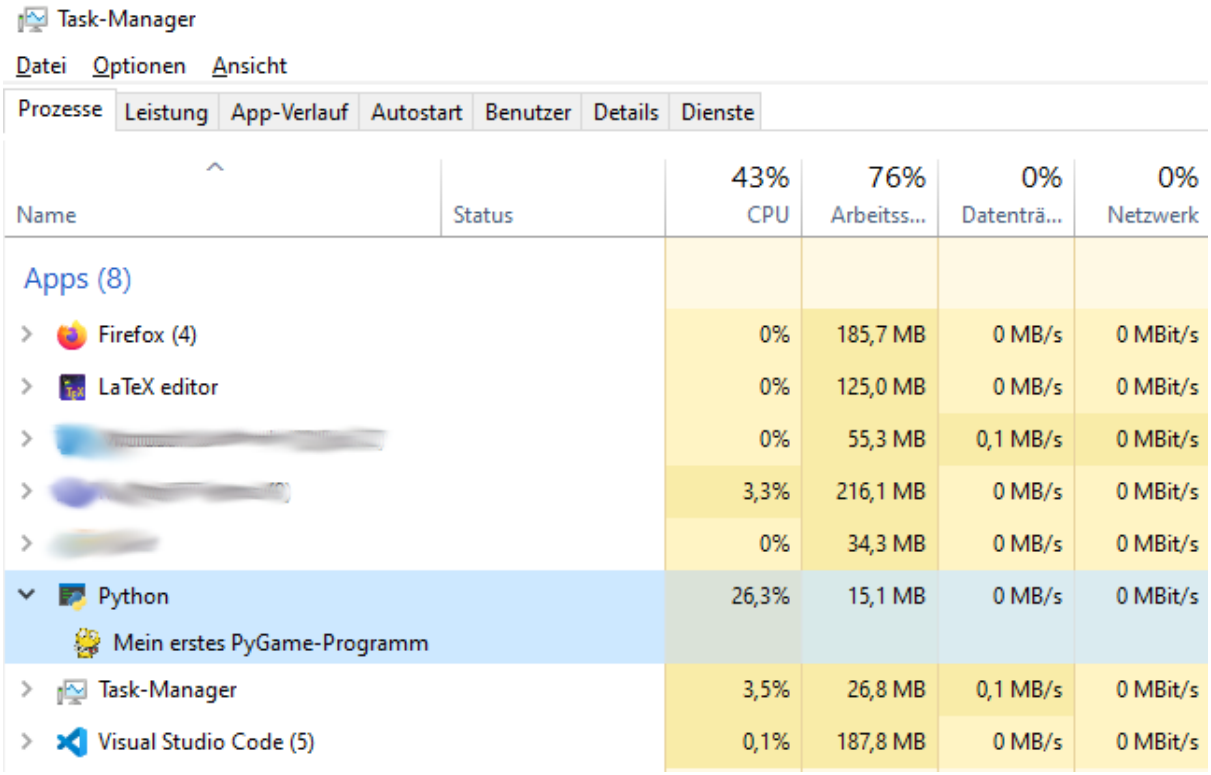

Abbildung 2.2: Ressourcenverbrauch ohne Taktung

Wenn wir uns die Hauptprogrammschleife anschauen, sollte es allerdings nicht wirklich verwundern. Da wird ungebremst ein Bitmap auf den Bildschirm gemalt und dass ohne Unterbrechung. Das ist natürlich überhaupt nicht sinnvoll. Vielmehr sollte jeder Schleifendurchlauf genügend Zeit zur Verfügung stellen, um die Ereignisse einzusammeln, die neuen Zustände zu berechnen und erst dann die Bildschirmausgabe zu generieren. Die Bildschirmausgabe selbst sollte auch nicht beliebig schnell und oft passieren, sondern in der Regel reichen 60 Bilder pro Sekunde, um eine Bewegung als flüssig wahrzunehmen.

Quelltext 2.2: Mein erstes *Spiel*, Version 1.1

```
1  import pygame
2  import os
3
4  if __name__ == '__main__':
5      os.environ['SDL_VIDEO_WINDOW_POS'] = "10,␣50"
6      pygame.init()
7      pygame.display.set_caption('Mein␣erstes␣PyGame-Programm');
8
9      screen = pygame.display.set_mode((600, 400))
10     clock = pygame.time.Clock()                    # Clock-Objekt
11
12     running = True
13     while running:
14         clock.tick(60)                             # Taktung auf 60 fps
15         for event in pygame.event.get():
16             if event.type == pygame.QUIT:
17                 running = False
18         screen.fill((0, 255, 0))
19         pygame.display.flip()
20
21     pygame.quit()
```

In Zeile 10 wird zur Taktung ein `pygame.time.Clock`-Objekt erzeugt. Mit Hilfe dieses Objektes können verschiedene zeitbezogene Aufgaben bewältigt werden, wir brauchen es im Moment nur für Taktung in Zeile 14. Dort wird `pygame.time.Clock.tick()` mit einer Framerate gemessen in *frames per second (fps)* aufgerufen. Diese Funktion sorgt dafür, dass die Anwendung nun mit maximal 60 fps abläuft. Dies ist an dem deutlich reduzierten CPU-Verbrauch in Abbildung 2.3 zu erkennen.

Clock

tick()

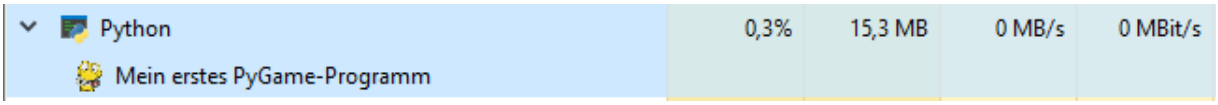

Abbildung 2.3: Ressourcenverbrauch mit Taktung

Hinweis: In der PyGame-Dokumentation wird darauf verwiesen, dass die Funktion `tick()` zwar sehr ressourcenschonend, aber etwas ungenau sei. Falls Genauigkeit aber bei der Taktung wichtig ist, wird die Funktion `tick_busy_loop()` empfohlen. Deren Nachteil ist, dass sie aber erheblich mehr Rechenzeit als `tick()` verbraucht.

tick_busy_loop()

2.2 Was war neu?

- `import pygame`:
  https://www.pygame.org/docs/tut/ImportInit.html
- `os.environ['SDL_VIDEO_WINDOW_POS']`:
  https://docs.python.org/3/library/os.html#os.environ
- `pygame.init()`:
  https://www.pygame.org/docs/ref/pygame.html#pygame.init
- `pygame.quit()`:
  https://www.pygame.org/docs/ref/pygame.html#pygame.quit
- `pygame.display.set_mode()`:
  https://www.pygame.org/docs/ref/display.html#pygame.display.set_mode
- `pygame.display.set_caption()`:
  https://www.pygame.org/docs/ref/display.html#pygame.display.set_caption
- `pygame.display.flip()`:
  https://www.pygame.org/docs/ref/display.html#pygame.display.flip
- `pygame.time.Clock`:
  https://www.pygame.org/docs/ref/time.html#pygame.time.Clock
- `pygame.time.Clock.tick()`:
  https://www.pygame.org/docs/ref/time.html#pygame.time.Clock.tick
- `pygame.time.Clock.tick_busy_loop()`:
  https://www.pygame.org/docs/ref/time.html#pygame.time.Clock.tick_busy_loop
- `pygame.event.get()`:
  https://www.pygame.org/docs/ref/event.html#pygame.event.get
- `pygame.event.type`:
  https://www.pygame.org/docs/ref/event.html#pygame.event.EventType.type
- `pygame.QUIT`:
  https://www.pygame.org/docs/ref/event.html#pygame.event.EventType.type
- `pygame.Surface.fill()`:
  https://www.pygame.org/docs/ref/surface.html#pygame.Surface.fill

3 Grafikprimitive

Index

Doublebuffer, 6

event
    type, 6

Flag, 6
frames per second, 9

Hauptprogrammschleife, 6

main loop, 6

pygame
    display
        flip(), 6
        set_caption(), 6
        set_mode(), 5
    event
        get(), 6
    init(), 5
    mixer
        init(), 5
    QUIT, 6
    quit(), 7
    Surface, 6
        fill(), 7
    time
        Clock, 9

SDL_VIDEO_WINDOW_POS, 5