Handbuch: Figuren mit der Klasse Animation animieren

Das Animationssystem spielt Bildfolgen für den Schüler, den Hausmeister und weitere Figuren ab. Jede Figur legt ihre Zustände, Blickrichtungen und Animationstempi fest. Die gemeinsame Klasse `Animation` übernimmt das Abspielen der Bilder.

Eine **Animation** ist eine Folge von Bildern, die nacheinander angezeigt werden. Ein einzelnes Bild heißt **Frame**. Das sichtbare Bild einer Spielfigur wird auch **Sprite** genannt.

1. Die Aufgaben sinnvoll aufteilen

Das Animationssystem besteht aus folgenden Teilen:

Teil	Aufgabe
scripts/animation.py	Zeit zählen, aktuelles Bild bestimmen, wiederholen oder am Ende stoppen
scripts/utils.py	Bilddateien finden und daraus Animationsvorlagen erstellen
scripts/player.py	Bewegung, Kollision, Blickrichtung und Spielerzustand bestimmen
game.py	Eingaben und Spielablauf verwalten, etwa die Interaktion mit `E` starten
tests/test_animation.py	Die gemeinsame Klasse und die Schüler- und Hausmeister-Assets prüfen
tests/test_player.py	Das Zusammenspiel mit der Spielersteuerung prüfen

Die Figur wählt die passende Animation für ihre Tätigkeit und Blickrichtung. Die Animation bestimmt anhand der vergangenen Zeit das sichtbare Bild.

`Animation` kennt deshalb weder WASD noch Wände, Lebenspunkte oder bestimmte Namen wie `walk` und `attack`. Die Klasse kann jede nicht leere Bildfolge abspielen.

Methoden stehen eingerückt innerhalb ihrer Klasse. Code innerhalb von `__init__()`, `update()` oder einer Bedingung erhält die entsprechende zusätzliche Einrückung.

2. Die Assets der beiden Figuren verstehen

Die Einzelbilder liegen in diesen Ordnern:

```
data/textures/
├── schueler/64/frames/
|   ├── idle/south/00.png
|   ├── walk/east/00.png
|   └── ...
└── hausmeister/64/frames/
    ├── idle/south/00.png
    ├── attack/east/00.png
    └── ...
```

Die allgemeine Struktur lautet:

```
<Figur>/<Auflösung>/frames/<Zustand>/<Blickrichtung>/<Bildnummer>.png
```

Jeder vorhandene Zustand hat Bilder für die acht Richtungen `south`, `southwest`, `west`, `northwest`, `north`, `northeast`, `east` und `southeast`.

Die Figuren verfügen über folgende Zustände und Bildanzahlen pro Richtung:

Zustand	Schüler	Hausmeister
`idle`	2	2
`walk`	6	6
`run`	6	—
`interact`	4	—
`alert`	—	2
`attack`	—	6
`hurt`	2	2
`death`	6	6

Ein Gedankenstrich kennzeichnet einen Zustand ohne zugehörige Bildfolge.

Die Zustandsnamen werden beim Laden und Auswählen der Animationen verwendet. Die Klasse `Animation` verarbeitet die Bildfolge unabhängig von ihrem Zustandsnamen.

Die Einzelbilder lassen sich direkt laden. Ein Zerlegen der ebenfalls vorhandenen Spritesheets wie `walk.png` ist dafür nicht erforderlich.

3. Den Pfad zu den Bildern festlegen

Datei: `scripts/utils.py`

Der Pfad zu den Bildern wird aus dem Speicherort von `utils.py` bestimmt:

```python
from pathlib import Path

import pygame

from scripts.animation import Animation

images_path = Path(__file__).resolve().parent.parent / "data" / "textures"


def load_image(path: str | Path) -> pygame.Surface:
    return pygame.image.load(images_path / path)
```

`__file__` bezeichnet `utils.py`. Das erste `.parent` führt zum Ordner `scripts`, das zweite zum Projektordner. Danach werden `data` und `textures` angehängt.

Der Schrägstrich verbindet bei `Path` einzelne Pfadteile. Der Bildpfad hängt dadurch nicht davon ab, welcher Arbeitsordner beim Start eingestellt ist.

`load_image()` akzeptiert einen Text oder ein `Path`-Objekt und liefert eine `pygame.Surface`. Das ist eine Bildfläche, die Pygame später zeichnen kann.

Der Import von `Animation` ermöglicht es dem Bildlader, aus den geladenen Bildflächen Animationsvorlagen zu erstellen.

4. Eine Animation mit eigenem Abspielzustand anlegen

Datei: `scripts/animation.py`

Der Konstruktor legt fest, welche Bilder abgespielt werden und wie lange jedes Bild sichtbar bleibt:

```python
import math


class Animation:
    def __init__(self, images, image_duration=0.125, loop=True):
        if not images:
            raise ValueError("Eine Animation benötigt mindestens ein Bild.")
        if not math.isfinite(image_duration) or image_duration <= 0:
            raise ValueError("Die Bilddauer muss endlich und größer als null sein.")

        self.images = images
        self.image_duration = image_duration
        self.loop = loop
        self.time = 0.0
        self.done = False
```

Attribut	Bedeutung
`images`	Die bereits geladenen Bilder in der richtigen Reihenfolge
`image_duration`	Anzeigedauer eines Bildes in Sekunden
`loop`	`True`: wiederholen; `False`: einmal abspielen
`time`	Aktuelle Abspielzeit innerhalb der Bildfolge
`done`	Ob eine einmalige Animation vollständig abgespielt wurde

`0.125` Sekunden pro Bild entsprechen acht Animationsbildern pro Sekunde, denn `1 / 0.125 = 8`.

Leere Bildfolgen sowie ungültige Bilddauern werden sofort abgewiesen. `math.isfinite()` schließt auch unendliche Werte und `NaN` aus. So lässt sich mit den Zeitwerten zuverlässig rechnen.

Bilder teilen, Abspielzeit trennen

Die Methode `copy()` erzeugt eine neue Animation mit denselben Bildern:

```
def copy(self):
    return Animation(self.images, self.image_duration, self.loop)
```

Die Bildflächen werden dabei nicht erneut geladen oder kopiert. Jede Kopie bekommt aber ihre eigene `time` und ihr eigenes `done` und startet beim ersten Bild.

Das ist für mehrere Gegner hilfreich: Sie können dieselben geladenen Bilder verwenden, ohne gleichzeitig an derselben Stelle ihrer Animation zu sein.

5. Die Animation mit dt aktualisieren

Datei: `scripts/animation.py`, Methode `update()`

```
def update(self, dt):
    if not math.isfinite(dt) or dt < 0:
        raise ValueError("dt muss endlich und größer oder gleich null sein.")

    self.time += dt
    total_duration = len(self.images) * self.image_duration
    if self.loop:
        self.time %= total_duration
    else:
        self.time = min(self.time, total_duration)
        self.done = self.time >= total_duration
```

`dt` ist die seit dem letzten Spielbild vergangene Zeit in Sekunden. Sie wird in `Game.run()` berechnet:

```
self.dt: float = self.clock.tick(60) / 1000.0
```

`tick()` liefert Millisekunden. Die Division durch `1000.0` rechnet diese in Sekunden um. Der Wert wird bis zu `Animation.update(dt)` weitergegeben.

Damit hängt das Animationstempo von der verstrichenen Zeit ab. Es hängt nicht davon ab, wie oft `update()` aufgerufen wird.

Wiederholte Animationen

Bei `loop=True` hält `%` die Abspielzeit innerhalb der Animationsdauer. Wenn eine Bildfolge insgesamt `0,75` Sekunden dauert, führt eine Zeit von `0,80` Sekunden wieder zu `0,05` Sekunden innerhalb dieser Folge.

Auch ein größerer Zeitschritt, der mehrere Durchläufe überspringt, landet wieder an der passenden Stelle. `done` bleibt bei einer Schleife `False`.

Einmalige Animationen

Bei `loop=False` wird die Abspielzeit auf die Gesamtdauer begrenzt. `done` wird erst dann `True`, wenn auch das letzte Bild seine vollständige Anzeigedauer erreicht hat.

Beispiel mit drei Bildern und `0,25` Sekunden pro Bild:

Zeit	Sichtbares Bild	done
`0,00` bis vor `0,25`	Erstes Bild	`False`
`0,25` bis vor `0,50`	Zweites Bild	`False`
`0,50` bis vor `0,75`	Letztes Bild	`False`
Ab `0,75`	Letztes Bild bleibt sichtbar	`True`

Die Klasse wechselt danach nicht selbst zu `idle`. Ob eine Figur wieder steht oder tot liegen bleibt, entscheidet die Figur.

6. Das aktuelle Bild bestimmen

Datei: `scripts/animation.py`, Methode `img()`

```
def img(self):
    index = int(self.time / self.image_duration)
    return self.images[min(index, len(self.images) - 1)]
```

Beispiel: Nach `0,3` Sekunden bei einer Bilddauer von `0,125` Sekunden ergibt sich:

```
int(0,3 / 0,125) = int(2,4) = 2
```

Listen beginnen bei `0`. Index `2` bezeichnet deshalb das dritte Bild, hier `02.png`.

Die Begrenzung mit `min()` verhindert einen Zugriff hinter das letzte Bild. Das ist besonders am Ende einer einmaligen Animation wichtig: Bei drei Bildern ist die größte gültige Position `2`, auch wenn die Rechnung nach der Gesamtdauer `3` ergibt.

Der Zugriff auf das aktuelle Bild erfolgt über `img()`:

```
image = animation.img()
```

7. Einen gemeinsamen Lader für verschiedene Figuren verwenden

Datei: `scripts/utils.py`, Funktion `load_animations()`

`load_animations()` lädt die Bildfolgen einer Figur und erstellt daraus Animationsvorlagen:

```
def load_animations(path, animation_fps, directions, non_looping=()):
    directions = tuple(directions)
    animations = {}
    for state, fps in animation_fps.items():
        if not fps > 0:
            raise ValueError(f"Das Animationstempo für {state} muss größer als null sein."
        animations[state] = {}
        for facing in directions:
            folder = images_path / path / state / facing
            files = sorted(folder.glob("*.png"))
            if not files:
                raise FileNotFoundError(f"Keine Animationsbilder in {folder}")
            images = [load_image(file.relative_to(images_path)).convert_alpha() for file i
            animations[state][facing] = Animation(
                images, image_duration=1 / fps, loop=state not in non_looping
            )
    return animations
```

Die Funktion bekommt vier Angaben:

Parameter	Beispiel	Bedeutung
`path`	`"schueler/64/frames"`	Pfad relativ zu `data/textures`
`animation_fps`	`{"idle": 3, "walk": 8}`	Zustände und ihre Animationsbilder pro Sekunde
`directions`	`("south", "east")`	Die zu ladenden Richtungsordner
`non_looping`	`("hurt", "death")`	Zustände, die nur einmal abgespielt werden

Damit sind weder die Schüler-Assets noch bestimmte Zustände fest in den Lader eingebaut. Auch eine Figur mit weniger Richtungen kann diese Funktion verwenden, wenn sie nur die tatsächlich vorhandenen Richtungsnamen übergibt.

`directions = tuple(directions)` hält die Richtungen für alle Zustände bereit. Das funktioniert auch, wenn der Aufrufer einen Iterator übergibt, der sonst nach dem ersten Durchlauf verbraucht wäre.

`glob("*.png")` findet die Bilder. `sorted()` sortiert `00.png`, `01.png` und so weiter in die richtige Reihenfolge. Die führenden Nullen sind dafür hilfreich.

`convert_alpha()` bereitet die Bilder für die Anzeige mit Transparenz vor. Dafür muss bereits `pygame.display.set_mode(...)` aufgerufen worden sein. In `Game.__init__()` geschieht das vor dem Erzeugen des Spielers.

Die Bilddauer wird aus dem Tempo berechnet: `image_duration=1 / fps`. Mit `loop=state not in non_looping` wird für jeden Zustand festgelegt, ob er sich wiederholt.

Das Ergebnis ist ein Dictionary mit der Struktur Zustand → Richtung → `Animation`:

```
vorlage = animations["walk"]["east"]
aktive_animation = vorlage.copy()
```

Die Vorlagen werden aufbewahrt und nicht selbst aktualisiert. Jede Figur spielt eine eigene Kopie ab. Innerhalb einer geladenen Vorlage teilen sich die Kopien die Bildflächen. Separate Aufrufe von `load_animations()` laden die Bilder erneut; die Funktion hat keinen globalen Cache.

8. Player mit einer aktiven Animation ausstatten

Datei: `scripts/player.py`

`Player` importiert den gemeinsamen Lader:

```
from scripts.utils import load_animations
```

Die Klassenattribute `Player.ANIMATION_FPS`, `Player.DIRECTIONS` und `Player.ACTIONS` legen das Animationstempo, die Richtungszuordnung und die einmaligen Aktionen des Schülers fest:

```
ANIMATION_FPS = {
    "idle": 3,
    "walk": 8,
    "run": 12,
    "interact": 8,
    "hurt": 8,
    "death": 8,
}
DIRECTIONS = {
    (0, 1): "south",
    (-1, 1): "southwest",
    (-1, 0): "west",
    (-1, -1): "northwest",
    (0, -1): "north",
    (1, -1): "northeast",
    (1, 0): "east",
    (1, 1): "southeast",
}
ACTIONS = ("interact", "hurt", "death")
```

Direkt nach dem Erzeugen der Kollisionsbox in `Player.__init__()` steht:

```
self.animations = load_animations(
    "schueler/64/frames", self.ANIMATION_FPS, self.DIRECTIONS.values(), non_looping=self.A
)
self.state = "idle"
self.facing = "south"
self.animation = self.animations[self.state][self.facing].copy()
```

Die Vorlagen und die aktive Animation werden getrennt gespeichert:

Attribut	Inhalt
`self.animations`	Alle verfügbaren Vorlagen nach Zustand und Richtung
`self.animation`	Die gerade abgespielte Kopie einer dieser Vorlagen

Die Abspielzeit liegt in `self.animation.time`. Die aktive Animation verwaltet außerdem die Wiederholung und den aktuellen Bildindex.

9. Zustandswechsel und Richtungswechsel unterscheiden

Datei: `scripts/player.py`

Ein neuer Zustand beginnt mit einer frischen Kopie der passenden Vorlage:

```
def _set_state(self, state):
    if self.state != state:
        self.state = state
        self.animation = self.animations[self.state][self.facing].copy()
```

Beim Wechsel von `walk` zu `run` startet die Rennanimation also vorne. Bleibt der Zustand unverändert, wird nichts neu gestartet. Sonst würde in jedem Spielbild erneut das erste Animationsbild ausgewählt.

Beim Drehen innerhalb desselben Zustands erhält `_set_facing()` die Abspielphase:

```
def _set_facing(self, facing):
    if self.facing != facing:
        elapsed = self.animation.time
        self.facing = facing
        self.animation = self.animations[self.state][self.facing].copy()
        self.animation.update(elapsed)
```

Die Methode merkt sich die bisherige Abspielzeit, wählt die Vorlage für die neue Richtung und spult deren Kopie mit `update(elapsed)` an dieselbe zeitliche Stelle vor.

Bei den vorhandenen Assets sind alle Richtungen eines Zustands gleich lang. Eine Gehbewegung kann dadurch beim Drehen ohne Neustart weiterlaufen.

10. Bewegung und einmalige Aktionen verbinden

Datei: `scripts/player.py`

Eine Aktion starten

`play_action()` legt fest, welche Aktionen gestartet oder unterbrochen werden dürfen:

```python
def play_action(self, action):
    if action not in self.ACTIONS:
        raise ValueError(f"Unbekannte Aktion: {action}")
    if self.state == "death" or self.state == action:
        return
    if action == "interact" and self.state in self.ACTIONS:
        return
    self._set_state(action)
```

Ein erneuter Aufruf startet dieselbe laufende Aktion nicht neu. Eine Interaktion unterbricht keine andere Aktion. Eine Verletzung kann eine Interaktion unterbrechen, und der Tod kann beide unterbrechen. Nach dem Tod wird keine weitere Aktion gestartet.

Diese Regeln gelten für den Spieler. Andere Figuren können eigene Unterbrechungsregeln festlegen, beispielsweise für Angriffe.

Die aktive Animation aktualisieren

`Player._animate()` aktualisiert die aktive Animation und reagiert auf ihr Ende:

```python
def _animate(self, dt):
    self.animation.update(dt)
    if self.animation.done and self.state in ("interact", "hurt"):
        self._set_state("idle")
```

`Animation` meldet über `done`, dass das Abspielen fertig ist. `Player` entscheidet dann, bei Interaktion und Verletzung zu `idle` zurückzukehren.

Beim Zustand `death` wird nicht zurückgewechselt. Die einmalige Animation hält selbst ihr letztes Bild fest.

Während einer Aktion stehen bleiben

Am Anfang von `Player.update()` wird geprüft, ob eine einmalige Aktion läuft:

```python
if self.state in self.ACTIONS:
    self._animate(dt)
    return
```

Die Methode endet hier, bevor Bewegungstasten ausgewertet werden. Position und Blickrichtung bleiben während der Aktion fest. Nach einer Interaktion oder Verletzung wird die Bewegung beim nächsten Aufruf von `update()` wieder ausgewertet.

Die Blickrichtung aus WASD bestimmen

Die Tastatur liefert den Bewegungsvektor. `_set_facing()` wählt die Blickrichtung:

```
keys = pygame.key.get_pressed()
direction = pygame.Vector2(keys[pygame.K_d] - keys[pygame.K_a], keys[pygame.K_s] - keys[py

if direction.length_squared() > 0:
    self._set_facing(self.DIRECTIONS[(int(direction.x), int(direction.y))])
    direction = direction.normalize()
```

Bei `D` entsteht `(1, 0)`, also `east`. Bei `W` und `D` entsteht `(1, -1)`, also `northeast`.

Die Zuordnung muss vor dem Normalisieren erfolgen: Danach wäre eine diagonale Richtung ungefähr `(0.707, -0.707)`. Diese Werte passen nicht zu den Dictionary-Schlüsseln. Das Normalisieren sorgt dafür, dass diagonales Gehen nicht schneller ist.

Ohne Bewegungstaste wird die Blickrichtung nicht geändert. Der Schüler behält seine letzte Richtung.

Den Zustand nach Bewegung und Kollision auswählen

Vor der Bewegung wird die alte Position kopiert:

```
previous_pos = self.pos.copy()
movement: pygame.Vector2 = direction * self.speed * dt
```

`copy()` ist wichtig: `previous_pos = self.pos` würde auf denselben veränderbaren Vektor zeigen und keine unabhängige alte Position bewahren.

Anschließend folgen die Bewegung auf der x- und y-Achse sowie die Bildschirm- und Wandkollision. Am Ende von `update()` bestimmt die resultierende Position den Animationszustand:

```
if self.pos != previous_pos:
    self._set_state("run" if self.speed > 150 else "walk")
else:
    self._set_state("idle")
self._animate(dt)
```

Die tatsächliche Bewegung bestimmt den Zustand. Blockiert eine Wand den Schüler vollständig, wird `idle` gewählt. Beim Gleiten entlang einer Wand bleibt er in `walk` oder `run`.

Die Steuerung in `game.py` setzt die Geschwindigkeit auf `150` Pixel pro Sekunde beim Gehen und auf `300` Pixel pro Sekunde bei gedrückter linker Shift-Taste.

11. Das Bild zeichnen und die Interaktion auslösen

Datei: `scripts/player.py`

Die Property `image` liefert das Bild der aktiven Animation:

```
@property
def image(self):
    return self.animation.img()
```

Durch `@property` erfolgt der Zugriff mit `self.image` ohne Klammern.

`render()` zeichnet dieses Bild auf die Spielfläche:

```
def render(self, surface):
    image = self.image
    surface.blit(image, image.get_rect(midbottom=self.rect.midbottom))
```

`blit()` zeichnet das aktuelle Bild auf die Spielfläche. Die untere Mitte der gesamten PNG-Fläche wird an der unteren Mitte der Kollisionsbox ausgerichtet. Transparente Bildränder gehören zur PNG-Fläche.

`PLAYER_SIZE` legt die Größe der Kollisionsbox auf 40 × 40 Pixel fest. Die Bildfläche ist 64 × 64 Pixel groß und wird unabhängig von der Kollisionsbox positioniert.

Der gelbe Kreis im Spiel stellt das Licht der Taschenlampe dar. Er ist unabhängig vom Sprite.

Datei: `game.py` , innerhalb der `KEYDOWN` -Behandlung

```
if event.key == pygame.K_e:
    self.player.play_action("interact")
```

Die Taste `E` startet die Interaktionsanimation. Die Benutzung von Gegenständen oder das Öffnen von Türen erfordert zusätzliche Spiellogik.

Innerhalb von `Game` startet folgender Aufruf die Verletzungsanimation:

```
self.player.play_action("hurt")
```

Die Todesanimation wird mit `death` gestartet:

```
self.player.play_action("death")
```

Die Aufrufe steuern die Animationen. Das Berechnen von Schaden und das Verwalten von Lebenspunkten sind separate Aufgaben der Spiellogik.

12. Dieselbe Klasse für den Hausmeister verwenden

Der Hausmeister hat eigene Zustände. Deshalb bekommt er beim Laden seine eigene Tempotabelle und seine eigenen einmaligen Animationen.

Voraussetzung für das Laden ist ein initialisiertes Pygame mit gesetztem Anzeigemodus durch `pygame.display.set_mode(...)` .

Die Hausmeister-Animationen werden wie folgt geladen und als unabhängige Kopien verwendet:

```python
from scripts.utils import load_animations

# Die vorhandenen Richtungsordner des Hausmeisters.
directions = (
    "south", "southwest", "west", "northwest",
    "north", "northeast", "east", "southeast",
)

hausmeister_animations = load_animations(
    "hausmeister/64/frames",
    animation_fps={
        "idle": 3,
        "walk": 8,
        "alert": 6,
        "attack": 10,
        "hurt": 8,
        "death": 8,
    },
    directions=directions,
    non_looping=("alert", "attack", "hurt", "death"),
)

# Zwei Gegner verwenden dieselben Bilder, aber eigene Abspielzeiten.
gegner_a_animation = hausmeister_animations["walk"]["south"].copy()
gegner_b_animation = hausmeister_animations["walk"]["south"].copy()

gegner_a_animation.update(0.2)

# A zeigt bereits das zweite Bild. B steht noch beim ersten Bild.
assert gegner_a_animation.img() is gegner_a_animation.images[1]
assert gegner_b_animation.img() is gegner_b_animation.images[0]
```

Die Tempi und die Auswahl einmaliger Zustände in diesem Beispiel sind Einstellungen für den Hausmeister, keine Vorgaben der Klasse `Animation`.

Beim Wechsel zum Angriff wird eine neue aktive Kopie ausgewählt:

```python
gegner_a_animation = hausmeister_animations["attack"]["east"].copy()
```

Im Update der Figur wird sie mit der Zeit aus der Spielschleife aktualisiert:

```python
gegner_a_animation.update(dt)
```

Zum Zeichnen wird ihr aktuelles Bild verwendet:

```python
surface.blit(gegner_a_animation.img(), (400, 300))
```

`dt` und `surface` stammen dabei wie beim Spieler aus dem Spielablauf. Die Position ist hier nur ein Beispiel.

Wenn `gegner_a_animation.done` wahr wird, kann die Gegnerlogik entscheiden, ob der Hausmeister wieder steht, weiterläuft oder eine andere Aktion beginnt. Die Animation selbst entscheidet das nicht.

Weitere Figuren verwenden dieselbe Klasse mit ihren jeweiligen Bildern, Tempi, Richtungen und Wiederholungsregeln.

`game.py` erzeugt derzeit nur den Spieler. Für die Darstellung des Hausmeisters müssen seine aktive Animation und Position in den Update- und Renderablauf des Spiels eingebunden werden.

13. Das Ergebnis prüfen

Das Spiel starten

Im Projektordner:

```
venv/bin/python game.py
```

Prüfe nacheinander:

1. Der Schüler steht ohne Eingabe und spielt `idle` ab.
2. Mit WASD geht er in alle vier Hauptrichtungen.
3. Mit Tastenkombinationen wie `W` + `D` geht er diagonal.
4. Beim Drehen während des Gehens läuft die Bildfolge ohne Neustart weiter.
5. Nach dem Anhalten bleibt die letzte Blickrichtung erhalten.
6. Mit linker Shift-Taste und Bewegungstaste verwendet er `run`.
7. Mit `E` spielt er die Interaktion ab und steht dabei still.
8. Die Wandkollision verhindert das Durchlaufen von Wänden.

Die automatischen Tests prüfen zusätzlich die Hausmeister-Bildfolgen und das unabhängige Abspielen mehrerer Kopien.

Die automatischen Tests ausführen

```
venv/bin/python -m unittest discover -s tests -v
```

Die Tests sind auf zwei Dateien verteilt:

Datei	Prüfungen
`tests/test_animation.py`	Zeitsteuerung, Schleifen, vollständige letzte Bilddauer, Kopien, Einzelbilder, ungültige Eingaben, fehlende Bilder sowie beide Figuren mit allen Zuständen und Richtungen
`tests/test_player.py`	WASD, Blickrichtung beim Stehen, Phasenerhalt beim Drehen, Rennen, Aktionen, Tod, Kollision und Sprite-Position

Die Tests verwenden standardmäßig die SDL-Treiber `dummy`, damit kein sichtbares Spielfenster und keine Audioausgabe benötigt werden. Tastenzustände werden mit `unittest.mock.patch` vorgegeben.

Ein erfolgreicher Durchlauf endet mit `Ran 18 tests` und `OK`.

14. Die Bilder beim Weitergeben des Projekts mitnehmen

Die `.gitignore` enthält den Eintrag `data/`. Neue Dateien unter `data/textures/schueler/` und

`data/textures/hausmeister/` werden dadurch nicht automatisch in Git aufgenommen.

Beim Übertragen des Projekts auf einen anderen Rechner müssen die benötigten Bilddateien ebenfalls übertragen werden. Der Lader meldet fehlende Animationsbilder mit dem betroffenen Ordnerpfad.

Der Ablauf eines Spielbildes

```
Game.update()
    |
    ├── Eingaben verarbeiten, etwa E → Player.play_action("interact")
    |
    └── Player.update(walls, dt)
            → Zustand und Blickrichtung bestimmen
            → passende Animationskopie auswählen
            → Animation.update(dt)
            → gegebenenfalls auf animation.done reagieren

Game.render()
    |
    └── Player.render(surface)
            → Animation.img() liefert das aktuelle Bild
            → blit() zeichnet es auf die Spielfläche
```

Die Hausmeisterlogik kann denselben Ablauf für ihre eigenen Entscheidungen verwenden: eine passende Animationskopie wählen, mit `dt` aktualisieren und das aktuelle Bild zeichnen.