Objektorientiertes Python mit Pygame

Benjamin Schmid

Pygame ist eine Python-Bibliothek, um Computerspiele zu entwickeln. Wir werden ein einfaches Garten-Spiel entwickeln, um dabei sowohl Pygame als auch die objektorientierte Programmierung in Python kennen zu lernen. Die objektorientierte Programmierung ist eine Möglichkeit, den Code besser zu strukturieren.

Inhaltsverzeichnis

1 Pygame Einführung 2
2 Tilesheets 11
3 Die ersten Klassen 18
4 Gemüseanbau 24
5 Vererbung 30
6 Gameobject, Interfaces, und Ducktyping 36
7 Mögliche Erweiterungen 39

Kapitel 1: Pygame Einführung

In den nächsten paar Kapiteln werden wir ein kleines Computerspiel entwickeln. Wir verwenden hierfür Python mit Pygame. Pygame ist eine Python-Bibliothek für die Entwicklung von Computerspielen.

Wir werden zunächst ein neues Projekt in PyCharm einrichten und alle notwendigen Dateien anlegen. Anschliessend werden wir eine erste Grafik mit Pygame auf den Bildschirm zeichnen und eine erste Interaktion damit implementieren.

Da für die Verwendung von Pygame einiges an Standardcode notwendig ist, der von den wichtigen Themen ablenkt, werden wir einen ganz kleinen Wrapper um Pygame verwenden. Der Wrapper stellt Funktionalität von Pygame auf eine vereinfachte Art zur Verfügung, um die viel genutzten Funktionen einfacher nutzen zu können. Falls Sie interessiert sind, können Sie jedoch jederzeit den Code des Wrappers anschauen, um zu sehen, was dieser im Hintergrund macht.

Einige Themen werden nur benötigt, damit das Spiel besser aussieht oder mehr Spass zum Spielen macht. Für die zentralen Themen, die wir uns anschauen wollen, sind diese nicht notwendig. Diese zusätzlichen Themen sind jeweils als optional gekennzeichnet und können nach eigener Präferenz übersprungen werden. Zusätzlich finden Sie in Kapitel 7 weitere Vorschläge, wie das Spiel erweitert werden könnte.

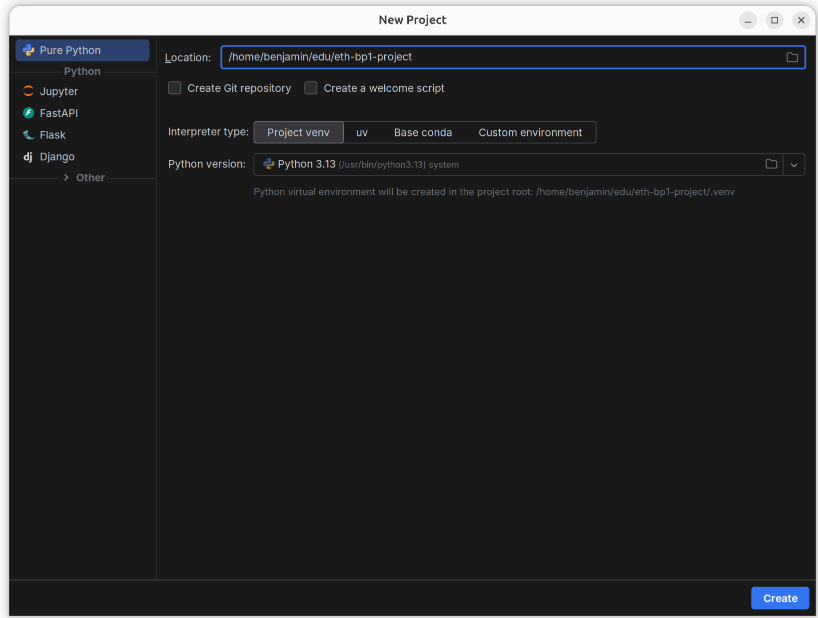
1.1. Das Spiel

Das Spiel, das wir entwickeln werden, ist ein einfaches Garten-Spiel. Auf der ersten Seite ist das fertige Spiel abgebildet. Im Spiel kümmert man sich um einen Garten einer Farm. Man kann verschiedene Pflanzen anpflanzen, die dann regelmässig bewässert werden müssen, damit sie wachsen. Schlussendlich können sie geerntet werden.

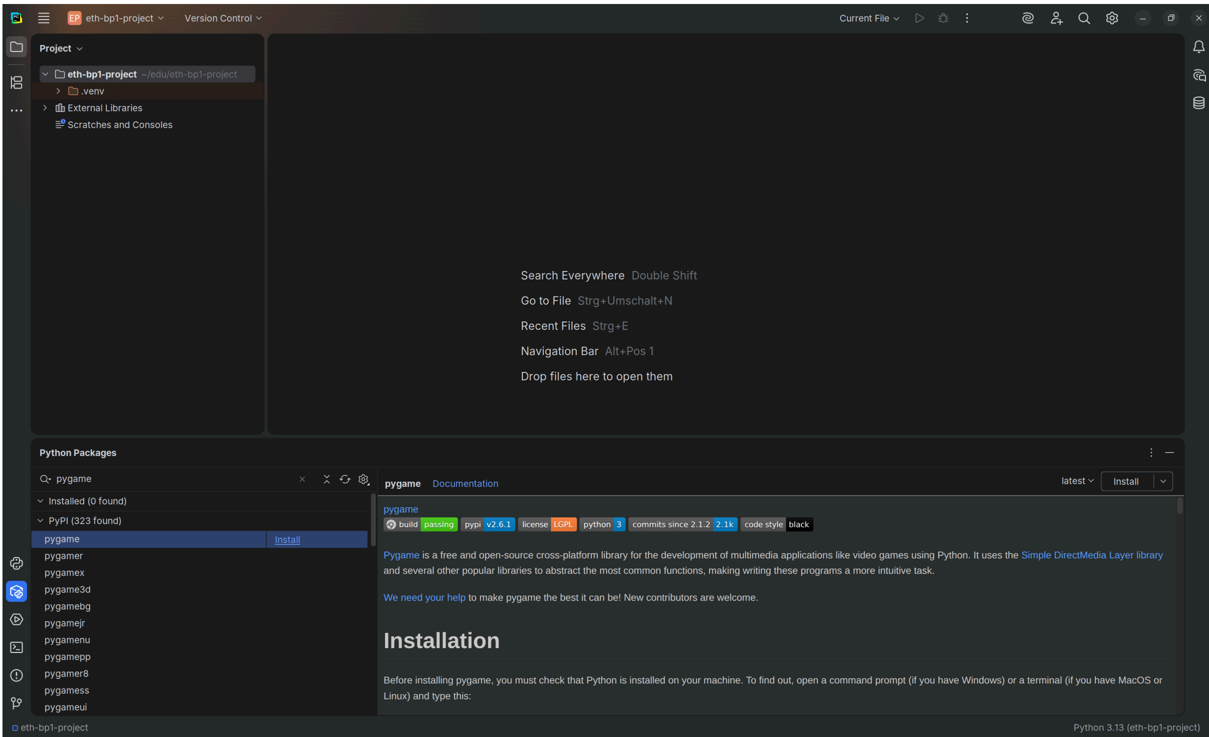
Der/die Spieler/in steuert dabei eine Figur, die auf der Farm herumlaufen kann, Pflanzen anpflanzen kann und diesen Wasser gibt.

1.2. PyCharm Projekt

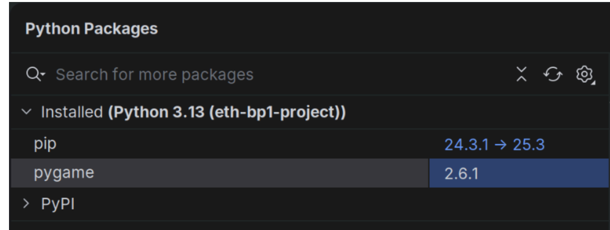
Erstellen Sie ein neues Projekt in PyCharm.



Nun müssen wir Pygame in diesem neuen Projekt installieren. Klicken Sie hierzu auf das Symbol für Python Pakete unten rechts. Suchen Sie dort nach „pygame“ und installieren Sie das Paket. Falls Sie nach der Version gefragt werden, können Sie die Neuste installieren. Zum Erstellungszeitpunkt dieses Skripts war dies 2.6.1.



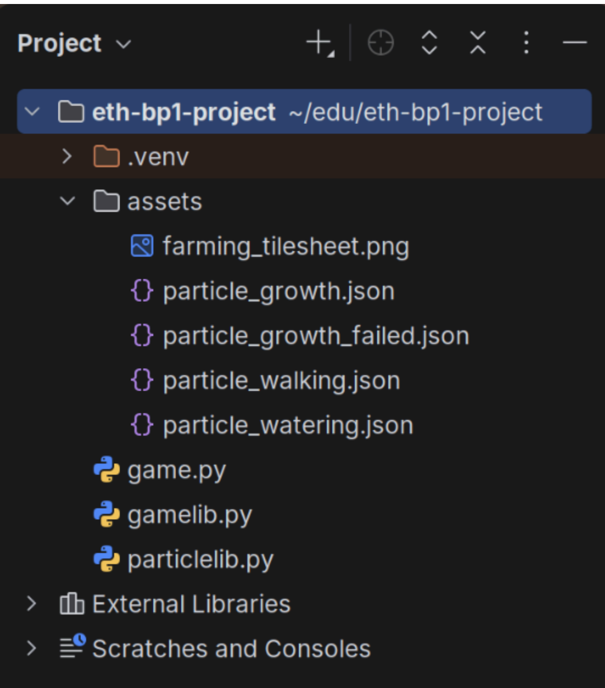
Nach der Installation sollte dies wie folgt aussehen:



Wir benötigen nun noch einige Dateien, wie unter anderem den Wrapper um Pygame und die Grafiken, die wir verwenden werden. Laden Sie die ZIP-Datei von https://smbe.xyz/pg herunter und entpacken Sie sie im Projektordner. Auf der Webseite finden Sie auch Links zur Dokumentation von Pygame.

Erstellen Sie eine neue Python-Datei namens `game.py`. Der gesamte Code, den wir erarbeiten werden, schreiben wir in dieser Datei.

Ihr PyCharm-Projekt sollte nun die folgenden Dateien enthalten:



Im Ordner `assets` sind die Grafiken abgespeichert. `gamelib.py` ist der kleine Wrapper um Pygame. Die restlichen Dateien (`particlelib.py` und die Dateien mit `.json` am Ende) werden nur für optionale Erweiterungen in Abschnitt 7.2 benötigt und können vorläufig ignoriert werden.

1.3. Einen Kreis zeichnen

Nun können wir damit beginnen, eine erste Grafik auf den Bildschirm zu bringen. Bevor wir mit komplizierten Grafiken arbeiten können, müssen wir zuerst die Grundlagen von Pygame verstehen. Wir werden daher zunächst einen Kreis zeichnen und diesen anschliessend bewegen. Sobald wir aber angeschaut haben, wie man vorgefertigte Grafiken zeichnet, können wir ganz einfach den Kreis durch unsere Spielerfigur ersetzen.

Wir beginnen mit folgendem Code in `game.py`:

python

```python
from gamelib import Game


def main():
    game = Game("OOP in Python", (1024, 800))
    while game.game_loop():
        pass


if __name__ == "__main__":
    main()
```

Wir haben hier verschiedene Bestandteile:

- In Zeile 1 importieren wir die Funktionalität aus dem Wrapper, um Pygame verwenden zu können. Wir können anschliessend `Game` in unserem Programm verwenden.
- In Zeile 4 definieren wir die Hauptfunktion, die wir in Zeilen 10 und 11 aufrufen.
- In Zeile 5 erstellen wir ein neues Spiel mithilfe des Wrappers, den wir in Zeile 1 importiert haben. Hier geben wir den Titel („OOP in Python“) sowie die Breite (1024) und Höhe (800) des Fensters mit. Sie können gerne einen eigenen Titel für Ihr Spiel wählen.
- In Zeilen 6 und 7 haben wir schlussendlich die sogenannte Spiel-Schleife.

Theorie

Um ein Spiel darzustellen, müssen wir den folgenden Ablauf immer wieder wiederholen:

1. Eingaben des Spielers einlesen,
2. Zustand des Spiels aktualisieren,
3. den aktuellen Zustand zeichnen.

Dieser Prozess wird mehrere Male pro Sekunde wiederholt, in unserem Fall ca. 60 Mal pro Sekunde. Dies nennen wir die *Spiel-Schleife* (engl. Game Loop). Jedes Zeichnen des aktuellen Zustands nennen wir ein *Frame*. Entsprechend nennt man die Anzahl Wiederholungen pro Sekunde auch „Frames pro Sekunde“ oder kurz *FPS*.

Momentan ist der Inhalt unserer Spiel-Schleife nur ein `pass`. D.h. wir lesen keine Eingaben ein und zeichnen auch nichts auf den Bildschirm.

Wenn wir das Programm jetzt ausführen, sehen wir nur einen schwarzen Bildschirm. Um dies zu ändern, zeichnen wir nun einen blauen Kreis auf den Bildschirm:

python

```python
import pygame

from gamelib import Game


def main():
    game = Game("OOP in Python", (1024, 800))
    position_x = 512
    position_y = 400
    while game.game_loop():
        pygame.draw.circle(
            game.get_screen(),
            (24, 32, 184),
            (position_x, position_y),
            20,
        )


if __name__ == "__main__":
    main()
```

Mit `pygame.draw.circle(...)` auf Zeile 11 können wir einen Kreis zeichnen. Diese Funktion braucht dabei die folgenden Argumente:

1. das Ziel, wohin gezeichnet werden soll,
2. die Farbe des Kreises,
3. die Position des Kreises,
4. der Radius des Kreises.

Als Ziel für das Zeichnen verwenden wir momentan immer den Bildschirm, den wir mit `game.get_screen()` bekommen.

Die Farbe wird in RGB mittels drei Zahlen angegeben.

Theorie

Bei *RGB* geben wir die Farbbestandteile in Rot, Grün und Blau, jeweils zwischen 0 und 255, an. Es handelt sich dabei um Lichtfarben (im Gegensatz zu Deckfarben, wenn wir z.B. Wasserfarbe mischen), die additiv gemischt werden. Das heisst: wenn wir alle Farben mischen, ergibt es Weiss. Eine Mischung aus Rot und Grün ergibt Gelb. Tabelle 1 zeigt einige Beispiele von RGB-Werten und den zugehörigen Farben.

(255, 0, 0)	(221, 191, 23)	(0, 255, 0)	(0, 0, 255)	(24, 32, 184)

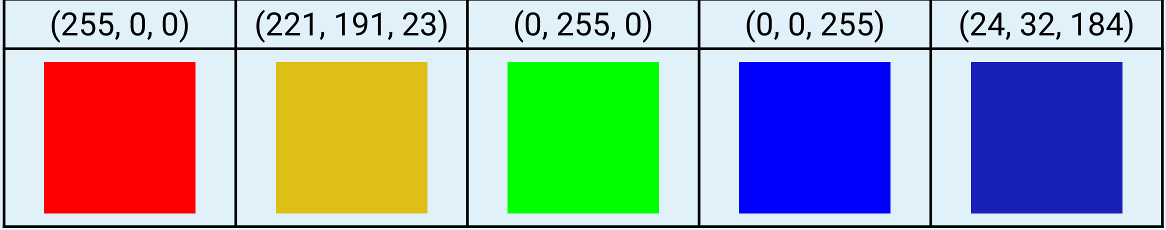
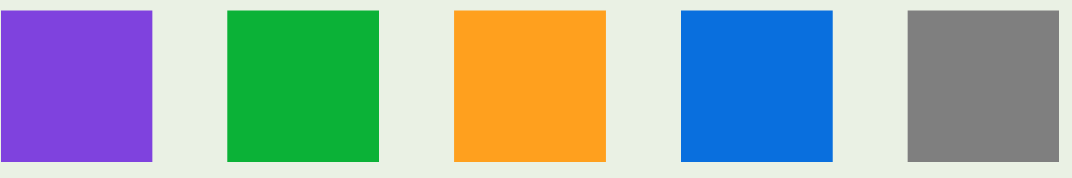


Tabelle 1: Einige Beispiele von RGB-Farben.

Übung 1

Um die RGB-Bestandteile für eine Farbe zu bestimmen, kann man sogenannte Farbwähler (engl. Color Picker) verwenden. Auf https://smbe.xyz/pg ist ein solcher verlinkt. Nutzen Sie diesen, um die ungefähren RGB-Bestandteile für die folgenden Farben zu bestimmen.



Die Position geben wir als Anzahl Pixel von der oberen linken Ecke an. Wir verwenden hier 512 Pixel auf der x-Achse (links-rechts) und 400 Pixel auf der y-Achse (oben-unten). Dies ist genau die Hälfte der Fenstergrösse von 1024 mal 800 Pixel. Achtung: Die Koordinaten verhalten sich leicht anders, als Sie sich wahrscheinlich aus der Mathematik gewohnt sind. Die y-Koordinaten werden nach unten grösser, nicht nach oben. Das heisst: oben links sind die Koordinaten 0, 0 und unten links dann z.B. 0, 800. Die x-Koordinaten werden wie gewohnt nach rechts grösser. Abbildung 6 zeigt verschiedene Punkte im Koordinatensystem sowie wo unser Kreis gezeichnet wird. Die Abbildung ist nicht massstabsgerecht.

Der Radius schlussendlich wird ebenfalls in Pixeln angegeben.

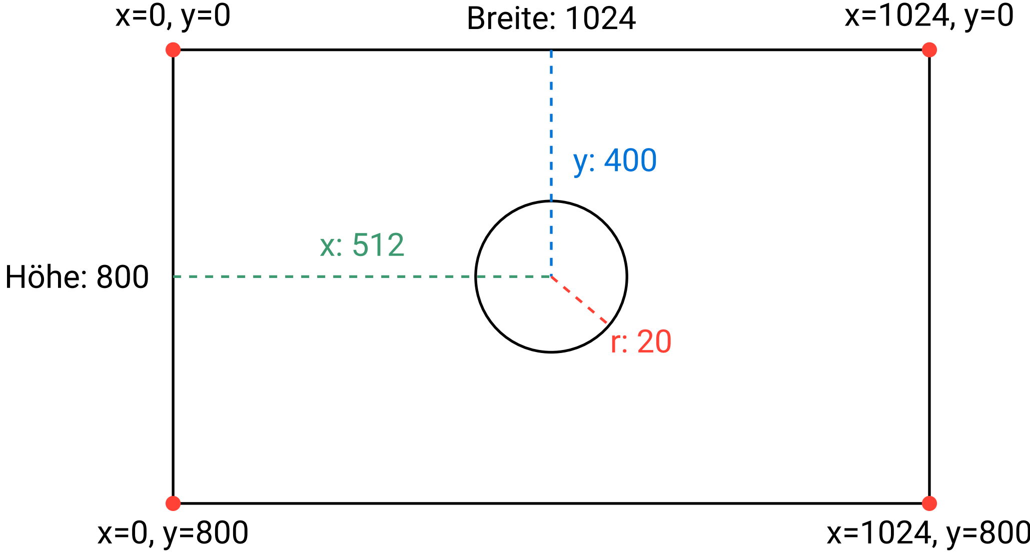
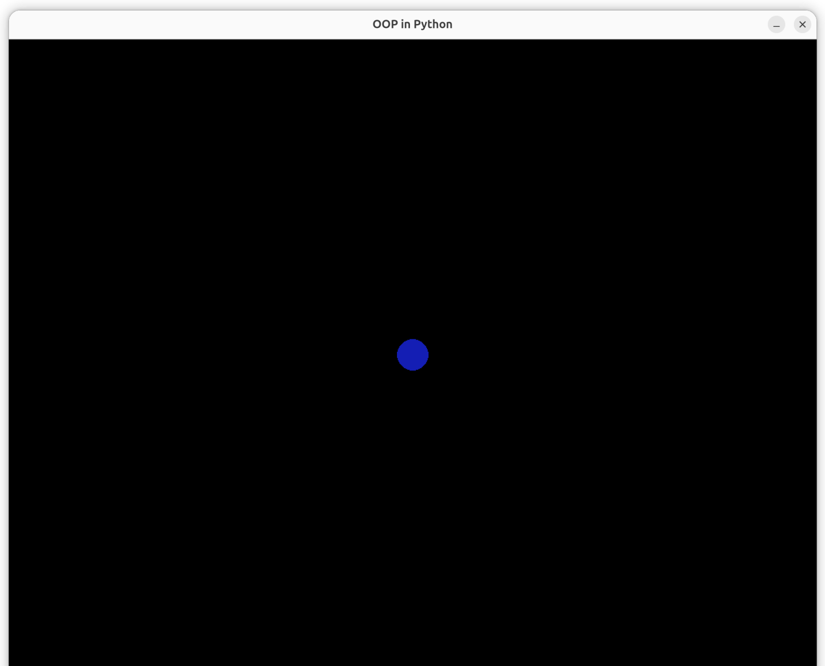


Abbildung 6: Das Koordinatensystem von Pygame.

Wenn Sie das Programm nun ausführen, sehen Sie ein schwarzes Fenster mit einem blauen Kreis in der Mitte.



Übung 2

- Ändern Sie die Farbe des Kreises zu Gelb.
- Ändern Sie die Position des Kreises, sodass dieser oben rechts ist.
- Ändern Sie die Form des Kreises zu einem Rechteck. Nutzen Sie hierfür `pygame.draw.rect(...)`. Nutzen Sie die Online-Dokumentation, um die benötigten Argumente herauszufinden.

1.4. Den Kreis bewegen

Wir können nun etwas auf den Bildschirm zeichnen, jedoch ist das Ganze noch nicht wirklich interaktiv. Pygame bietet uns die Möglichkeit, abzufragen, welche Tasten auf der Tastatur vom Spieler momentan gedrückt werden.

Mit `pygame.key.get_pressed()` erhalten wir ein Dictionary, das angibt, welche Tasten momentan gedrückt werden. Wir werden nun dem Spieler erlauben, mit den Pfeiltasten die Position des Kreises anzupassen.

Dies sieht dann wie folgt aus:

```python
import pygame

from gamelib import Game


def main():
    game = Game("OOP in Python", (1024, 800))
    position_x = 512
    position_y = 400
    while game.game_loop():
        keys = pygame.key.get_pressed()
        if keys[pygame.K_LEFT]:
            position_x -= 250 * game.get_dt()
        if keys[pygame.K_RIGHT]:
            position_x += 250 * game.get_dt()
        if keys[pygame.K_UP]:
            position_y -= 250 * game.get_dt()
        if keys[pygame.K_DOWN]:
            position_y += 250 * game.get_dt()
        position_x = pygame.math.clamp(position_x, 0, 1024)
        position_y = pygame.math.clamp(position_y, 0, 800)
        pygame.draw.circle(
            game.get_screen(),
            (24, 32, 184),
            (position_x, position_y),
            20,
        )


if __name__ == "__main__":
    main()
```

Zunächst speichern wir uns die gedrückten Tasten in `keys`. Diese können wir dann mit den Konstanten von Pygame für die verschiedenen Tasten, wie z.B. `pygame.K_LEFT` abfragen. Abhängig von den gedrückten Tasten aktualisieren wir die Position des Kreises. Als Geschwindigkeit haben wir 250 Pixel pro Sekunde gewählt.

Neu verwenden wir auch `game.get_dt()`. Diese Funktion gibt uns zurück, wie viele Sekunden seit dem letzten Frame vergangen sind. Dies stellt sicher, dass sich der Kreis immer mit der genau gleichen Geschwindigkeit fortbewegt, unabhängig davon, wie schnell der Computer ist, der das Spiel ausführt.

Zu den Anfangszeiten der Computerspiele nutzte man diese Technik noch nicht. Wenn dann schnellere Computer auf den Markt kamen, liefen die Spiele auf diesen nicht mehr korrekt, da sich alles viel zu schnell bewegte aufgrund der höheren FPS. Es ist deshalb wichtig, dass alle Berechnungen unabhängig davon sind, wie schnell die Spiel-Schleife läuft.

Wir haben auch die Funktion `pygame.match.clamp(...)` neu genutzt. Diese Funktion beschränkt den ersten Wert auf einen bestimmten Bereich, der durch den zweiten und dritten Wert gegeben ist. Hier verwenden wir diese Funktion, um zu verhindern, dass der Kreis den Bildschirm verlassen kann.

Die Bestandteile von Pygame, die wir bisher kennengelernt haben, reichen bereits aus, um ein einfaches Spiel wie z.B. Pong zu implementieren.

1.5. Zusammenfassung

Wir haben ein neues Projekt in PyCharm angelegt und für die Entwicklung mit Pygame eingerichtet. Wir verwenden Pygame nicht direkt, sondern mithilfe eines kleinen Wrappers.

Wir haben unsere ersten visuellen Ausgaben auf den Bildschirm gebracht und eine einfache Interaktion damit implementiert. Mit `pygame.draw.circle(...)` kann ein Kreis gezeichnet werden, mit `pygame.draw.rect(...)` ein Rechteck.

Mit `pygame.key.get_pressed()` können wir abfragen, welche Tasten momentan gedrückt werden.

Wir haben die Spiel-Schleife kennengelernt. Eine Iteration der Spiel-Schleife besteht aus folgenden Schritten:

1. Eingaben einlesen,
2. Zustand aktualisieren,
3. Grafik zeichnen.

Wir haben uns auch kurz angeschaut, wie Farben aufgebaut sind. Wir geben jeweils den Rot-, Grün- und Blau-Anteil an. Die Farben werden additiv gemischt im Gegensatz zu z.B. Wasserfarben.

Kapitel 2: Tilesheets

Wir können nun zwar einfache geometrische Formen zeichnen, doch damit komplexere Grafiken zu zeichnen, wäre zu kompliziert. Deshalb verwendet man normalerweise vorgefertigte Grafiken.

Theorie

Als Optimierung verwendet man oft eine einzige Grafik, die viele kleine Grafiken enthält. So muss nur eine Grafik statt vielen kleinen geladen werden. Es wird dann immer nur ein kleiner Teil davon gezeichnet. Eine solche Grafik nennt man ein *Tilesheet* und die kleinen Teilgrafiken nennt man *Tiles* (deutsch *Kachel*).

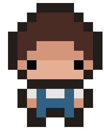
Für unser Spiel werden wir folgendes Tilesheet verwenden:

Die einzelnen Teilgrafiken, die z.B. eine Figur oder ein Gemüse zeigen, sind die Tiles. Unser Tilesheet besteht aus insgesamt 169 Tiles, aufgeteilt in 13 Spalten und 13 Zeilen.

Das Tilesheet ist von hier: https://josie-makes-stuff.itch.io/pixel-art-farming-assets.

2.1. Figur zeichnen

Der Wrapper lädt das Tilesheet und stellt eine Methode bereit, um eines der Tiles an einem bestimmten Ort zu zeichnen. Dabei geben wir die Koordinaten des Tiles als Spalte und Zeile an. Wie bereits beim Zeichnen des Kreises werden die Koordinaten gegen unten und nach rechts grösser. Wir werden nun die Figur an Position 0,0 (ganz oben links) statt einem Kreis verwenden.



Der angepasste Code sieht wie folgt aus:

python

```python
import pygame

from gamelib import Game


def main():
    game = Game("OOP in Python", (1024, 800))
    position_x = 512
    position_y = 400
    while game.game_loop():
        keys = pygame.key.get_pressed()
        if keys[pygame.K_LEFT]:
            position_x -= 250 * game.get_dt()
        if keys[pygame.K_RIGHT]:
            position_x += 250 * game.get_dt()
        if keys[pygame.K_UP]:
            position_y -= 250 * game.get_dt()
        if keys[pygame.K_DOWN]:
            position_y += 250 * game.get_dt()
        position_x = pygame.math.clamp(position_x, 0, 1024)
        position_y = pygame.math.clamp(position_y, 0, 800)
        game.draw_tile((0, 0), (position_x, position_y))


if __name__ == "__main__":
    main()
```

Wir geben der Methode `game.draw_tile(...)` zuerst die Koordinaten des Tiles an und dann die Position, wo die Grafik gezeichnet werden soll.

Wir haben nun eine Figur, die wir auf einem schwarzen Hintergrund bewegen können. Es gibt aber noch Verbesserungspotenzial. So ist es z.B. möglich, die Figur auf der rechten Seite und unten aus dem Bildschirm zu bewegen. Ausserdem wäre es schön, wenn wir einen interessanteren Hintergrund hätten.

2.2. Refactoring

Wir werden den Code etwas umzustellen, um zu verhindern, dass die Figur aus dem Bildschirm bewegt werden kann. Hierfür erstellen wir eine Konstante mit der Grösse des Fensters. Ausserdem können wir eine Konstante aus dem Wrapper verwenden, die angibt, wie gross ein Tile ist.

Abbildung 10 zeigt, wie wir die Position begrenzen müssen. Für die Position der Grafik für die Figur geben wir den oberen linken Ecken an. Deshalb können wir die Position links und oben weiterhin auf 0 beschränken. Auf der rechten Seite und unten müssen wir jeweils die Grösse eines Tiles von der Grösse des Fensters abziehen. So ist die untere rechte Ecke genau in der unteren rechten Ecke des Fensters.

Im Beispiel in Abbildung 10 nehmen wir an, dass die Figur 100 Pixel gross ist. Das Fenster ist wie bisher 1024 Pixel breit und 800 Pixel hoch. Wir müssen daher die Position auf 924 nach rechts beschränken und auf 700 nach unten.

Achtung: In unserem Spiel hat die Figur eine andere Grösse. Wir müssen die Konstante `TILE_SIZE` verwenden, damit wir immer die korrekte Grösse der Figur verwenden.

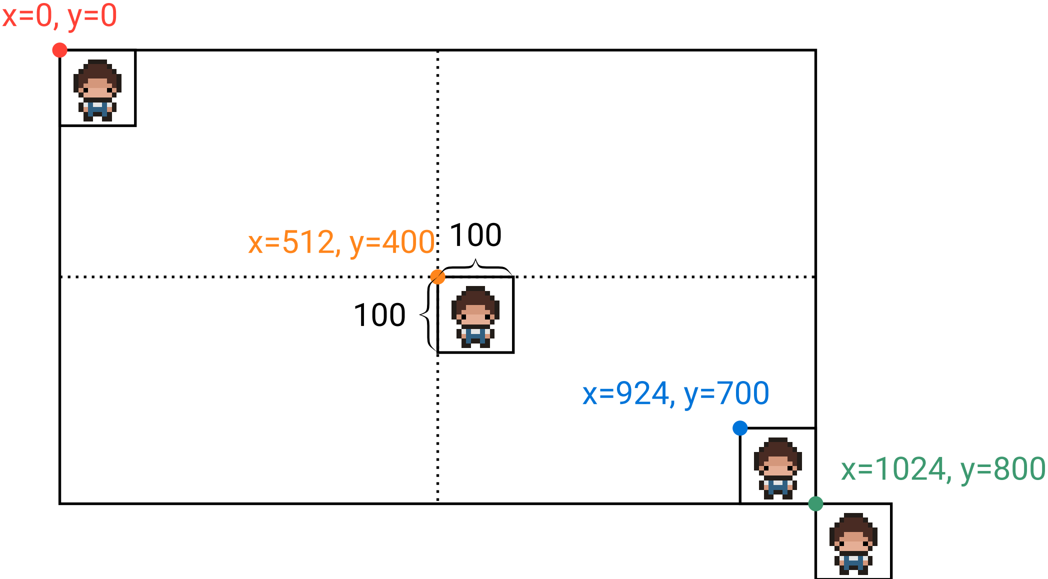


Abbildung 10: Verschiedene Positionen der Spieler-Figur.

Um die Startposition der Figur zu berechnen, müssen wir ebenfalls berücksichtigen, dass wir jeweils die obere linke Ecke angeben. Wenn wir die Grösse des Fensters halbieren (gestrichelt in Abbildung 10), müssen wir anschliessend noch die halbe Grösse der Figur abziehen, damit die Mitte der Figur genau in der Mitte des Fensters ist.

Der neue Code sieht wie folgt aus:

```python
import pygame

from gamelib import Game, TILE_SIZE

WINDOW_SIZE = (1024, 800)


def main():
    game = Game("OOP in Python", WINDOW_SIZE)
    position_x = (WINDOW_SIZE[0] - TILE_SIZE) // 2
    position_y = (WINDOW_SIZE[1] - TILE_SIZE) // 2
    while game.game_loop():
        keys = pygame.key.get_pressed()
        if keys[pygame.K_LEFT]:
            position_x -= 250 * game.get_dt()
        if keys[pygame.K_RIGHT]:
            position_x += 250 * game.get_dt()
        if keys[pygame.K_UP]:
```

```python
                position_y -= 250 * game.get_dt()
            if keys[pygame.K_DOWN]:
                position_y += 250 * game.get_dt()
            position_x = pygame.math.clamp(position_x, 0, WINDOW_SIZE[0] - TILE_SIZE)
            position_y = pygame.math.clamp(position_y, 0, WINDOW_SIZE[1] - TILE_SIZE)
            game.draw_tile((0, 0), (position_x, position_y))


if __name__ == "__main__":
    main()
```

Optional

Die Figur bewegt sich schneller, wenn man diagonal läuft, als wenn man horizontal oder vertikal läuft. Der Grund dafür ist, dass die horizontale und vertikale Geschwindigkeit momentan einfach addiert werden. Passen Sie die Berechnung der neuen Position so an, dass die Figur auch diagonal die gleiche Geschwindigkeit hat.

2.3. Hintergrund zeichnen

Als nächstes möchten wir nun die Wiese unserer Farm zeichnen. Später werden einzelne Quadrate durch ein Beet mit den wachsenden Pflanzen ersetzt. Aber vorläufig zeichnen wir für alle Quadrate eine Wiese. Auf dem Tilesheet gibt es verschiedene Tiles, die wir verwenden können. Wir werden das grüne Quadrat verwenden.

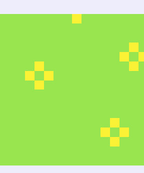
Um die Koordinaten der gewünschten Tiles zu bestimmen, können Sie die Taste ‚P' gedrückt halten, während das Spiel läuft. Der Wrapper wird dann das Tilesheet mit den Koordinaten zeichnen.

Sie werden sicher festgestellt haben, dass neben dem Tilesheet von oben auch noch die gleichen Tiles jeweils horizontal und vertikal gespiegelt vorhanden sind. Dies ist eine weitere Optimierung, da viele Tiles eine Spiegelsymmetrie haben und somit nur einmal in der Grafik gespeichert werden müssen. Zum Beispiel sieht ein Huhn, das nach links oder nach rechts schaut, genau gleich aus bis auf die Spiegelung. Der Wrapper erstellt automatisch die gespiegelte Version des Tilesheets, wenn die Grafik geladen wird.

Mit der Übersicht haben wir nun die Koordinaten bestimmt: 12,5 für das grüne Quadrat.

Optional

Zusätzlich zum grünen Quadrat (12,5) könnten Sie noch das Quadrat mit den gelben Blumen (1,7) verwenden. Dann müssen Sie auf eine interessante Weise entscheiden, welches der beiden Tiles Sie an jeder Position anzeigen möchten.



Das Zeichnen sieht dann wie folgt aus:

python

```python
import pygame

from gamelib import Game, TILE_SIZE

WINDOW_SIZE = (1024, 800)


def main():
    game = Game("OOP in Python", WINDOW_SIZE)
    position_x = (WINDOW_SIZE[0] - TILE_SIZE) // 2
    position_y = (WINDOW_SIZE[1] - TILE_SIZE) // 2
    while game.game_loop():
        keys = pygame.key.get_pressed()
        if keys[pygame.K_LEFT]:
            position_x -= 250 * game.get_dt()
        if keys[pygame.K_RIGHT]:
            position_x += 250 * game.get_dt()
        if keys[pygame.K_UP]:
            position_y -= 250 * game.get_dt()
```

```
20         if keys[pygame.K_DOWN]:
21             position_y += 250 * game.get_dt()
22         position_x = pygame.math.clamp(position_x, 0, WINDOW_SIZE[0] -
   TILE_SIZE)
23         position_y = pygame.math.clamp(position_y, 0, WINDOW_SIZE[1] -
   TILE_SIZE)
24
25         for y in range(WINDOW_SIZE[1] // TILE_SIZE + 1):
26             for x in range(WINDOW_SIZE[0] // TILE_SIZE + 1):
27                 game.draw_tile((12, 5), (x * TILE_SIZE, y * TILE_SIZE))
28         game.draw_tile((0, 0), (position_x, position_y))
29
30
31 if __name__ == "__main__":
32     main()
```

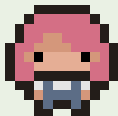
Da das grüne Quadrat nicht gross genug ist, um den gesamten Hintergrund abzudecken, müssen wir es mehrfach zeichnen. Das Quadrat hat eine Breite von `TILE_SIZE`, d.h. wir müssen es mindestens `WINDOW_SIZE[0] // TILE_SIZE` oft horizontal zeichnen und analog für die vertikale Richtung. Dabei kann jedoch noch ein kleiner Rand verbleiben, weshalb wir noch `1` hinzuzählen müssen.

Bei `game.draw_tile(...)` müssen wir die Position in Pixeln angeben. Jedes neue Quadrat muss deshalb um `TILE_SIZE` versetzt sein, weshalb wir `x * TILE_SIZE` (bzw. mit `y`) für die Position verwenden.

Es ist dabei wichtig, in welcher Reihenfolge wir die Tiles zeichnen. Zuerst müssen wir die Wiese im Hintergrund zeichnen und erst anschliessend können wir die Figur zeichnen. Hätten wir zuerst die Figur gezeichnet und dann die Wiese, wäre die Figur übermalt worden und wäre nicht mehr sichtbar.

Übung 3

Es gibt neben der bisher verwendeten Grafik für die Figur auch noch eine weibliche Version. Passen Sie den Code so an, dass diese Version verwendet wird.



Optional

Sie können das Aussehen der Figur ganz individuell anpassen, indem Sie die Grafik im Tilesheet bearbeiten. Stellen Sie dabei sicher, dass Sie die Grösse nicht verändern und die Datei wieder im selben Format (PNG) abspeichern. Damit können Sie später z.B. auch Ihre eigenen Pflanzen hinzufügen.

Optional

In Abschnitt 7.1 ist beschrieben, wie man einfache Animationen hinzufügen kann. Damit können bewegte Grafiken angezeigt werden, während sich die Figur bewegt.

2.4. Zusammenfassung

Unser Spiel sieht nun in etwa so aus:

Wir haben uns angeschaut, wie ein Tilesheet aufgebaut ist und wie wir mit dem Wrapper ein Tile zeichnen können (`game.draw_tile(...)`). Damit haben wir das Spiel weiterentwickelt, sodass eine Spielergrafik angezeigt wird und im Hintergrund Gras sichtbar ist.

Kapitel 3: Die ersten Klassen

Wir haben in unserem Spiel nun zwei unterschiedliche Konzepte, die wir verwalten: den Spieler und die Farm. Wir werden nun eine Möglichkeit kennenlernen, wie diese Konzepte im Code getrennt werden können und separat implementiert werden können. In späteren Kapiteln werden wir dann auch sehen, wie diese neue Möglichkeit verwendet werden kann, um verschiedene ähnliche, verwandte Konzepte einfacher zu implementieren.

3.1. Klassen und Objekte

Theorie

Eine *Klasse* beschreibt ein bestimmtes Konzept. Sie fasst den Zustand und die verfügbaren Methoden zusammen. Ein *Objekt* ist eine konkrete Instanz einer Klasse. Für jedes Objekt hat der Zustand einen individuellen Wert, der durch die Methoden verändert werden kann. Die einzelnen Variablen des Zustandes nennt man *Felder* oder *Members*.

Methoden sind sehr ähnlich wie *Funktionen*. Der Unterschied ist, dass Methoden immer im Kontext einer Klasse definiert werden und auf einem konkreten Objekt aufgerufen werden. Funktionen sind unabhängig von Klassen.

Ein praktisches Beispiel ist z.B. ein Hund. Für das Konzept eines Hundes würden wir eine Klasse erstellen. Ein Hund hat verschiedene Zustände, wie z.B. sein Name und wie viel Energie er noch hat. Ein Hund hat auch verschiedene Methoden, wie man mit ihm interagieren kann. Z.B. kann man den Hund füttern oder mit ihm Laufen gehen. Ersteres wird das Energieniveau erhöhen, während Zweiteres das Energieniveau senkt. Konkrete Instanzen der Klasse „Hund“ wären dann z.B. der Golden Retriever „Chipsi“ und der Dackel „Bello“. Die beiden Hunde werden dann durch Objekte dargestellt, die Instanzen der Klasse „Hund“ sind. Die beiden Hunde haben jeweils ihren eigenen Namen und ihr eigenes Energieniveau. Aber sie haben beide die gleichen Methoden, wie man mit ihnen interagieren kann.

Theorie

Eine Klasse hat eine spezielle Methode, welche ein neues Objekt der Klasse erstellt. Diese nennt man den *Konstruktor* der Klasse. Dem Konstruktor kann man Argumente mitgeben, die bei der Erstellung des Objekts verwendet werden. Im Konstruktor werden alle Member initialisiert.

In unserem Beispiel der Hunde würde man den Namen im Konstruktor mitgeben. Weiter würden wir im Konstruktor das Energieniveau auf einen Startwert gesetzt.

In Python implementiert sieht dies dann wie folgt aus. Achtung: Dies ist ein separates Beispiel, das nichts mit unserem Spiel direkt zu tun hat.

python

```python
class Dog:
    def __init__(self, name):
        self.name = name
        self.energy = 0

    def feed(self):
        print("Feed", self.name)
        self.energy = 50

    def walk(self):
        if self.energy < 20:
            print(self.name, "is too tired for a walk")
        else:
            print("Go for a walk with", self.name)
            self.energy -= 20
```

In Python beginnen wir eine Klasse mit `class ClassName:`, also z.B. `class Dog:`. Anschliessend definieren wir alle Methoden für die Klasse. Alle Methoden haben als ersten Parameter `self`, der das aktuelle Objekt enthält. Auf diesem Objekt setzen wir die Members (Zustandsvariablen).

Den Konstruktor schreiben wir als `def __init__(self):`, allenfalls noch mit weiteren Parametern. Wie bei normalen Methoden enthält der Parameter `self` das aktuelle Objekt. In diesem Fall wird dieses gerade konstruiert.

Wenn wir Parameter im Konstruktor mitgeben möchten, können wir dies wie bei normalen Methoden hinzufügen, z.B. `def __init__(self, name):`. Wenn wir ein Objekt erstellen, können wir dann das entsprechende Argument mitgeben: `Dog("Bello")`.

Der Zustand des Hundes speichern wir in den Membern `self.name` und `self.energy`. Der Name wird mit dem Parameter initialisiert, während die Energie auf einen Standardwert gesetzt wird.

Weiter definieren wir die beiden Methoden `feed` und `walk`. Beide verändern das Energieniveau und geben eine Nachricht auf die Konsole aus, damit wir sehen, was passiert. Die Methoden sehen fast gleich aus wie normale Funktionen, mit dem Unterschied, dass sie innerhalb einer Klasse definiert werden und als ersten Parameter `self` haben.

Wenn wir die Klasse verwenden möchten, sieht das z.B. wie folgt aus:

python

```python
if __name__ == "__main__":
    dog1 = Dog("Chipsi")
    dog2 = Dog("Bello")

    dog1.feed()
```

```
23      dog2.walk()
24      dog1.walk()
25      print("---")
26
27      dog2.feed()
28      dog2.walk()
29      print("---")
30
31      dog1.walk()
32      dog1.walk()
33      dog2.walk()
```

Um ein Objekt einer bestimmten Klasse zu erstellen, rufen wir `ClassName()` auf, also z.B. `Dog("Chipsi")`. Dies erstellt dann ein neues Objekt, welches dann den Methoden jeweils als `self` mitgegeben wird, und ruft den Konstruktor auf.

Wir erstellen zwei Objekte für unsere Klasse: `dog1` ist ein Objekt, das den Hund Chipsi darstellt, und `dog2` ist ein Objekt, das den Hund Bello darstellt. Wir geben jeweils den Namen als Argument für den Konstruktor an.

Anschliessend rufen wir die beiden Methoden auf unseren zwei Objekten auf. Wir sehen folgende Ausgabe:

```
 1 Feed Chipsi
 2 Bello is too tired for a walk
 3 Go for a walk with Chipsi
 4 ---
 5 Feed Bello
 6 Go for a walk with Bello
 7 ---
 8 Go for a walk with Chipsi
 9 Chipsi is too tired for a walk
10 Go for a walk with Bello
```

Hier sehen wir gut, dass die Zustände der beiden Objekte unabhängig voneinander sind. Wir füttern zunächst Chipsi. Dies erhöht das Energieniveau von Chipsi und wir können mit ihm spazieren gehen, doch Bello ist noch immer zu müde für einen Spaziergang. Sobald wir auch Bello gefüttert haben, können wir mit ihm spazieren gehen.

Am Ende sind wir so oft mit Chipsi laufen gegangen, dass er müde ist. Doch sein Energieniveau ist unabhängig von Bello und wir können mit Bello weiterhin laufen gehen.

Übung 4

Schreiben Sie eine Klasse, um eine Katze darzustellen. Die Katze hat einen Namen und eine Fellfarbe. Die Katze kann auf Mäusejagd gehen. Jedes dritte Mal, dass eine Katze auf Mäusejagd geht, wird sie auch eine Maus fangen.

Erstellen Sie mindestens zwei Objekte der Klasse und schicken Sie die Katzen auf Mäusejagd. Geben Sie jeweils auf der Konsole aus, was passiert, inklusive dem Namen und Fellfarbe der beteiligten Katze.

Wie bereits das Beispiel mit dem Hund werden wir diese Klasse nicht für unser Spiel verwenden.

Bei unserem Spiel haben wir bereits mit Klassen gearbeitet, ohne dass es aufgefallen ist: `Game` aus dem Wrapper ist eine Klasse und wir haben eine Objektinstanz `game` davon erstellt. Als Konstruktor-Argumente haben wir den Fensternamen und die Fenstergrösse mitgegeben.

Diese Art des Programmierens, bei der wir verschiedene Klassen definieren und viel mit Objekten arbeiten, nennt man *objektorientiertes Programmieren*.

3.2. Refactoring

Wir werden nun den Code für unser Spiel etwas umstellen und Klassen verwenden. Wir möchten die beiden voneinander unabhängigen Konzepte des Spielers und der Farm voneinander trennen und jeweils in separaten Klassen implementieren.

Für beide Klassen benötigen wir eine Methode `def draw(self):` welche den aktuellen Zustand zeichnet. Der Spieler muss zusätzlich noch eine `def update(self):` Methode haben, da wir die Eingaben einlesen müssen. Wir könnten dies theoretisch auch in `update` machen, doch dann würden wir verschiedene Themen mischen und der Code wäre weniger übersichtlich.

Da wir in den Methoden Zugriff auf das `game` Objekt benötigen, aber nicht immer als Argument mitgeben möchten, werden wir das im Konstruktor mitgeben und als Member speichern.

Wir beginnen nochmals von einer neuen, leeren Datei und werden anschliessend die bisherige Funktionalität unseres Spiels übernehmen. Das Grundgerüst unserer Klassen sieht wie folgt aus:

python

```python
1 from gamelib import Game
2
3 WINDOW_SIZE = (1024, 800)
4
5
6 class Farm:
```

```
7
8       def __init__(self, game):
9           pass
10
11      def draw(self):
12          pass
13
14
15  class Player:
16
17      def __init__(self, game):
18          pass
19
20      def update(self):
21          pass
22
23      def draw(self):
24          pass
25
26
27  def main():
28      game = Game("OOP in Python", WINDOW_SIZE)
29      farm = Farm(game)
30      player = Player(game)
31      while game.game_loop():
32          player.update()
33
34          farm.draw()
35          player.draw()
36
37
38  if __name__ == "__main__":
39      main()
```

Vor der Spiel-Schleife erstellen wir jeweils ein Objekt für die Farm und für den Spieler. In der Spiel-Schleife haben wir den gesamten Code durch die Aufrufe von `update` und `draw` auf den beiden Objekten ersetzt.

Übung 5

Befüllen Sie alle Methoden der beiden Klassen mit dem korrekten Code. Der Grossteil des Codes können Sie aus der Spiel-Schleife Ihres bisherigen Programms übernehmen. Ersetzen Sie dafür jeweils `pass` in der Vorlage oben mit dem benötigten Code aus Ihrem alten Programm.

Bei der Farm sollten Sie die Anzahl der Quadrate, die wir zeichnen sollen, im Zustand der Klasse speichern und nicht bei jedem Zeichnen neu berechnen. Dies wird die weiteren Erweiterungen erleichtern.

Durch die Aufteilung unseres Spiels in verschiedene Klassen konnten wir die Strukturierung unseres Codes verbessern. Alles, was zusammen gehört, ist nun schön beieinander. Wenn wir z.B. die Darstellung der Spielerfigur verändern möchten, wissen wir genau, dass dies in der `Player` Klasse gemacht werden muss.

Die Aufteilung in Klassen vereinfacht auch den Fall, wenn wir etwas mehrfach benötigen. Wir könnten nun z.B. mit relativ wenig Aufwand eine zweite Spielerfigur hinzufügen, die von einer zweiten Person gesteuert wird. In Kapitel 4 werden wir ein konkretes Beispiel sehen, wie das aussieht, wenn wir mehrere Objekte einer Klasse erstellen. In Kapitel 5 werden wir uns noch einen weiteren Vorteil von Klassen anschauen, mit dem wir unseren Code vereinfachen können.

3.3. Zusammenfassung

Wir haben Klassen und Objekte kennengelernt, die Grundlagen für die objektorientierte Programmierung. Klassen bestehen aus einem Zustand (Felder oder Members genannt) und Methoden, um den Zustand zu manipulieren. Ein Objekt ist eine konkrete Instanz einer Klasse. In einem Objekt hat jeder Member einen bestimmten Wert. In verschiedenen Objekten derselben Klasse kann der Zustand verschiedene Werte haben.

Klassen haben eine spezielle Methode, welche das Objekt initialisiert, welche wir Konstruktor nennen. In Python heisst diese Methode `__init__`.

In Python haben alle Methoden in Klassen als ersten Parameter `self`, mit welchem auf den Zustand zugegriffen werden kann.

Kapitel 4: Gemüseanbau

Bisher können wir nur auf der Farm herumlaufen, aber noch nichts anpflanzen. Dies wollen wir nun ändern.

4.1. Referenzen

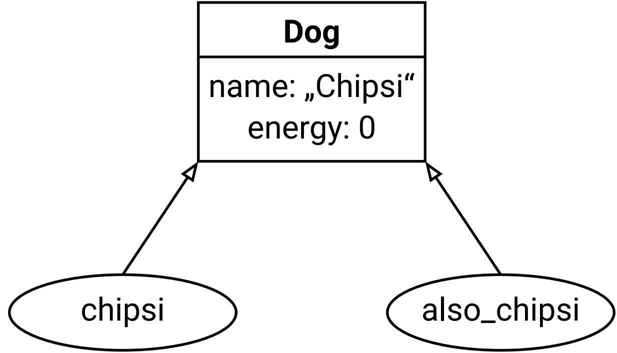
Zunächst benötigen wir jedoch noch etwas Theorie zu Objekten.

Theorie

Eine *Referenz* ist ein „Zeiger" auf ein Objekt. Ein bestimmtes Objekt existiert nur einmal, aber es können viele Referenzen auf das Objekt existieren. Diese verweisen alle auf dasselbe Objekt. In Python sind Variablen, die auf ein Objekt verweisen, immer Referenzen.

Im Beispiel der Hunde könnten wir folgendes schreiben:

```python
18 if __name__ == "__main__":
19     chipsi = Dog("Chipsi")
20     also_chipsi = chipsi
21 
22     chipsi.feed()
23     chipsi.walk()
24     also_chipsi.walk()
25     chipsi.walk()
26     also_chipsi.feed()
27     chipsi.walk()
```



Wir haben zwei Variablen, die aber beide auf dasselbe Objekt verweisen. Wenn wir `also_chipsi` verwenden, sehen wir den Effekt anschliessend auch mit `chipsi`. Wir erhalten folgende Ausgabe auf der Konsole:

```
1 Feed Chipsi
2 Go for a walk with Chipsi
3 Go for a walk with Chipsi
4 Chipsi is too tired for a walk
5 Feed Chipsi
6 Go for a walk with Chipsi
```

Um Pflanzen anpflanzen zu können, muss der Spieler mit der Farm interagieren können. Hierfür übergeben wir eine Referenz auf die Farm im Konstruktor des Spielers, indem wir wir einen weiteren Parameter `farm` für die Methode `__init__` hinzufügen. Wenn wir das Objekt für den Spieler erstellen, können wir dann auch ein weiteres Argument mit der Referenz auf die Farm mitgeben. Dies sieht dann wie folgt aus:

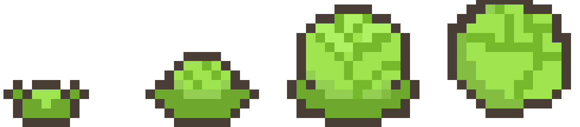
```python
79 class Player:
80
81     def __init__(self, game, farm):
82         self.game = game
83         self.farm = farm
```

```python
126 def main():
127     game = Game("OOP in Python", WINDOW_SIZE)
128     farm = Farm(game)
129     player = Player(game, farm)
```

Da Python immer Referenzen verwendet, können wir nun aus der Spielerklasse auf dasselbe Farm-Objekt zugreifen wie in der Spiel-Schleife. Referenzen haben wir bereits genutzt, als wir der Farm und dem Spieler eine Referenz auf `game` im Konstruktor übergeben haben.

4.2. Salat

Wir möchten nun Salat anpflanzen können. Im Tilesheet haben wir verschiedene Wachstumsstadien für verschiedene Pflanzen, unter anderem Salat. Wir möchten, dass der Spieler der wachsenden Pflanze Wasser geben kann und diese dann jedes Mal zum nächsten Wachstumsstadium weitergeht. Der Salat ist ein weiteres Konzept, das in sich mehr oder weniger geschlossen ist, das wir deshalb gerne in einer Klasse abstrahieren möchten.



Übung 6

Schreiben Sie eine Klasse `class Salad:` mit folgenden Methoden:

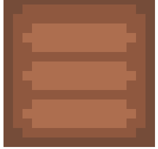
- `def water(self):` wechselt zum nächsten Wachstumsstadium, falls vorhanden.
- `def draw(self):` zeichnet den aktuellen Wachstumsstadium.

Im Konstruktor benötigt die Klasse eine Referenz auf `game` und die Position, wo die Pflanze gezeichnet werden soll:

- `def __init__(self, game, position):`

Als nächstes müssen wir die Farm-Klasse anpassen, damit wir die Pflanzen speichern und zeichnen können. Hierfür fügen wir einen Member `plants` hinzu, der eine zweidimensionale Liste mit den Pflanzen oder `None` ist. `None` wird in Python verwendet, um die Absenz eines Werts darzustellen. Wir sagen also, dass wir zu Beginn keine Pflanzen-Objekte haben.

Beim Zeichnen verwenden wir einen anderen Hintergrund, falls auf diesem Feld eine Pflanze existiert.



```python
42 class Farm:
43
44     def __init__(self, game):
45         self.game = game
46         self.farm_size = (
47             WINDOW_SIZE[0] // TILE_SIZE + 1,
48             WINDOW_SIZE[1] // TILE_SIZE + 1
49         )
50         self.plants = [[None for x in range(self.farm_size[0])] for y in range(self.farm_size[1])]
51
52     def draw(self):
53         for y in range(self.farm_size[1]):
54             for x in range(self.farm_size[0]):
55                 if self.plants[y][x] is not None:
56                     self.game.draw_tile((4, 7), (x * TILE_SIZE, y * TILE_SIZE))
57                     self.plants[y][x].draw()
58                 else:
59                     self.game.draw_tile((12, 5), (x * TILE_SIZE, y * TILE_SIZE))
```

Weiter benötigen wir eine Möglichkeit, einen Salat pflanzen zu können und Pflanzen zu wässern. Hierfür fügen wir zwei weitere Methoden zur Farm hinzu:

python
```
61     def water(self, x, y):
62         if self.plants[y][x] is not None:
63             self.plants[y][x].water()
```

python
```
68     def plant_salad(self, x, y):
69         if self.plants[y][x] is None:
70             new_salad = Salad(self.game, (x * TILE_SIZE, y * TILE_SIZE))
71             self.plants[y][x] = new_salad
```

Übung 7

Fügen Sie eine weitere Methode `remove_plant(self, x, y)` hinzu, um eine Pflanze wieder entfernen zu können. Hierfür können Sie den entsprechenden Wert in `plants` auf `None` setzen.

Optional

In Abschnitt 7.5 gibt es einen Vorschlag, wie man die fertig gewachsenen Pflanzen ernten kann statt nur zu entfernen.

Schlussendlich müssen wir noch den Spieler anpassen, damit wir die neuen Methoden auch aufrufen können. Als erstes erstellen wir eine Hilfsmethode, die uns die Koordinaten im Bezug auf die Pflanzenliste des Feldes zurückgibt, auf dem der Spieler momentan steht.

python
```
117     def current_farmplot(self):
118         x = int((self.position[0] + TILE_SIZE / 2) / TILE_SIZE)
119         y = int((self.position[1] + TILE_SIZE / 2) / TILE_SIZE)
120         return (x, y)
```

Wenn die Taste ‚2‘ gedrückt wird, soll ein Salat gepflanzt werden. Mit ‚E‘ kann man Wasser geben und mit ‚Q‘ kann man die Pflanze wieder entfernen.

Einen Teil davon kann man wie folgt implementieren:

python
```
87     def update(self):
```

python
```
105         x, y = self.current_farmplot()
```

python
```
108         if keys[pygame.K_2]:
109             self.farm.plant_salad(x, y)
```

python

```
112         if keys[pygame.K_q]:
113             self.farm.remove_plant(x, y)
```

Übung 8

Implementieren Sie die noch fehlende Taste.

Wenn wir das Wässern der Pflanzen genau gleich implementiert haben wie z.B. das Entfernen der Pflanzen, haben wir ein Problem. Für jeden Frame prüfen wir, ob die Taste momentan gedrückt wird. Falls dies der Fall ist, wässern wir die Pflanze, welche zum nächsten Wachstumsstadium wechselt. Dies bedeutet aber, dass die Pflanze sehr schnell wächst, wenn man die Taste gedrückt hält.

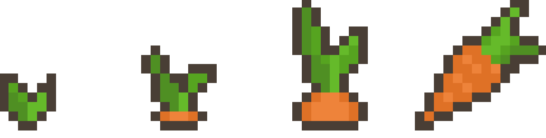
Wir wollen aber, dass die Pflanze pro Tastendruck nur um ein Stadium wächst. Hierfür müssen wir uns merken, ob die Taste im letzten Frame gedrückt war oder nicht. Dann wässern wir die Pflanze nur, wenn die Taste neu gedrückt ist. In Kapitel 6 wird noch eine Alternative hierzu beschrieben.

Übung 9

- Fügen Sie einen weiteren Member `last_keys` in der Spieler-Klasse hinzu.
- Am Ende der `update` Methode setzen Sie den Member auf das Dictionary mit den gedrückten Tasten des aktuellen Frames: `self.last_keys = keys`.
- Wenn die Taste ‚E' gedrückt ist, müssen Sie zusätzlich prüfen, ob die Taste im letzten Frame nicht gedrückt war. Nur dann soll die Pflanze bewässert werden.

4.3. Karotten

Nun wollen wir auch noch Karotten anpflanzen können. Diese verhalten sich sehr ähnlich wie der Salat, jedoch wechseln sie beim Bewässern nur mit 50% Wahrscheinlichkeit in das nächste Wachstumsstadium. Karotten sollen mit der Taste ‚1' gepflanzt werden können.



In Python kann man mit `random.random()` eine Zufallszahl zwischen 0 und 1 erzeugen. Hierfür muss man `import random` am Anfang des Programms einfügen.

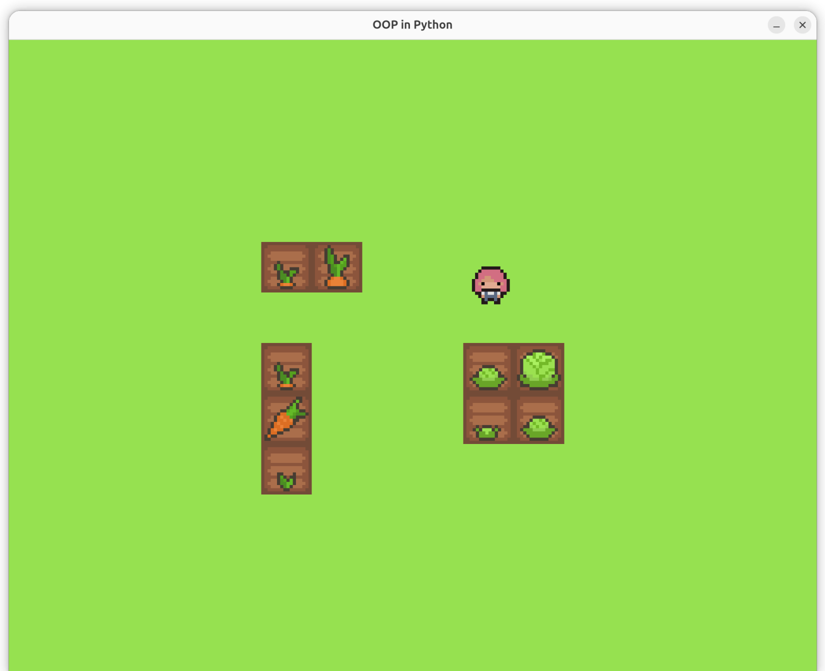
Übung 10

Implementieren Sie das Anpflanzen von Karotten:

- Implementieren Sie eine neue Klasse für Karotten. Die Methoden sollen die gleichen Namen haben wie beim Salat.
- Das Bewässern soll nur mit 50% Wahrscheinlichkeit erfolgreich sein.
- Implementieren Sie eine Methode in der Farm-Klasse, um eine Karotte zu pflanzen.
- Prüfen Sie in der Spieler-Klasse, ob die Taste ‚1' gedrückt ist, und pflanzen Sie dann entsprechend eine Karotte.

4.4. Zusammenfassung

Unser Spiel sieht nun in etwa so aus:



Wir haben unser Spiel erweitert und die Möglichkeit implementiert, Gemüse anzupflanzen. Wir haben uns auch kurz angeschaut, was Referenzen sind: „Zeiger" auf ein Objekt. Diese erlauben uns, von verschiedenen Orten in unserem Programm auf das gleiche Objekt zuzugreifen.

Kapitel 5: Vererbung

Die beiden Pflanzen, die wir implementiert haben, sind sich sehr ähnlich und haben teilweise den gleichen Code. Auch konzeptuell sind sie sich ähnlich. Wir werden nun eine Möglichkeit kennenlernen, wie wir diese beiden Klassen miteinander „verbinden" können, damit wir nicht alles doppelt implementieren müssen.

5.1. Kindklassen und Elternklassen

Theorie

Eine Klasse kann von einer anderen Klasse *erben*. Die erbende Klasse wird auch *abgeleitete Klasse* oder *Kindklasse* genannt und die Klasse, von der geerbt wird, nennt man analog die *Basisklasse* oder *Elternklasse*.

Die Kindklasse übernimmt alle Member und Methoden der Elternklasse. Sie kann zusätzliche Member und Methoden definieren.

Weiter kann die Kindklasse Methoden der Elternklasse *überschreiben*, indem sie eine Methode mit dem gleichen Namen wie in der Elternklasse definiert. Wird nun diese Methode auf einem Objekt aufgerufen, wird die Methode aus der Kindklasse ausgeführt statt die Methode aus der Elternklasse.

In Python gibt man die Basisklasse in Klammern nach dem Klassennamen an, z.B. `class Dog(Animal):`. In einer Methode, die wir überschreiben, haben wir auch die Möglichkeit, die ursprüngliche Methode aus der Elternklasse aufzurufen. Dafür schreibt man `super().method()`.

Schauen wir uns das Ganze mit unserem Beispiel mit den Hunden und Katzen an. Wir wollen das gleiche Verhalten wie in Kapitel 3 implementieren, aber durch Vererbung wo möglich die Methoden nur einmal implementieren. Abbildung 21 zeigt die Basisklasse `Animal` sowie die beiden Kindklassen `Dog` und `Cat`, die wir implementieren möchten. Tabelle 2 zeigt farblich, welche Member und welche Methoden wir in welcher Klasse implementieren werden. Wie wir sehen, können wir in `Dog` alle Methoden von `Animal` übernehmen und müssen nur eine zusätzliche hinzufügen. Für die Klasse `Cat` müssen wir jedoch zusätzliche Member definieren und einige Methoden überschreiben, da sie sich von der Implementation in `Animal` unterscheiden.

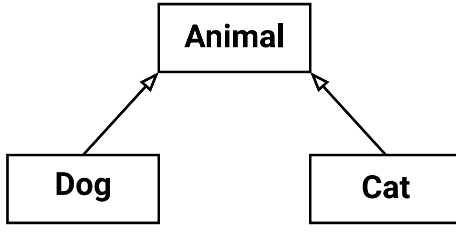


Abbildung 21: Die Basisklasse Animal mit den beiden Kindklassen Dog und Cat.

Animal	Dog	Cat
name	name	name
energy	energy	energy
		color
		chace_count
__init__()	__init__()	__init__()
full_name()	full_name()	full_name()
feed()	feed()	feed()
	walk()	
		chase()

Tabelle 2: Übersicht über die Member und Methoden der Klassen.

Zunächst definieren wir eine Basisklasse für alle Tiere. Dieses Beispiel ist wieder unabhängig von unserem Spiel und dient nur dem einfacheren Verständnis.

python

```python
1  class Animal:
2
3      def __init__(self, name):
4          self.name = name
5          self.energy = 0
6
7      def full_name(self):
8          return self.name
9
10     def feed(self):
11         print("Feed", self.full_name())
12         self.energy = 50
```

Alle Tiere haben einen Namen und eine aktuelle Energie, und wir können sie füttern. Zusätzlich haben wir eine Methode `full_name`, die den kompletten Namen zurückgibt. Für die meisten Tiere wird dies einfach der angegeben Name sein, aber für Katzen wollen wir auch die Fellfarbe als Teil des Namens ausgeben. In der Implementation in `Animal` geben wir daher den Namen zurück und in der Klasse `Cat` werden wir die Methode dann überschreiben.

Die Hund-Klasse können wir wie folgt schreiben:

python

```python
15 class Dog(Animal):
16
17     def walk(self):
18         if self.energy < 20:
19             print(self.full_name(), "is too tired for a walk")
20         else:
21             print("Go for a walk with", self.full_name())
22             self.energy -= 20
```

Diese Klasse erbt neu von `Animal`. Der Konstruktor und die Methoden `full_name` und `feed` sind in der Elternklasse implementiert und müssen wir nicht erneut implementieren. Der Hund hat die spezielle Methode `walk`, die wir noch implementiert haben. Wie in Tabelle 2 ersichtlich, übernimmt `Dog` die Methoden `full_name` und `feed` von der Basisklasse `Animal`.

Bei der Katze müssen wir noch weitere Methoden überschreiben, da wir hier zusätzlich die Fellfarbe ausgeben müssen und wir einen zusätzlichen Member benötigen, um die Anzahl Mäusejagden zu speichern.

```python
25 class Cat(Animal):
26 
27     def __init__(self, name, color):
28         super().__init__(name)
29         self.color = color
30         self.chase_count = 0
31 
32     def full_name(self):
33         return self.name + " (" + self.color + ")"
34 
35     def chase(self):
36         if self.energy < 20:
37             print(self.full_name(), "is too tired for a chase")
38         else:
39             self.energy -= 20
40             self.chase_count += 1
41             if self.chase_count == 3:
42                 self.chase_count = 0
43                 print(self.full_name(), "caught a mouse")
44             else:
45                 print(self.full_name(), "did not catch a mouse")
```

Hier sieht man nun, wie wir im Konstruktor den Konstruktor der Elternklasse aufrufen. Dank der Methode `full_name` wird beim Aufruf von `feed` automatisch die Fellfarbe miteinbezogen, ohne dass wir diese Methode überschreiben mussten. Dies ist ein verbreitetes Muster, dass in der Elternklasse gewisse Klassen explizit zum Überschreiben definiert werden, damit die Kindklassen das Verhalten der Elternklasse einfacher anpassen können.

Unsere Klassen testen wir nun mit folgendem Code:

```python
48 if __name__ == "__main__":
49     dog = Dog("Chipsi")
50     cat = Cat("Tweety", "black/white")
51 
52     dog.feed()
53     dog.walk()
54     dog.walk()
```

```
55     dog.walk()
56     print("---")
57 
58     cat.feed()
59     cat.chase()
60     cat.chase()
61     cat.chase()
62     cat.feed()
63     cat.chase()
```

Wir erhalten die folgende Ausgabe:

```
 1 Feed Chipsi
 2 Go for a walk with Chipsi
 3 Go for a walk with Chipsi
 4 Chipsi is too tired for a walk
 5 ---
 6 Feed Tweety (black/white)
 7 Tweety (black/white) did not catch a mouse
 8 Tweety (black/white) did not catch a mouse
 9 Tweety (black/white) is too tired for a chase
10 Feed Tweety (black/white)
11 Tweety (black/white) caught a mouse
```

5.2. Refactoring

Wir möchten nun unsere Klassen für die Pflanzen anpassen, damit wir durch Vererbung weniger Code doppelt schreiben müssen. Die beiden Pflanzen unterscheiden sich nur durch die anzuzeigenden Tiles und dem Verhalten beim Bewässern.

Dieses Teilen von Verhalten zwischen ähnlichen Klassen ist ein weiterer Vorteil von objektorientierter Programmierung. Mit Vererbung können wir ähnliches Verhalten sehr gut abstrahieren und müssen den Code nicht mehrfach schreiben.

Die verschiedenen Tiles können wir lösen, indem wir im Konstruktor der Elternklasse einen Parameter für die Tiles hinzufügen. In den Kindklassen können wir dann den Konstruktor überschreiben und die benötigten Tiles der Elternklasse übergeben.

Für das Verhalten beim Bewässern stellen wir eine generische Methode zur Verfügung, die ins nächste Wachstumsstadium wechselt. Die beiden Pflanzenklassen überschreiben dann die Methode zur Bewässerung und rufen die generische Methode auf, falls die Kriterien erfüllt sind.

Die Basisklasse sieht wie folgt aus:

python

```python
10 class Plant:
11 
12     def __init__(self, game, position, growth_tiles):
13         self.game = game
```

```
14          self.position = position
15          self.growth_tiles = growth_tiles
16          self.current_growth = 0
17
18      def water(self):
19          pass
20
21      def advance_growth(self):
22          if self.current_growth + 1 < len(self.growth_tiles):
23              self.current_growth += 1
24
25      def draw(self):
26          self.game.draw_tile(self.growth_tiles[self.current_growth],
self.position)
```

Die Methode `advance_growth` ist die generische Methode für den Wechsel des Wachstumsstadiums.

Die Klasse für die Karotten kann man wie folgt implementieren:

```python
38  class Carrot(Plant):
39
40      def __init__(self, game, position):
41          super().__init__(game, position, [(11, 2), (10, 2), (9, 2), (8,
2)])
42
43      def water(self):
44          if random.random() < 0.5:
45              self.advance_growth()
```

Im Konstruktor geben wir zusätzlich zu `game` und `position`, die wir als Parameter erhalten, auch die zu verwendenden Tiles an den Konstruktor der Elternklasse mit. In `water` rufen wir die generische Methode nur auf, wenn die Zufallszahl im akzeptablen Bereich war.

Wie Sie sehen, ist die neue Implementation viel kürzer, da der Grossteil der Funktionalität in der Basisklasse implementiert ist.

Übung 11

Passen Sie die Klasse für den Salat an, um die neue Basisklasse zu verwenden.

Optional

Im Tilesheet gibt es noch Grafiken für weitere Pflanzen. Mit der Basisklasse ist es nun nicht mehr viel Aufwand, auch noch eine weitere Pflanzensorte hinzuzufügen.

5.3. Zusammenfassung

In diesem Kapitel haben wir das Konzept der Vererbung kennengelernt. Wenn eine Klasse von einer anderen Klasse erbt, übernimmt sie alle Members und Methoden. Die ableitende Klasse kann auch Methoden überschreiben, um das Verhalten anzupassen.

Vererbung kann man unter anderem dazu verwenden, Code in verschiedenen Klassen wiederzuverwenden.

Kapitel 6: Gameobject, Interfaces, und Ducktyping

Mit dem Wissen aus den vorherigen Kapiteln können wir einige unserer Klassen noch etwas anpassen und vereinfachen. Wir werden uns auch noch ganz kurz anschauen, inwiefern sich objektorientiertes Programmieren in Python von anderen Sprachen unterscheidet.

Optional

Das gesamte Kapitel 6 ist optional. Es enthält einige Hintergrundinformationen für Interessierte, ist aber nicht unbedingt notwendig für das Verständnis der behandelten Themen.

6.1. Gameobject

Der Wrapper stellt eine Basisklasse `GameObject` zur Verfügung für Klassen, die in jedem Frame aktualisiert und/oder gezeichnet werden müssen. Objekte von diesen Klassen können dann mit `game.add_game_object(obj)` hinzugefügt werden.

Klassen, die von `GameObject` erben, können die folgenden Methoden überschreiben:

- `def update(self):` wird einmal pro Frame aufgerufen, um den Zustand zu aktualisieren.
- `def draw(self):` wird einmal pro Frame aufgerufen, um die Grafiken zu zeichnen. Die Objekte werden in der Reihenfolge gezeichnet, in der sie hinzugefügt wurden.
- `def on_keydown(self, key):` wird jedesmal aufgerufen, wenn der Benutzer eine Taste neu gedrückt hat. Damit muss man nicht selbst den alten Status der Tasten speichern.

Der Konstruktor benötigt eine Referenz auf `game`.

Wenn alle Logik in Gameobjekten implementiert ist, wird die explizite Spiel-Schleife nicht mehr benötigt. Stattdessen kann man `game.run()` aufrufen. Dies wird dann solange laufen, bis der Spieler das Spiel beendet.

Die relevanten Teile sehen nun wie folgt aus:

python
```python
5 from gamelib import Game, TILE_SIZE, GameObject
```

python
```python
48 class Farm(GameObject):
49 
50     def __init__(self, game):
51         super().__init__(game)
52         self.farm_size = (
53             WINDOW_SIZE[0] // TILE_SIZE + 1,
54             WINDOW_SIZE[1] // TILE_SIZE + 1
55         )
```

```python
56          self.plants = [[None for x in range(self.farm_size[0])] for y in
range(self.farm_size[1])]
```

python

```python
85 class Player(GameObject):
86
87     def __init__(self, game, farm):
88         super().__init__(game)
89         self.farm = farm
90         self.position = ((WINDOW_SIZE[0] - TILE_SIZE) / 2, (WINDOW_SIZE[1]
- TILE_SIZE) / 2)
```

python

```python
110     def on_keydown(self, key):
111         x, y = self.current_farmplot()
112         if key == pygame.K_1:
113             self.farm.plant_carrot(x, y)
114         if key == pygame.K_2:
115             self.farm.plant_salad(x, y)
116         if key == pygame.K_e:
117             self.farm.water(x, y)
118         if key == pygame.K_q:
119             self.farm.remove_plant(x, y)
```

python

```python
130 def main():
131     game = Game("OOP in Python", WINDOW_SIZE)
132     farm = Farm(game)
133     player = Player(game, farm)
134     game.add_game_object(farm)
135     game.add_game_object(player)
136     game.run()
```

Übung 12

Setzen Sie die Änderungen um, sodass `Farm` und `Player` von `GameObject` erben und Sie die explizite Spiel-Schleife nicht mehr benötigen.

6.2. Interfaces

Theorie

Ein *Interface* ist eine Klasse, die selbst kein Verhalten implementiert, sondern nur dazu dient, eine Schnittstelle zu definieren.

Die Klasse `GameObject` ist ein Beispiel eines Interfaces. Die Hauptaufgabe der Klasse ist, zu dokumentieren, welche Methoden notwendig sind, damit man eine Klasse mit `game.add_game_object` verwenden kann. So ist der Wrapper auch sicher, dass alle notwendigen Methoden vorhanden sind.

In Python haben Interfaces wenig Bedeutung, da in Python vor dem Ausführen des Programms nicht überprüft wird, ob alle benötigten Members und Methoden vorhanden sind. In anderen Programmiersprachen wie z.B. Java werden Interfaces jedoch oft verwendet und sind ein wichtiger Bestandteil der objektorientierten Programmierung.

6.3. Ducktyping

Der Grund, warum Interfaces in Python wenig Bedeutung haben, ist das sogenannte *Ducktyping*. Man muss nicht explizit von einem Interface erben, es reicht, wenn man alle benötigten Methoden implementiert hat. In unserem Beispiel von `GameObject` hätten wir bei `Player` nur `on_keydown` definieren müssen, dann hätte man es für `game.add_game_object(...)` verwenden können, da wir bereits die Methoden `update` und `draw` definiert hatten.

Der Begriff *Ducktyping* kommt von „If it walks like a duck and it quacks like a duck, then it must be a duck“, also „Wenn es wie eine Ente läuft und wie eine Ente quackt, dann ist es eine Ente.“ Damit sagt man aus, dass wenn sich ein Objekt konform zu einem Interface verhält, dann kann man das Objekt überall verwenden, wo das Interface benötigt wird. In unserem Beispiel also wenn man etwas aktualisieren und zeichnen kann, dann kann man es als Gameobjekt verwenden.

In Python muss man nicht einmal die Klasse für das Interface schreiben. Das kann implizit geschehen, solange alle Klassen die benötigten Methoden bereitstellen.

6.4. Zusammenfassung

Wir haben noch etwas über den Tellerrand hinausgeschaut und uns Interfaces angeschaut. Dies sind Basisklassen, die selbst kein Verhalten implementieren und nur eine Schnittstelle definieren. Aufgrund von Ducktyping haben Interfaces in Python wenig Bedeutung, doch in anderen Sprachen sind diese sehr wichtig.

Kapitel 7: Mögliche Erweiterungen

Wir haben nun eine einfache Version eines Garten-Spiels programmiert. In diesem Kapitel finden Sie noch weitergehende Informationen zur Spieleentwicklung und Vorschläge, wie Sie das Spiel noch erweitern könnten. Diese zielen hauptsächlich darauf ab, dass das Spiel besser aussieht oder mehr Spass zum Spielen macht.

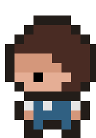
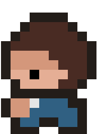
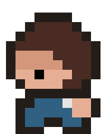
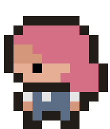
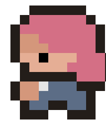
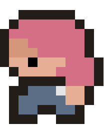
Optional

Das gesamte Kapitel 7 ist optional. Wählen Sie einige der Vorschläge aus, die Sie interessieren, und setzen Sie diese in Ihrem Spiel um.

7.1. Animationen

Wenn sich der Spieler bewegt, zeigen wir momentan immer die gleiche Grafik an. Wir können eine Animation anzeigen, indem wir alle paar Frames das Tile wechseln, das wir anzeigen. Im Tilesheet gibt es bereits Tiles für verschiedene Animationen des Spielers.

Z.B. die folgenden Tiles können für das Laufen nach Links verwendet werden:

Werden diese Tiles etwa 10 Mal pro Sekunde weitergeschaltet, ergibt dies eine halbwegs flüssige Bewegung.

Der Wrapper stellt eine Methode bereit, um eine Animation zu erzeugen:

```python
animation_left = self.game.create_animation(
  [(6, 0), (7, 0), (8, 0)],
  fps=10
)
```

Diese Animation kann dann gezeichnet werden:

```python
animation_left.draw(position)
```

Der Wrapper schaltet automatisch mit der angegebenen Geschwindigkeit (FPS) durch die verschiedenen Tiles.

Bei den Grafiken gibt es verschiedene Animationen für die verschiedenen Richtungen. Sie können für jede Richtung eine eigene Animation erzeugen und dann abhängig von der Geschwindigkeit des Spielers die korrekte Animation zeichnen.

7.2. Partikelsysteme

Eine beliebte Methode, um ein Spiel visuell interessanter zu gestalten, sind Partikelsysteme. Partikel sind kleine Teilchen, die aber oft in grossen Mengen auftreten. Beispiele aus der echten Welt sind z.B. Staub-Partikel. In Spielen sind Partikel ebenfalls kleine, oft farbige, Teile, die auf dem Bildschirm „herumwirbeln“.

Bei einem Partikelsystem werden kleine Partikel von einem Emitter generiert, die dann über Zeit ihre Geschwindigkeit, Farbe, und andere Parameter ändern. Nach einer bestimmten Lebensdauer verschwinden sie wieder.

Die Details, wie genau die Partikelsysteme funktionieren, sind aber nicht wichtig, um sie anzuwenden. Im `assets` Ordner gibt es ein paar vordefinierte Presets für Partikelsysteme (alle Dateien, die mit `.json` enden). Diese können mit `ParticleSystem` aus `particlelib` geladen und dargestellt werden.

Im Folgenden die wichtigsten Methoden eines Partikelsystems:

```python
from particlelib import ParticleSystem
# Ein Preset laden
watering_particle_system = ParticleSystem(game, "particle_watering.json")
# Position des Emitters setzen
watering_particle_system.set_position(position)
# Emitter für 0.5 Sekunden aktivieren
watering_particle_system.enable_burst_emitting(0.5)
# Emitter aktivieren (True) oder deaktivieren (False)
watering_particle_system.set_emitting(True)
# Partikelsystem zeichnen
watering_particle_system.draw()
```

Die folgenden Presets sind verfügbar:

- `particle_watering.json`: Eine Pflanze wird bewässert.
- `particle_walking.json`: Dreck, der vom Spieler beim Laufen aufgewirbelt wird.
- `particle_growth.json`: Eine Pflanze hat ihr nächstes Wachstumsstadium erreicht.
- `particle_growth_failed.json`: Eine Pflanze wurde bewässert, aber kein Wachstum hat stattgefunden.

Die allgemeine Verwendung der Partikelsysteme ist wie folgt angedacht:

- Bei der Erstellung der Objekte wird das Partikelsystem erstellt.
- Die Position des Partikelsystems wird aktualisiert, wenn sich das Objekt bewegt (z.B. wenn der Spieler läuft).
- Bei einem bestimmten Event (z.B. Bewässerung) wird das Partikelsystem kurz angeschalten.
- Während dem Zeichnen des Objekts werden auch die zugehörigen Partikelsysteme gezeichnet.

7.3. Bessere Wachstumsmechanik

Zur Zeit ist der Prozess, wie die Pflanzen wachsen, sehr einfach gestaltet. Dies könnte etwas spannender gestaltet werden. Eine Idee wäre folgender Ablauf:

- Beim Bewässern wird der Wasserstand der Pflanze aufgefüllt.
- Solange der Wasserstand genügend hoch ist, wächst die Pflanze selbständig.
- Nach einer gewissen Zeit von erfolgreichem Wachstum wechselt die Pflanze in den nächsten Entwicklungszustand.
- Der Wasserstand sinkt kontinuierlich.
- Ist der Wasserstand zu tief, stoppt das Wachstum der Pflanze.

Dies würde verhindern, dass eine Pflanze durch mehrfaches Bewässern sehr schnell wächst. Ausserdem gibt es Veränderungen bei den Pflanzen unabhängig davon, ob der Spieler gerade damit interagiert.

7.4. Fruchtbäume

Im Tilesheet gibt es Grafiken für Fruchtbäume. Diese bestehen jeweils aus vier Tiles, die entsprechend nebeneinander gezeichnet werden müssen. Man könnte diese ähnlich wie die bestehenden Pflanzen bewirtschaften, um Früchte davon zu erhalten. Je nach der gewählten Spielmechanik für die Bäume bietet sich allenfalls eine zusätzliche Basisklasse für die Bäume an.

7.5. Inventar

Momentan können fertig gewachsene Pflanzen nicht geerntet werden. Man könnte ein Inventar einführen, in dem die fertigen Pflanzen abgelegt werden. Statt nur Pflanzen zu entfernen, würden dann die Pflanzen, die das letzte Wachstumsstadium erreicht haben, ins Inventar verschoben. Dem Spieler kann so z.B. angezeigt werden, wie viele Pflanzen bereits geerntet wurden.

7.6. Toolbar

Es ist für den Spieler momentan nicht ersichtlich, welche Tasten gedrückt werden müssen für die verschiedenen Aktionen wie z.B. die Bewässerung oder das Pflanzen. Mit einer Toolbar, welche die verschiedenen Aktionen anzeigt, könnten die Tasten angezeigt werden. Zusätzlich wäre es möglich, dass die Aktionen auch mit der Maus in der Toolbar ausgelöst werden.

Um die Mausinteraktion zu implementieren, sind folgende Funktionen hilfreich:

- `pygame.mouse.get_pos()` gibt die aktuelle Position der Maus zurück.
- `left, middle, right = pygame.mouse.get_pressed()` gibt zurück, welche Maustasten gedrückt sind.

Um Text zeichnen zu können, kann folgender Code verwendet werden:

```python
1 # In Konstruktor einmalig ausführen
```

```
2 font = pygame.font.SysFont("Arial", 18)
3 # Text 'Hi!' in Farbe (0, 0, 0) zeichnen
4 rendered = font.render("Hi!", True, (0, 0, 0))
5 game.get_screen().blit(rendered, position)
```

7.7. Markt

Im Tilesheet gibt es eine Grafik für einen Marktstand. Statt die Pflanzen direkt anpflanzen zu können, könnte man zuerst Samen beim Marktstand kaufen müssen. Ebenfalls könnte man die fertig gewachsenen Pflanzen auf dem Markt wieder verkaufen.

Der Marktstand besteht aus vier Tiles im Tilesheet, die entsprechend nebeneinander gezeichnet werden müssen.

7.8. Zusammenfassung

Wir haben in diesem Skript ein kleines Garten-Spiel entwickelt und am Ende nun noch einige individuelle Weiterentwicklungen angeschaut. Sie haben nun alle Grundlagen, um das Spiel selbständig weiterzuentwickeln und Funktionen hinzuzufügen, die Sie interessant finden.