miniworldmaker Documentation

Release 2019

Andreas Siebel

Feb 02, 2020

Contents

MiniWorldMaker is a game-engine written in python and pygame designed for pupils to create 2D mini worlds and games.

CHAPTER 1

Miniworldmaker

MiniWorldMaker is a game-engine written in python and pygame designed for pupils to create 2D mini worlds and games.

1.1 Features

- MiniWorldMaker supports pixel-based games as well as games with tiles (e.g. Rogue-Likes)
- Easy creation of animations
- Music and sound effects
- Integrated GUI elements like console for output, toolbar, . . .
- Load and Save to SQLite Databases
- EXPERIMENTAL: Integrated Physics-Engine based on Pymunk
- Open Source
- Miniworldmaker is a 2D Engine based on **Python 3** and **pygame**.
- Processing Mode
- More Infos: Github | Documentation | PyPi | Status

CHAPTER 2

Tutorials

2.1 Basics Tutorial

2.1.1 Tutorial - English

Installation

Install the miniworldmaker framework with

```
pip install miniworldmaker
```

The playing field

Here we go!

A first world

We create the first world. This works with the following code:

```
from miniworldmaker import *


class MyBoard(TiledBoard):

    def setup(self):
        self.columns = 20
        self.rows = 8
        self.tile_size = 42
        self.add_image(path="images/soccer_green.jpg")
```

(continues on next page)

(continued from previous page)

```
board = MyBoard()
board.show()
```

First a new *class* MyBoard is created. This is a child class of TiledBoard and allows you to build all sorts of games based on tiles.

- Line 1: The **import** statement imports the miniworldmaker library.
- Line 4: The own playing field is created as child class of the class Tiledboard.
- Line 6: The setup() method is called when a new object is created (i.e. here in line 7).
- Lines 7-9: The size of the playing field is initialized.
- Line 7: A background is added to your board. Make sure that the file is on the specified path.

These two lines:

```
board = MyBoard()
board.show()
```

The last two lines of your program are always similar: Here the MyBoard() command creates a concrete playing field, and then the board.show() instructs the board to show itself.

Depending on your background image, the result will look like this:

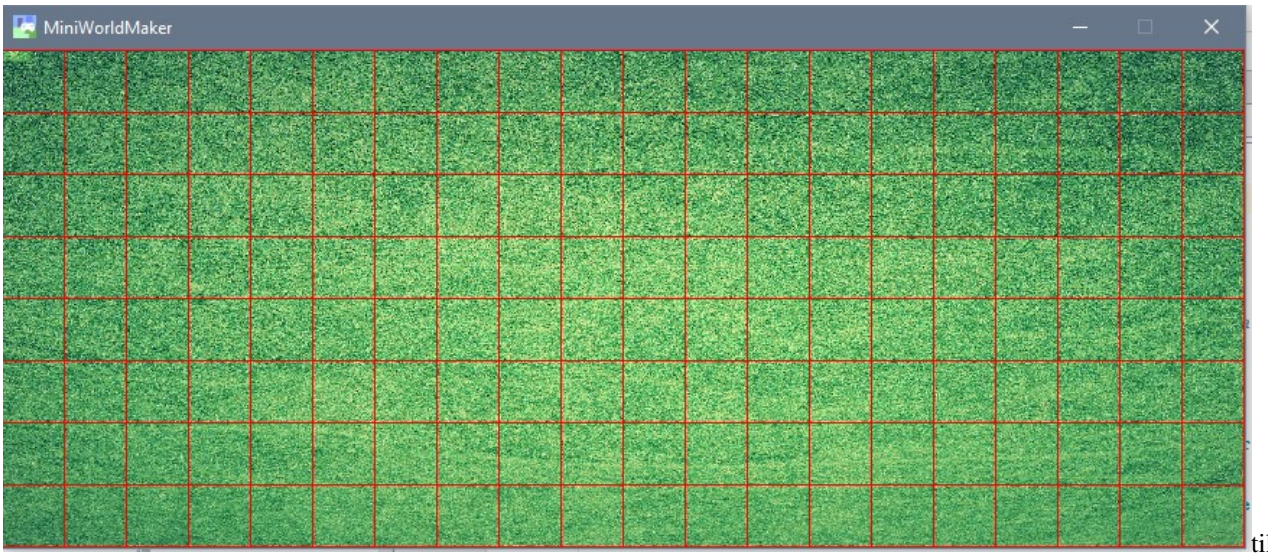
tiles

Show the grid

If you like, you can also have the borders of the individual tiles displayed. Change the method setup() in the class MyBoard:

```
    def setup()
        ...
        self.background.grid_overlay = True
```

That's how it looks:



tiles

PixelBoards and TiledBoards

There are several subclasses of the class board:

- A PixelGrid is intended for pixel-precise representation of content.
- A TiledBoard is intended for boards where the actors move on square tiles.

Most of the functions differ only slightly, since both boards are subclasses of the class **Boards**.

. . . inheritance-diagram:: miniworldmaker.boards.pixel_board.PixelBoard miniworldmaker.boards.tiled_board.TiledBoard

top-classes miniworldmaker.tokens.boards.board

parts 1

Translated with www.DeepL.com/Translator

Tokens

Create a new Token class

Next, an token is placed on the board.

This is done as follows

```
class Player(Token):

    def setup(self):
        self.add_image(path="images/char_blue.png")
```

- Line 1 creates a new class as a child class of the class Token.
- Line 3 defines the setup() method, which is called when a new Player object is created.
- A picture is then added to the Player object in line 4.

Add the token to the playing field

So far we have only created one template to create player objects.

Now we want to create concrete objects and add them to the board. Add the setup() method of the playing field class:

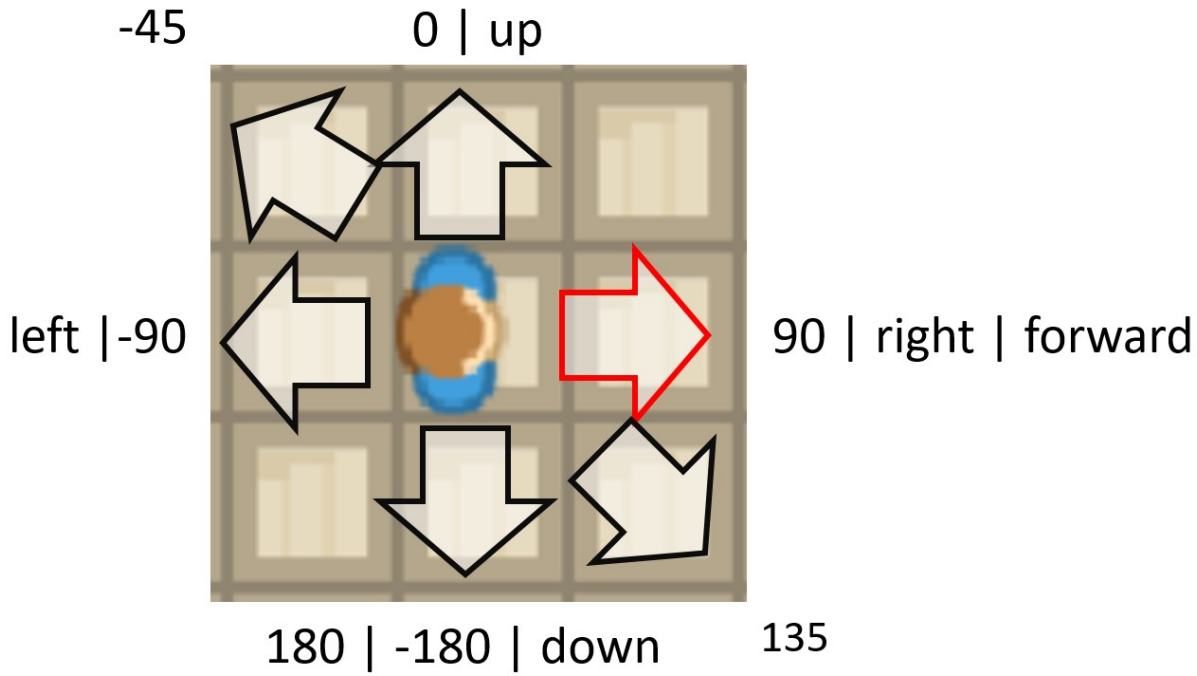
```
class MyBoard(TiledBoard):

    def setup(self):
        ...
        player1 = Player(position = (3, 3))
```

Translated with www.DeepL.com/Translator (free version)

Directions

Angles

An actor can move in different directions. First you need to know how angles are interpreted in Miniworldmaker. Angles are independent of the orientation of the character:



movement

- 0° means a movement upwards.
- 90° means a movement to the right..
- 180° oder - 180° means a movement downwards.
- -90° means a movement to the right.

The interpretation of directions corresponds to the popular programming language Scratch, see https://en.scratch-wiki.info/wiki/Direction_(value)

There is one exception: The default direction in Miniworldmaker is 0°, i.e. tokens point upwards.

Angles as Strings

Some angle sizes can also be called strings:

- right": is equivalent to 0°.
- "up" is equivalent to 90°.
- "left" is equivalent to 180°.
- Down is equivalent to 270 degrees.

A special specification is "forward": In contrast to the other specifications, "forward" means in the direction of the figure's gaze. In the picture above "forward" corresponds to 0°, because the actor looks to the right.

Methods and Attributs

You can use the following functions to change the alignment of an actor:

self.direction

Sets the direction directly.

self.turn_left

Turns the actor in left direction.

self.turn_right

Turns the actor in right direction.

self.flip_x

The actor rotates 180°. The figure is mirrored so that the actor is not upside down after the rotation.

Movements

Movement

The central function for moving is the move function.

Move has the following signature:

```
    def move(distance) -> BoardPosition:
```

This means

- By default, an actor moves **self.speed** steps in the direction he's looking.
- You can also set the distance it moves manually by using an integer value for the parameter distance.
- The function returns the position on the playing field where the player is after the move.

Methoden und Attribute

Moves an token

Sensing

Active tracking

An actor can track down whether he is at his position or before other actors and so on.

This can be done with the following function, for example:

```
actor.sensing_tokens(distance, token)
```

The function detects whether there are actors at the current position of the actor (or distance steps forward). If so, they are returned as a list, otherwise None is returned.

Example

The example checks whether the actor is standing in front of a locked door:

```
actors_in_front = self.sensing_tokens(distance = 1, token = door)
        if self.board.door in actors_in_front:
            if self.board.door.closed:
                message = "The Door is closes"
```

Sensing via event methods

Alternatively, you can also implement event methods: The on_sensing_xy method is called every time an actor detects something.

–> see also events

Functions to find objects

Sensing Tokens

Tracks tokens

. . . autoclass:: miniworldmaker.tokens.token.Token.Token

members sensing_tokens

noindex

Sensing Token

Tracks a single token. The method is more efficient than sensing_tokens

. . . autoclass:: miniworldmaker.tokens.token.Token.Token

members sensing_token

noindex

Sensing Border

Checks if there's an edge nearby.

. . . autoclass:: miniworldmaker.tokens.token.Token.Token

members sensing_borders

noindex

Sensing Border

Checks if the position is on the playing field.

. . . autoclass:: miniworldmaker.tokens.token.Token.Token

members sensing_on_board

noindex

Sensing Color

Checks the color under the actor (related to the background of the game world)

. . . autoclass:: miniworldmaker.tokens.token.Token.Token

members sensing_on_board

noindex

Translated with www.DeepL.com/Translator

events

There are several important methods that you can fill with content:

Special Event Methods

The act() method

The Act method is called again and again at short intervals. Here you can place code that should be executed over and over again, e.g:

```
    def act(self):
        if not self.look_on_board(direction = "forward"):
            self.turn_left(90)
        self.move()
```

The Actor looks one square forward and checks whether it is still on the playing field. If so, he moves one square forward. Otherwise he turns 90° to the left.

Sensing methods

Each time an actor detects something different, its corresponding sensing methods are called:

on_sensing_tokens(token_list): Tracks all tokens at the same location. **on_sensing_borders(borders)**: Returns a list of borders that are currently touched (e.g. ["right", "top"]). **on_sensing_on_board**(): Called when the player is on the field. **on_sensing_not_on_board**(): Called when the player is not on the field.

General event handling with the get_event method

The get_event(event, data) method

The get_event(event, data) method) is used to react to events of various kinds.

- The parameter **event** always contains a string with the event, for example:
  - mouse-left": The left mouse button was pressed.
  - ...
- The **data** parameter contains information that matches the event.
  - mouse-left", "mouse-right" : The tile on the board that was clicked.

Example:

```
    def get_event(self, event, data):
        if event == "key_down":
            if "W" in data:
                self.move(direction="up")
            elif "S" in data:
                self.move(direction="down")
            elif "A" in data:
                self.move(direction="left")
            elif "D" in data:
                self.move(direction="right")
```

The code checks whether the event "key_down" was thrown. The parameter data returns a list with all buttons pressed at this moment. It can therefore be checked: **If* the button X was pressed, do ...

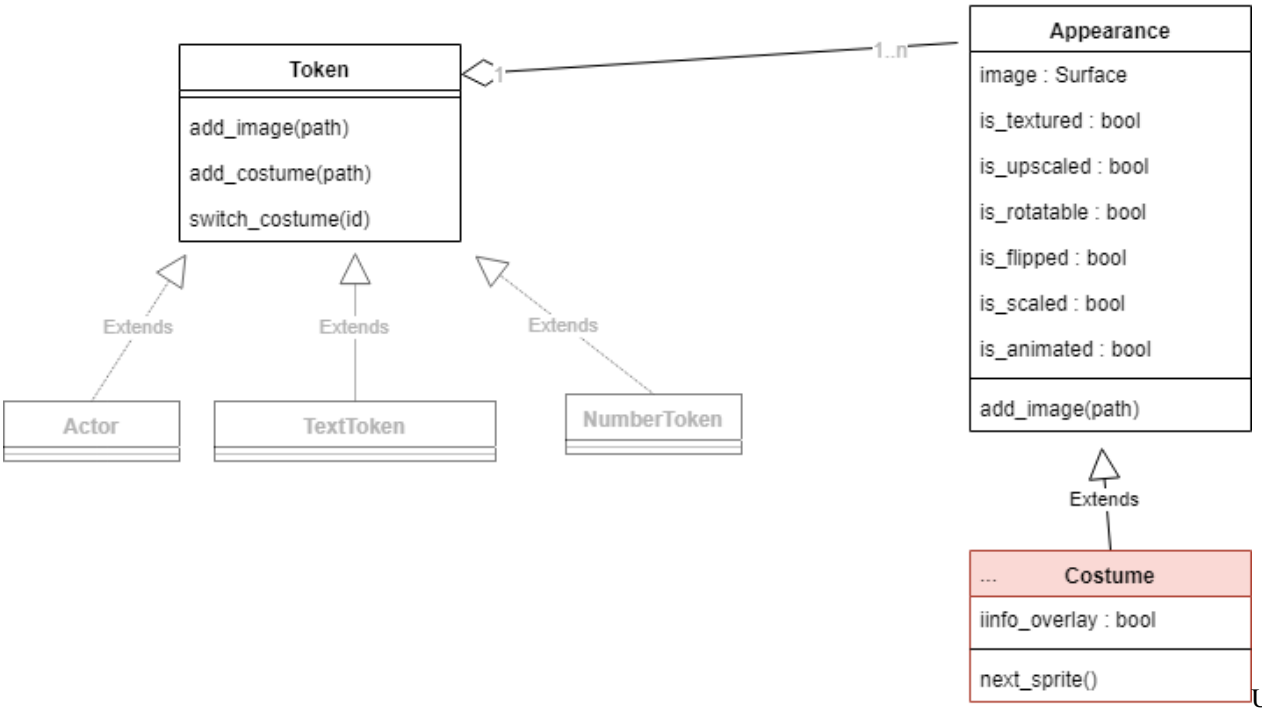
Translated with www.DeepL.com/Translator

Costumes and Background

Costumes

Each actor has one or more costumes.

A costume consists of one or more pictures and instructions on how to display them.



UML Diagram

Add images to a costume:

Add a new image to a costume:

```
self.add_image("image.jpg")
```

In the same way you can also add several pictures to a costume:

```
self.add_image("image1.jpg")
self.add_image("image2.jpg")
```

Representation of pictures

The following instructions change the appearance of costumes:

Info Overlay

Shows an info overlay with frame and direction above the token.

is_rotatable

Specifies whether the image is rotated with the direction of the actor.

is_upscaled

Specifies whether the image should be upscaled to the size of the token. This action maintains the ratio between length and width.

is_scaled

Specifies whether the image should be upscaled to the size of the token. This action may change the size ratio between length and width.

is_textured

In addition to scale and upscale, there is also the option to "wallpaper" the background with an image for backgrounds.

Create multiple costumes

You can create several costumes as follows:

```
my_costume = self.add_costume("image.png")
```

A new costume is created with the image image.png. You can also add more pictures to the costume:

```
my_costume.add_image("image2.png")
```

Switching between costumes

Here's how you switch between two costumes:

```
self.switch_costume()
```

The instruction jumps to the next costume. You can also specify a number as a parameter to jump to a specific costume.

The background

The playing field has a background. Many actions work similar to the costume, but there are some actions that only make sense for the background:

Show Grid Overlay

Displays margins for all cells.

Animations

You can imagine 2D animations like a flip book.

The fact that the image of an actor/token is changed quickly one after the other makes it seem as if the actor is moving.

Here's how you can create animations:

1. add images

Simply add multiple images in the **init**() method:

```
    def __init__(self):
        super().__init__()
        self.add_image("images/robot_blue1.png")
        self.add_image("images/robot_blue2.png")
```

Start second animation

Set the speed and start the animation:

```
    def __init__(self):
        super().__init__()
        self.add_image("images/robot_blue1.png")
        self.add_image("images/robot_blue2.png")
        [...]
        self.costume.animation_speed = 30
        &lt;font color="#ffff00"&gt;-===- proudly presents
```

In the last two lines it is indicated that the costume of the actor should change with the speed 30 (try different values here) and at the very end the animation will start.

See also the example roboanimation on github:

_images/roboanimation.gif

GUI elements

You can offer different GUI elements to your miniworld.

Some of the GUI elements help you as designer and programmer, others you can integrate as user interface directly into your miniworld.

The Toolbar

The toolbar is a list of buttons.

You initialize a new toolbar like this:

```
        toolbar = self.window.add_container(Toolbar(), dock="right")
        toolbar.add_widget(ToolbarButton("Spin"))
```

You can add the following widgets at this time:

- ToolbarButton: A button. When you click on the button, the Event button will be called. For data the text of the button is provided (in the example above e.g. “Spin”).
- A label is used to display text.

See also the example spinning_wheel on Github.

_images/roboanimation.gif

The console

The console is used to output information.

The console is initialized as follows.

```
        self.console = self._window.add_container(Console(), "bottom")
```

Then you can write output to the console as follows:

```
                self.board.console.print("You're lighting the fire.")
```

The Event Console

The Event bar is a special console that outputs events that are sent. You can initialize the console as follows:

```
        self.window.add_container(event_console, dock="right", size=400)
```

You will find that the output is quickly confusing. So you can decide what kind of events you want the console to respond to.

```
        event_console.register_events = {"key pressed"}
```

This line of code means, for example, that the console only responds to the key_pressed event.

The action bar

With the Action Bar you can analyze the program flow by running the program step by step. This is how you initialize the Actionbar:

```
        self.window.add_container(ActionBar(self), dock="bottom")
```

The ActiveActorToolbar

The ActiveActorToolbar shows you all current information about an actor (position, direction, borders touched, . . .).

You can initialize the token toolbar as follows:

```
        actor_toolbar = ActiveActorToolbar(self)
        self.window.add_container(actor_toolbar, dock="right", size=400)
```

The actor that is displayed with the toolbar can be selected by clicking on the corresponding actor.

physics

MiniWorldmaker has an integrated physics environment.

To physically simulate an object, you must overwrite the method **setup_physics()**.

Example:

```
class Paddle(Rectangle):
    def setup(self):
        self.size = (10, 80)
        self.costume.is_rotatable = False

    def setup_physics(self):
        self.physics.stable = True
        self.physics.can_move = True
        self.physics.mass = "inf"
        self.physics.friction = 0
        self.physics.gravity = False
        self.physics.elasticity = 1
```

If the method is implemented, the physics engine is initialized before executing the setup() method. Once the engine is initialized, you can "push" objects. This works like this:

```
class Ball(Circle):

    def on_setup(self):
        self.direction = 30
        self.physics.impulse_in_direction(300)
```

or like this:

```
class Bird(Actor):

    def on_setup(self):
        ...
        self.physics.velocity_x = 600
        self.physics.velocity_y = - self.board.arrow.direction * 50
```

2.1.2 Tutorial - Deutsch

Installation

Installiere das Framework mit:

```
pip install miniworldmaker
```

Das Spielfeld

Los geht es!

Eine erste Welt

Wir erschaffen die erste Welt. Dies geht mit folgendem Code:

```
from miniworldmaker import *


class MyBoard(TiledBoard):

```

(continues on next page)

(continued from previous page)

```
    def setup(self):
        self.columns = 20
        self.rows = 8
        self.tile_size = 42
        self.add_image(path="images/soccer_green.jpg")

board = MyBoard()
board.show()
```

Zunächst wird eine eigene *Klasse* MyBoard erstellt. Diese ist eine Kindklasse von TiledBoard und erlaubt es dir, alle möglichen Spiele zu bauen, die auf Tiles basieren.

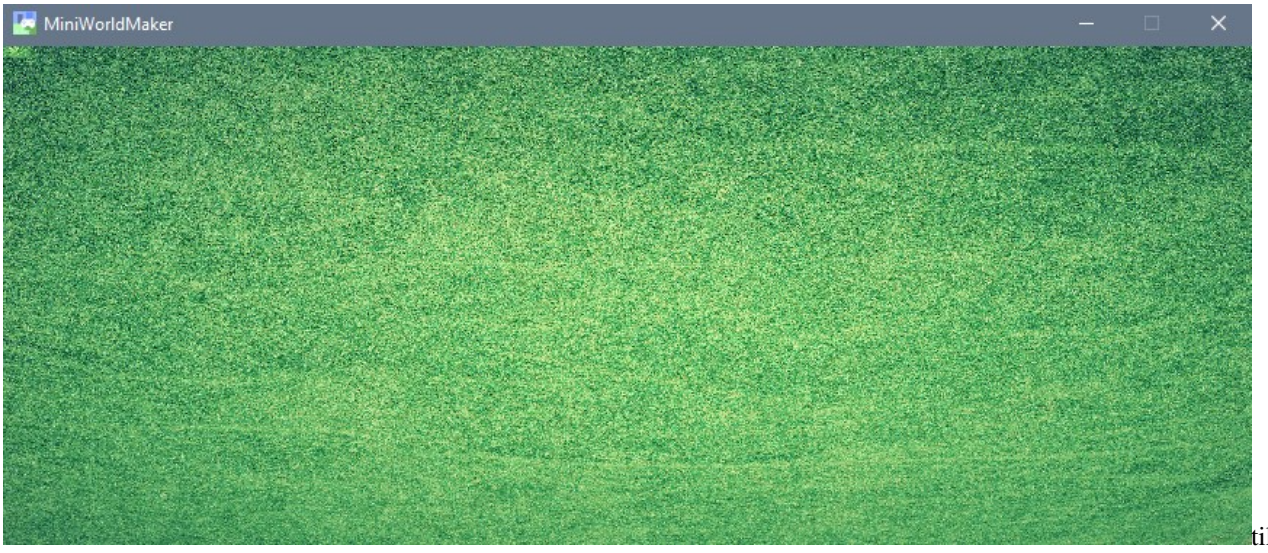
- Zeile 1: Mit der **import** Anweisung wird die Bibliothek miniworldmaker importiert.
- Zeile 4: Das eigene Spielfeld wird als Kindklasse der Klasse Tiledboard erstellt.
- Zeile 6: Die setup() - Methode wird bei Erstellen eines neuen Objektes aufgerufen (d.h. hier in Zeile 7).
- Zeile 7-9: Die Größe des Spielfeldes wird initialisiert.
- Zeile 7: Deinem Board wird ein Hintergrund hinzugefügt. Achte darauf, dass die Datei an dem angegebenen Pfad liegt.

Diese beiden Zeilen:

```
board = MyBoard()
board.show()
```

Sind so ähnlich immer die letzten beiden Zeilen deines Programms: Hier wird mit dem Befehl MyBoard() ein konkretes Spielfeld erzeugt und anschießend wird mit board.show() das Board angewiesen, sich zu zeigen.

Je nach Hintergrundbild sieht das Ergebnis bei dir so aus:



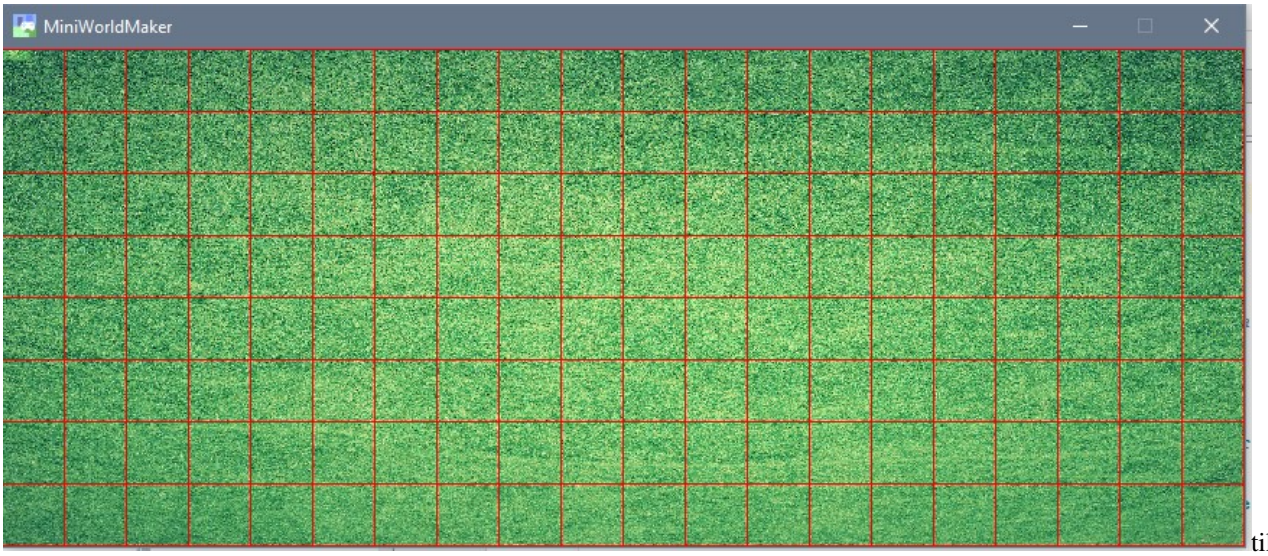
tiles

Das Grid anzeigen

Wenn du möchtest kannst du dir auch die Grenzen der einzelnen Tiles anzeigen lassen. Ändere dazu die Methode setup() in der Klasse MyBoard ab:

```
    def setup()
        ...
        self.background.grid_overlay = True
```

So sieht es dann aus:



tiles

PixelBoards und TiledBoards

Es gibt verschiedene Unterklassen der Klasse Board:

- Ein PixelGrid ist für Pixelgenaue Darstellung von Inhalten gedacht.
- Ein TiledBoard ist für Boards gedacht, bei denen sich die Akteure auf quadratischen Kacheln bewegen.

Die meisten der Funktionen unterscheiden sich nur geringfügig, da beide Boards Unterklassen der Klasse **Boards** sind.

Tokens

Eine neue Token-Klasse erstellen

Als nächtes wird ein Token, d.h. eine Spielfigur auf dem Board platziert.

Dies geht so:

```
class Player(Token):

    def setup(self):
        self.add_image(path="images/char_blue.png")
```

- In Zeile 1 wird eine neue Klasse als Kindklasse der Klasse Token definiert.
- In Zeile 3 wird die setup()-Methode definiert, welche beim Erstellen eines neuen Player-Objektes aufgerufen wird.
- In Zeile 4 wird dann zu dem Player-Objekt ein Bild hinzugefügt.

Das Token zum Spielfeld hinzufügen

Bis jetzt haben wir nur eine Schablone erstellt, um Player-Objekte zu erzeugen.

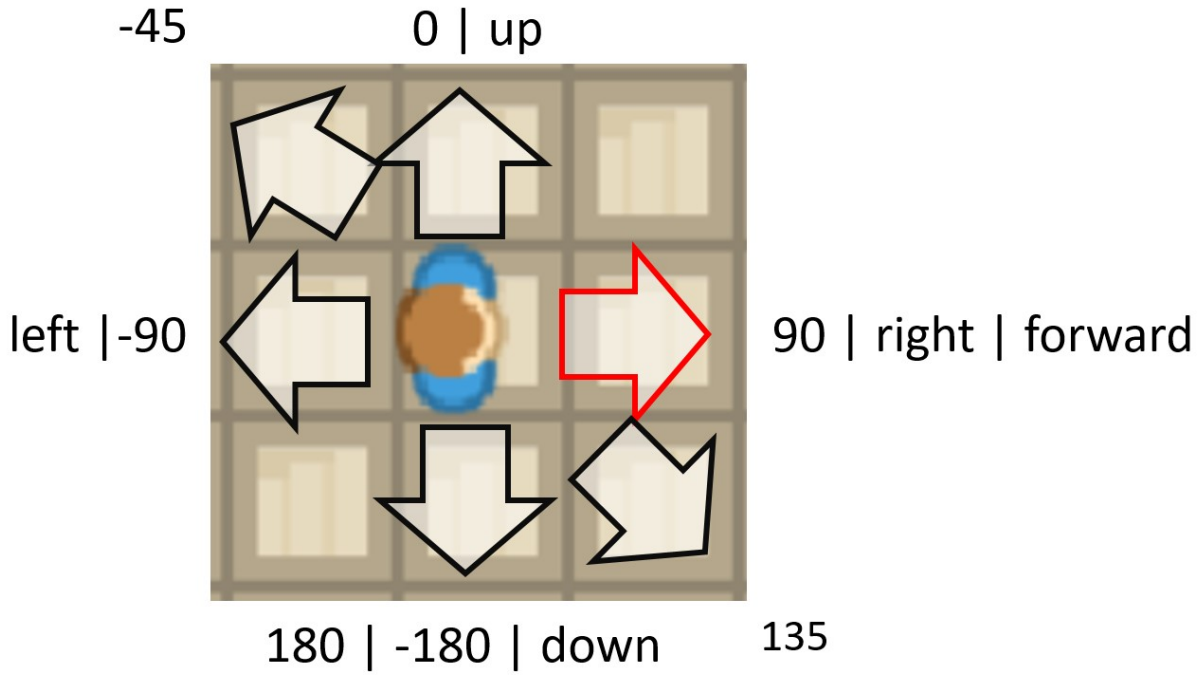
Jetzt sollen konkrete Objekte erzeugt und zum Spielfeld hinzugefügt werden. Ergänze dazu die setup() - Methode der Board-Klasse, die du zuvor erstellt hast:

```
class MyBoard(TiledBoard):

    def setup(self):
        ...
        player1 = Player(position = (3, 3))
```

Richtungen

Winkel

Ein Token kann sich in verschiedene Richtungen bewegen. Zunächst musst du dazu wissen, wie in Miniworldmaker Winkel interpretiert werden. Winkel sind unabhängig von der Ausrichtung der Spielfigur:



movement

- 0° bedeutet eine Bewegung nach oben.
- 90° eine Bewegung nach rechts.
- 180° oder - 180° bedeutet eine Bewegung nach unten
- -90° bedeutet eine Bewegung nach links

Die Interpretation von Richtungen entsprechen der populären Programmiersprache Scratch, siehe https://en.scratch-wiki.info/wiki/Direction_(value)

Es gibt eine Ausnahme: Die Default-Direction ist in Miniworldmaker 0°, d.h. Tokens zeigen nach oben.

Winkel als Strings

Einige Winkelgrößen kannst du auch mit Strings bezeichnen:

- "right": ist äquivalent zu 0°.
- "up" ist äquivalent zu 90°.
- "left" ist äquivalent zu 180°.
- "down" ist äquivalent zu 270°.

Eine spezielle Angabe ist "forward": Im Gegensatz zu den anderen Angaben bedeutet "forward" in Blickrichtung der Figur. Im Bild oben entspricht "forward" 0°, da der Akteur nach rechts schaut.

Methoden und Attribute

self.direction

Setzt die Richtung des Akteurs.

self.turn_left

Dreht den Akteur nach links.

self.turn_right

Dreht den Akteur nach rechts.

self.flip_x

Der Akteur wird über eine zentrale y-Achse gespiegelt.

Bewegungen

Die Move-Funktion

Die zentrale Funktion zum Bewegen ist die Funktion move

Move hat folgende Signatur:

```
    def move(distance) -> BoardPosition:
```

Dies bedeutet:

- Standardmäßig bewegt sich ein Akteur um **self.speed** Schritte in die Richtung in die er gerade schaut.
- Du kannst die Distanz die er sich bewegt aber auch manuell festlegen, indem du für den Paramter distance einen Integer-Wert einsetzt.
- Die Funktion gibt als Rückgabewert die Position auf dem Spielfeld zurück, an der sich der Akteur nach dem Zug befindet.

Methoden und Attribute

Bewegt ein Akteur.

Aufspüren

Aktives Aufspüren

Ein Akteur kann aufspüren, ob sich an seiner Position oder vor ihm andere Akteure usw. befinden.

Dies geht z.B. mit folgender Funktion:

```
actor.sensing_tokens(distance, token)
```

Die Funktion spürt auf, ob sich an der aktuellen Position des Actors (oder distance Schritte nach vorne) Akteure befinden. Wenn ja, dann werden diese als Liste zurückgegeben, andernfalls wird None zurückgegeben.

Beispiel

In dem Beispiel wird überprüft, ob der Akteur vor einer verschlossenen Tür steht:

```
actors_in_front = self.sensing_tokens(distance = 1, token = Door)
        if self.board.door in actors_in_front:
            if self.board.door.closed:
                message = "The Door is closes"
```

Aufspüren über Event-Methoden

Alternativ kann man auch Event-Methoden implementieren: Die Methode on_sensing_xy wird aufgerufen, jedesmal dann, wenn ein Akteur etwas bestimmtes aufspürt.

–> Siehe dazu auch events

Funktionen zum Aufspüren von Objekten

Sensing Tokens

Spürt Tokens auf

Sensing Token

Spürt ein einzelnes Token auf. Die Methode ist effizienter als sensing_tokens

Sensing Border

Prüft, ob ein Rand in der Nähe ist.

Sensing Border

Prüft, ob die Position auf dem Spielfeld ist.

Sensing Color

Prüft die Farbe unter dem Actor (bezogen auf den Background der Spielwelt)

Ereignisse

Es gibt mehrere wichtige Methoden, die du mit Inhalten füllen kannst:

Spezielle Event-Methoden

Die act() - Methode

Die Act-Methode wird in kurzen Abständen immer wieder aufrufen. Hier kannst du Code platzieren, der immer wieder ausgeführt werden soll, z.B.:

```
    def act(self):
        if not self.look_on_board(direction = "forward"):
            self.turn_left(90)
        self.move()
```

Der Actor schaut ein Feld nach vorne und überprüft, ob dieses noch auf dem Spielfeld liegt. Wenn ja, geht er ein Feld vorwärts. Andernfalls dreht er sich um 90° nach links.

Sensing-Methoden

Jedesmal, wenn ein Akteur etwas anderes aufspürt, werden seine entsprechenden sensing-Methoden aufgerufen:

- **on_sensing_tokens(token_list)**: Spürt alle Tokens am selben Ort auf.
- **on_sensing_borders(borders)**: Gibt eine Liste mit Rändern zurück, die zur Zeit berührt werden (z.B. [“right”, “top”]
- **on_sensing_on_board()**: Wird aufgerufen, wenn der Akteur sich auf dem Spielfeld befindet.
- **on_sensing_not_on_board()**: Wird aufgerufen, wenn sich der Akteur nicht auf dem Spielfeld befindet.

Allgemeines Event-Handling mit der get_event-Methode

Die get_event(event, data)-Methode

Die get_event(event, data)-Methode) dient dazu, auf Ereignisse verschiedener Art zu reagieren.

- Der Parameter **event** enthält immer einen String mit dem Ereignis, z.B.:
  - “mouse-left”: Es wurde die linke Maustaste gedrückt.
  - ...
- Der Parameter **data** enthält Infos, die zu dem Ereignis passen.

- “mouse-left”, “mouse-right” : Die Kachel im Board auf die geklickt wurde.

Beispiel:

```
    def get_event(self, event, data):
        if event == "key_down":
            if "W" in data:
                self.move(direction="up")
            elif "S" in data:
                self.move(direction="down")
            elif "A" in data:
                self.move(direction="left")
            elif "D" in data:
                self.move(direction="right")
```

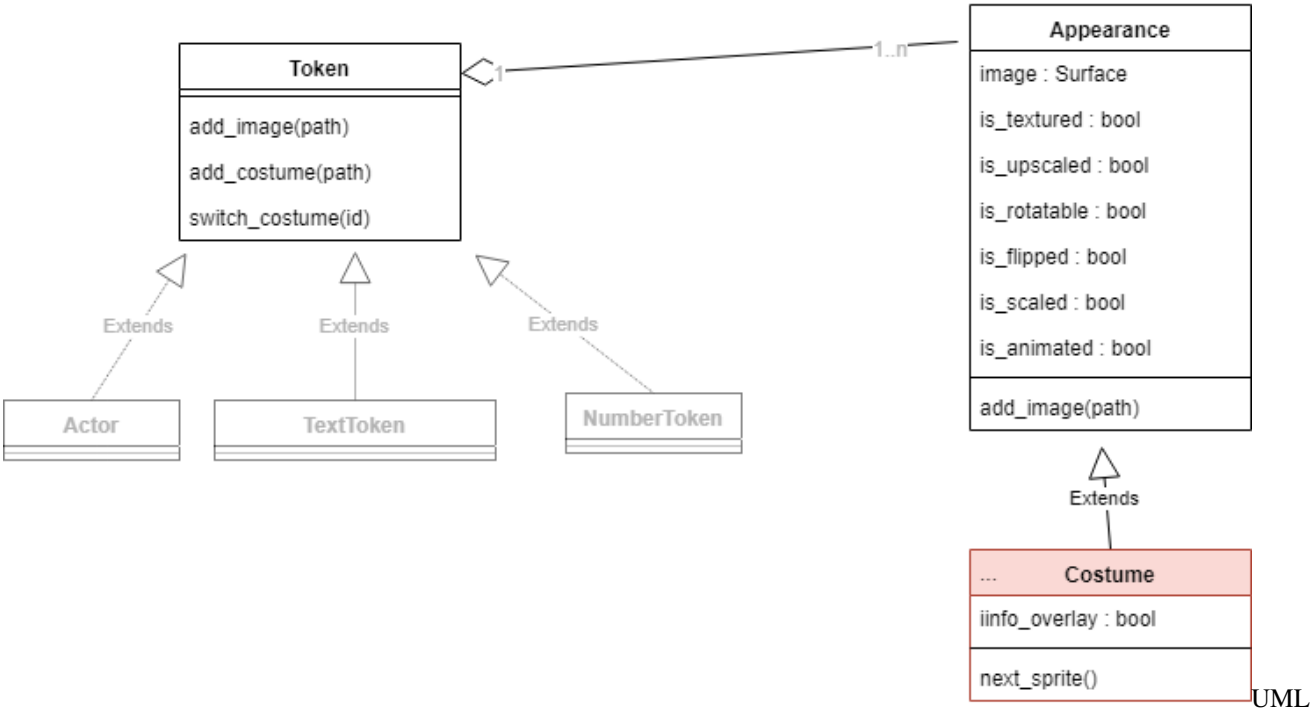
Der Code überprüft, ob das Ereignis “key_down geworfen wurde. Im Parameter data wird hier eine Liste mit allen in diesem Moment gedrückten Buttons zurückgegeben. Es kann daher überprüft werden: **Wenn* der Button X gedrückt wurde, tue . . .

Kostüme und Hintergrund

Kostüme

Jedes Token und jeder Akteur hat ein oder mehrere Kostüme.

Ein Kostüm besteht aus einem oder mehreren Bildern und Anweisungen, wie diese Bilder dargestellt werden sollen.



UML Diagramm

Bilder zu einem Kostüm hinzufügen:

Ein neues Bild zu einem Kostüm hinzufügen:

```
self.add_image("image.jpg")
```

Auf die gleiche Art kannst du auch mehrere Bilder zu einem Kostüm hinzufügen:

```
self.add_image("image1.jpg")
self.add_image("image2.jpg")
```

Darstellung von Bildern

Folgende Anweisungen ändern die Darstellung von Kostümen:

Info Overlay

Zeigt ein Info-Overlay mit Rahmen und Richtung über dem Token

is_rotatable

Gibt an, ob das Bild mit der Richtung des Actors mitgedreht wird.

is_upscaled

Gibt an, ob das Bild auf die Größe des Tokens hochskaliert werden soll. Diese Aktion behält das Größenverhältnis zwischen Länge und Breite bei.

is_scaled

Gibt an, ob das Bild auf die Größe des Tokens hochskaliert werden soll. Diese Aktion verändert gegebenenfalls das Größenverhältnis zwischen Länge Breite.

is_textured

Neben scale und upscale gibt es für Hintergründe auch die Option, den Hintergrund mit einemm Bild zu “tapezieren”

Mehrere Kostüme anlegen

Du kannst folgendermaßen mehrere Kostüme anlegen:

```
my_costume = self.add_costume("image.png")
```

Es wird ein neues Kostüm mit dem Bild image.png angelegt. Du kannst auch weitere Bilder zu dem Kostüm hinzufügen:

```
my_costume.add_image("image2.png")
```

Zwischen Kostümen wechseln

Folgendermaßen wechselst du zwischen zwei Kostümen:

```
self.switch_costume()
```

Die Anweisung springt zum nächsten Kostüm. Du kannst als Parameter auch eine Zahl angeben, um zu einem bestimmten Kostüm zu springen.

Der Hintergrund

Das Spielfeld hat einen Hintergrund. Viele Aktionen funktionieren ähnlich wie bei dem Kostüm, allerdings gibt es noch folgende Aktionen, die nur für den Hintergrund Sinn machen:

Grid Overlay anzeigen

Zeigt für alle Zellen Ränder an.

Animationen

2D-Animationen kannst du dir vorstellen wie ein Daumenkino.

Dadurch, dass schnell hintereinander das Bild eines Akteurs/Token geändert wird, macht es den Anschein, als würde sich der Akteur bewegen.

Folgendermaßen kannst du Animationen erstellen:

1. Bilder hinzufügen

Füge in der **init**()-Methode einfach mehrere Bilder hinzu:

```
    def __init__(self):
        super().__init__()
        self.add_image("images/robot_blue1.png")
        self.add_image("images/robot_blue2.png")
```

2. Animation starten

Lege die Geschwindigkeit fest und starte die Animation:

```
    def __init__(self):
        super().__init__()
        self.add_image("images/robot_blue1.png")
        self.add_image("images/robot_blue2.png")
        [...]
        self.costume.animation_speed = 30
        self.costume.is_animated = True
```

In den letzten beiden Zeilen wird angegeben, dass das Kostüm des Actors mit der Geschwindigkeit 30 wechseln soll (probiere hier unterschiedliche Werte aus) und ganz am Ende wird die Animation gestartet.

Schaue dir dazu auch das Beispiel roboanimation auf github an:

_images/roboanimation.gif

GUI Elemente

Du kannst an deine Miniwelt verschiedene GUI-Elemente anbieten.

Einige der GUI-Elemente helfen dir als Designer und Programmierer, andere kannst du als Bedienfläche direkt in deine Miniwelt einbauen.

Die Toolbar

Die Toolbar ist eine Liste von Schaltflächen.

Eine neue Toolbar initialisierst du so:

```
        toolbar = self.window.add_container(Toolbar(), dock="right")
        toolbar.add_widget(ToolbarButton("Spin"))
```

Folgende Widgets kannst du zur Zeit hinzufügen:

- ToolbarButton: Ein Button. Beim Klick auf den Button wird das Event Button aufgerufen. Für data wird der Text des Buttons mitgeliefert (Im Beispiel oben z.B. “Spin”)
- Ein Label dient dazu Text anzuzeigen.

Siehe auch das Beispiel spinning_wheel auf Github.

_images/roboanimation.gif

Die Konsole

Die Konsole dient dazu Informationen auszugeben.

Die Konsole wird folgendermaßen initialisiert.

```
        self.console = self._window.add_container(Console(), "bottom")
```

Anschließend kannst du folgendermaßen Ausgaben auf die Konsole schreiben:

```
                self.board.console.print("Du zündest die Feuerstelle an.")
```

Die Event-Console

Die Event-Leiste ist eine spezielle Console, welche Ereignisse ausgibt, die gesendet werden. Du kannst die Konsole folgendermaßen initialisieren:

```
        self.window.add_container(event_console, dock="right", size=400)
```

Du wirst feststellen, dass die Ausgabe aber schnell unübersichtlich ist. Daher kannst du entscheiden, auf welche Art von Ereignissen die Konsole reagieren soll.

```
        event_console.register_events = {"key pressed"}
```

Diese Zeile Code bedeutet z.B., dass die Konsole nur auf das Ereigniss key_pressed reagiert.

Die Actionbar

Mit der Actionbar kannst du den Programmfluss analysieren, indem du das Programm schrittweise ablaufen lässt. So initialisierst du die Actionbar:

```
        self.window.add_container(ActionBar(self), dock="bottom")
```

Die ActiveActorToolbar

Diese Toolbar zeigt dir alle aktuellen Informationen über einen Akteur (Position, Richtung, berührte Ränder, . . .) an,

Du kannst die Token-Toolbar folgendermaßen initialisieren:

```
        actor_toolbar = ActiveActorToolbar(self)
        self.window.add_container(actor_toolbar, dock="right", size=400)
```

Der Actor, der mit der Token-Toolbar angezeigt wird, kann durch Klick auf den entsprechenden Actor ausgewählt werden.

Physik

MiniWorldmaker hat eine integrierte Physik-Umgebung.

Damit ein Objekt physikalisch simuliert wird, musst du die Methode **setup_physics()** überschreiben.

Beispiel:

```
class Paddle(Rectangle):
    def setup(self):
        self.size = (10, 80)
        self.costume.is_rotatable = False

    def setup_physics(self):
        self.physics.stable = True
        self.physics.can_move = True
        self.physics.mass = "inf"
        self.physics.friction = 0
        self.physics.gravity = False
        self.physics.elasticity = 1
```

Wenn die Methode implementiert ist, wird die Physik-Engine vor Ausführung der setup()-Methode initialisiert. Sobald die Engine initialisiert ist kannst du Objekte “anschubsen”. Dies geht folgendermaßen:

```
class Ball(Circle):

    def on_setup(self):
        self.direction = 30
        self.physics.impulse_in_direction(300)
```

oder folgendermaßen:

```
class Bird(Actor):

    def on_setup(self):
        ...
        self.physics.velocity_x = 600
        self.physics.velocity_y = - self.board.arrow.direction * 50
```

2.2 Graphics Programming

2.2.1 Processing_Tutorial

Installation

Installiere das Framework mit:

```
pip install miniworldmaker
```

Erste Schritte

Kopiere zunächst das folgende Rahmenprogramm in deinen Code-Editor:

```
from miniworldmaker import *

class MyBoard(ProcessingBoard):
    def on_setup(self):
        pass

my_board = MyBoard(600, 400)
my_board.show()
```

Dieses Rahmenprogramm macht folgendes:

- Im oberen Teil des Quellcodes wird eine eigene Klasse (d.h. ein Bauplan) für deine eigene Miniwelt erstellt. Es handelt sich um ein spezielles ProcessingBoard.
- Im unteren Teil wird von diesem Bauplan ein neues Fenster mit Breite 600 und Höhe 400 erstellt.

Das erste Bild

Du kannst Bilder erstellen, indem du in die setup()-Methode Grafikobjekte hinzufügst:

Dies geht z.B. so:

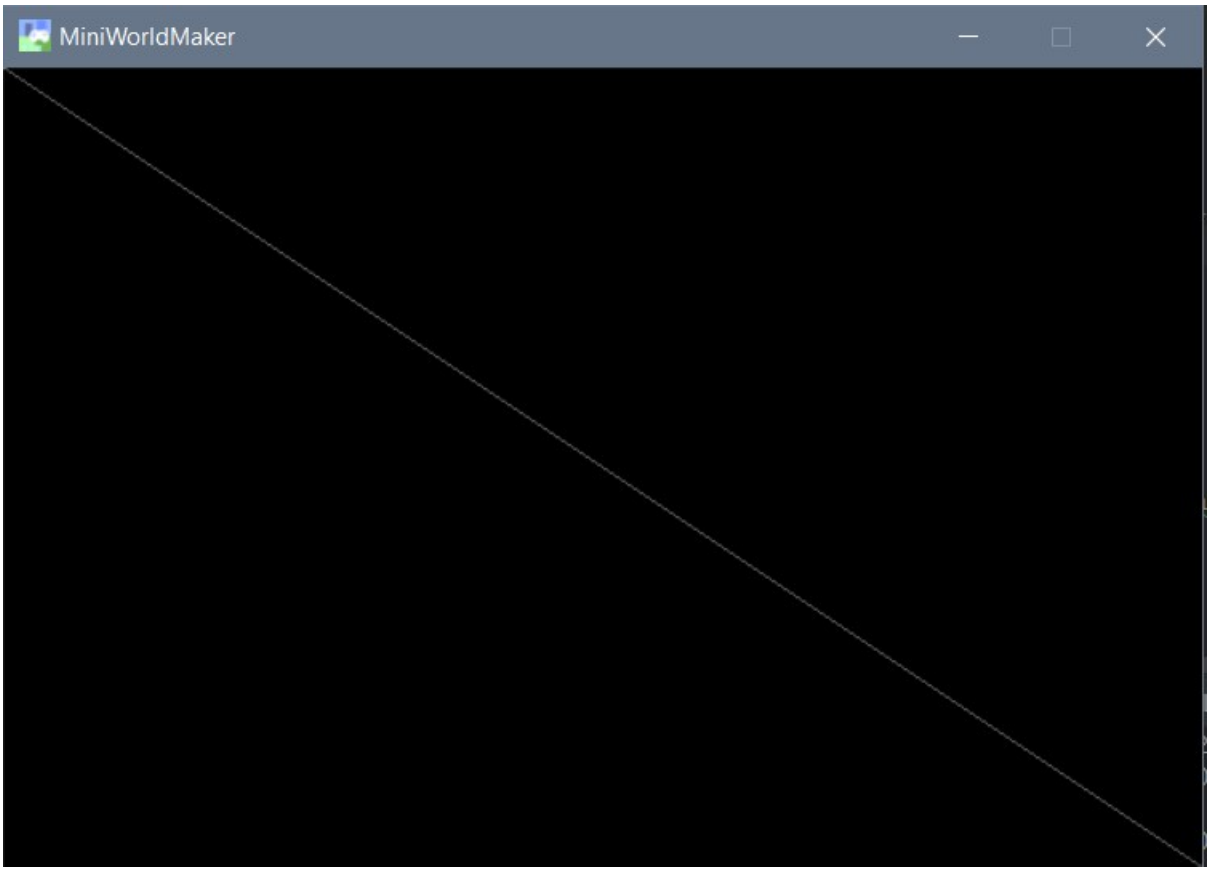
```
from miniworldmaker import *


class MyBoard(ProcessingBoard):
    def on_setup(self):
        self.fill((0, 0, 0, 255))
        Line((0, 0), (600, 400), color=(100, 100, 100, 255))
```

(continues on next page)

(continued from previous page)

```
my_board = MyBoard(600, 400)
my_board.show()
```

Der Hintergrund wird zunächst komplett schwarz gefärbt. Anschließend wird eine Diagonale quer durch das Bild gezeichnet.



Zeichnen der Grundformen

Die Syntax für die wichtigsten Befehle lautet

- Linie: **Line((x, y), (x, y), thickness)**
- Kreis: **Circle((x,y), radius, thickness)**
- Ellipse: **Ellipse((x, y), width, height, thickness)
- Polygon: **Polygon([(x1, y1), (x2, y2), (x3, y3), . . .], thickness)

Der Parameter **thickness** ist optional und gibt die Dicke der Linien an. Wenn du and dieser Stelle eine 0 setzt, dann wird das entsprechende Objekt ausgefüllt gezeichnet.

Beispiel:

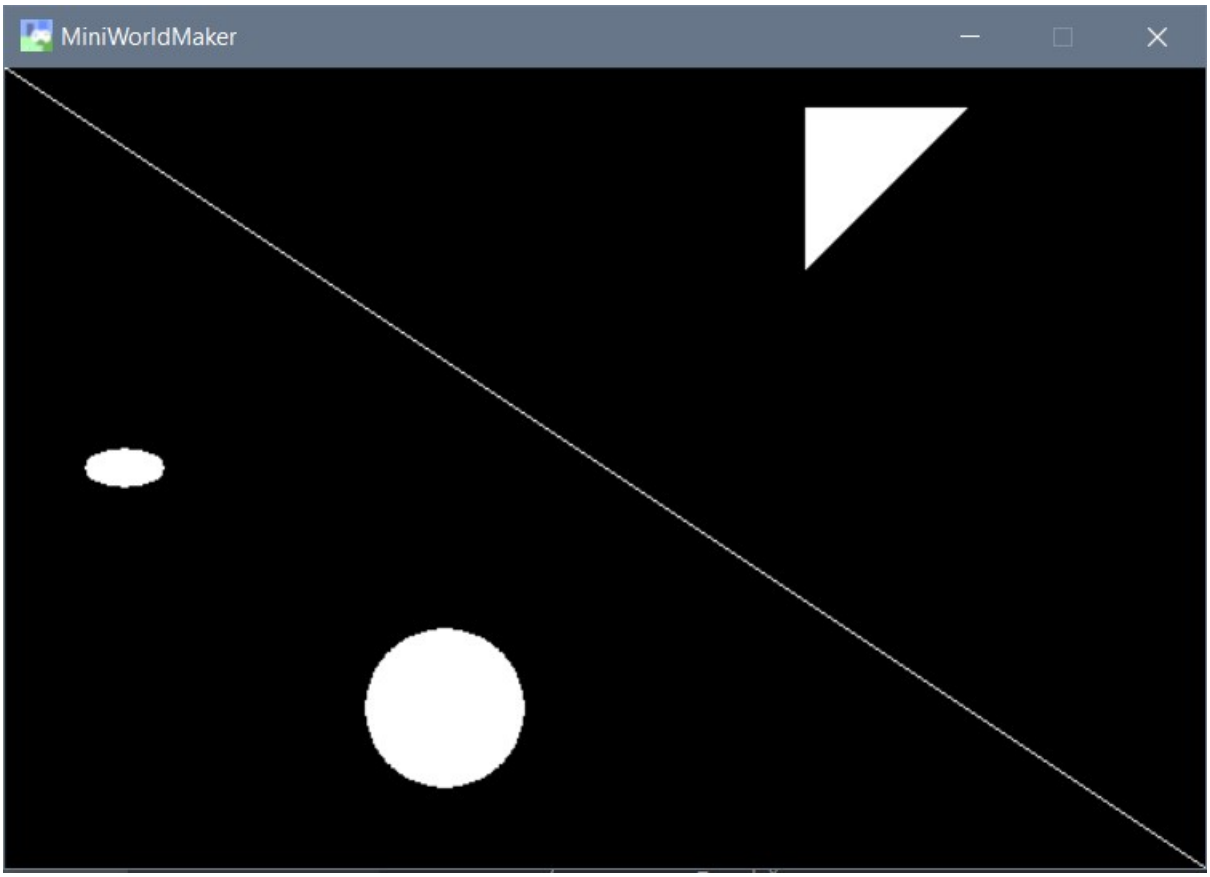
```
from miniworldmaker import *


class MyBoard(ProcessingBoard):
```

(continues on next page)

(continued from previous page)

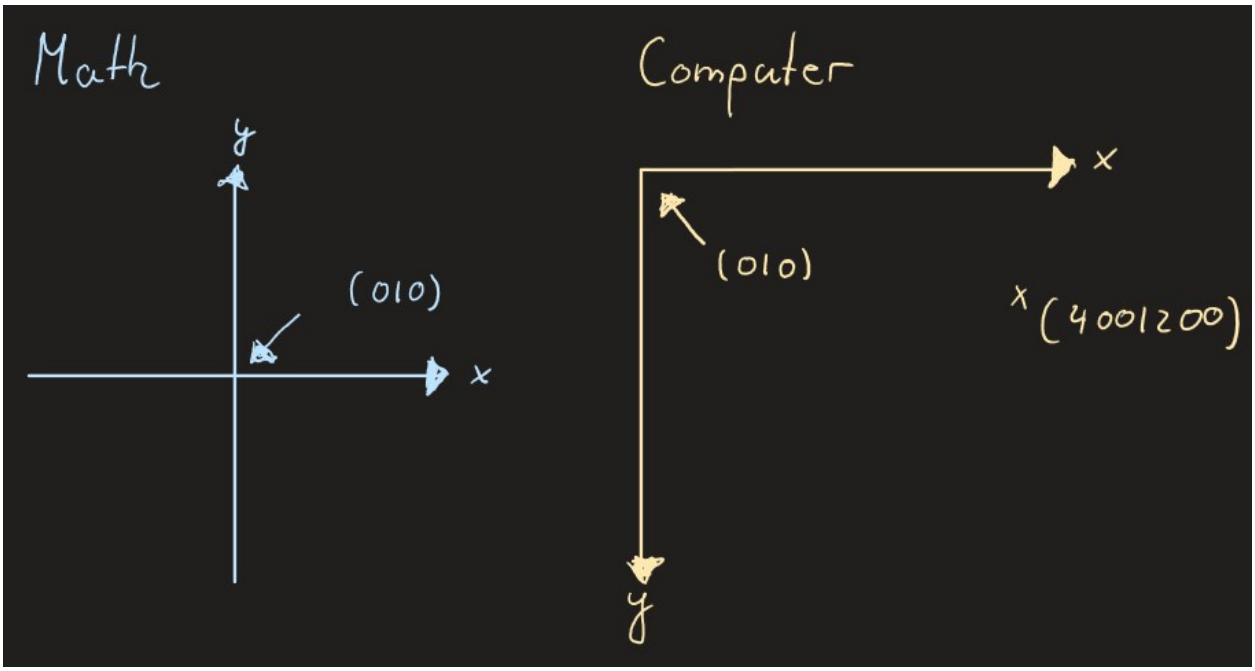
```
    def on_setup(self):
        self.fill((0, 0, 0, 255))
        Line((0, 0), (600, 400))
        Circle((200, 300), 40, 0)
        Ellipse((60, 200), 40, 20, 0)
        Polygon([(400, 100), (480, 20), (400, 20)], 0)


my_board = MyBoard(600, 400)
my_board.show()
```

Der Code ergibt folgendes Bild:



Das Koordinatensystem

Am Computer verwendet man in der Regel ein etwas anderes Koordinatensystem als im Mathematikunterricht



Die obere linke Ecke ist der Ursprung (0|0). Daher bedeuten große y-Werte, dass ein Punkt weiter unten liegt.

Jedes Objekt besitzt folgende Eigenschaften

- object.x - Die x-Koordinate des Objekts
- object.y - Die y-Koordinate des Objekts
- object.position - Die Position des Objekts als 2-Tupel, z.B. (400, 200)

Weiterhin kannst du auch die Rotation ändern mit

- object.direction

Farben

Jedes Objekt kann mit einem weiteren Parameter eingefärbt werden.

Eine Farbe ist ein 4-Tupel: (r, g, b, alpha) . Jeder dieser 4 Parameter ist eine Zahl zwischen 0 und 255.

Die ersten 3 Parameter geben den Anteil von rot, grün und blau im Bild an. Der letzte Parameter gibt die Transparenz der Farbe an. Ein Wert von 0 bedeutet, dass das Objekt absolut transparent (und damit unsichtbar ist). Ein Wert von 255 bedeutet, dass das Objekt nicht transparent ist.

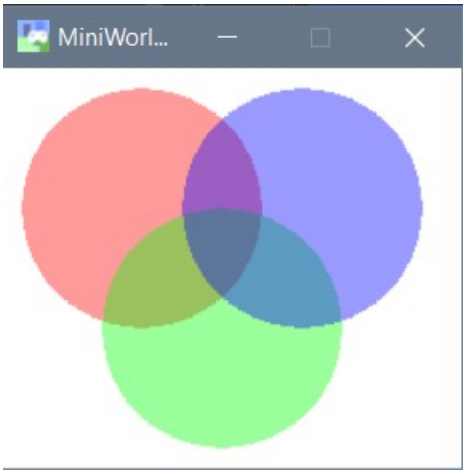
Im folgenden Beispiel wurde ein mittlerer Transparenzwert gewählt, so dass die unten liegenden Kreise noch sichtbar sind:

```
from miniworldmaker import *


class MyBoard(ProcessingBoard):

    def on_setup(self):
        self.fill((255, 255, 255, 255))
        self.circle1 = Circle((40, 40), 60, 0, color = (255, 0, 0, 100))
        self.circle2 = Circle((80, 100), 60, 0, color = (0, 255, 0, 100))
```

(continues on next page)

(continued from previous page)

```
        self.circle3 = Circle((120, 40), 60, 0, color = (0, 0, 255, 100))



my_board = MyBoard(230, 200)
my_board.show()
```



Animationen

Du kannst OBjekte animieren, in dem du Befehle in der Methode act() ergänzt. Diese Methode wird bei jedem Frame einmal aufgerufen.

Beispiel:

```
from miniworldmaker import *


class MyBoard(ProcessingBoard):

    def on_setup(self):
        self.fill((0, 0, 0, 255))
        self.circle1 = Circle((20, 400), 20, 1)
        self.circle2 = Circle((110, 410), 20, 1)
        self.circle3 = Circle((180, 420), 19, 1)
        self.circle4 = Circle((210, 500), 21, 1)
        self.circle5 = Circle((350, 400), 20, 1)
        # self.l1 ist eine Objektvariable. Du kannst in allen Methoden der Klasse auf␣
↪die Variable zugreifen

    def act(self):
        self.circle1.y -= 1
        self.circle2.y -= 2
        self.circle3.y -= 3
        self.circle4.y -= 1
        self.circle5.y -= 2


my_board = MyBoard(400, 400)
my_board.show()
```

So sieht das Programm aus:

Events

Events sind Ereignisse auf die das Programm reagieren kann. Ein Event kann z.B. sein:

- Eine Maustaste wurde gedrückt oder die Maus wurde bewegt.
- Eine Taste wurde gedrückt.
- Das Board wurde neu erstellt.

Events rufen automatisch spezielle Funktionen auf. Wenn du diese Funktionen **überschreibst**, kannst du das Verhalten des Programms auf die Ereignisse steuern.

Folgende Funktionen kannst du überschreiben:

- **on_setup()** - Beim erstellen des Boards
- **on_key_pressed(self, keys)** - Wenn eine Taste gedrückt wird (wird immer wieder aufgerufen, wenn du die Taste gedrückt lässt)
- **on_key_down(self, keys)** - Wenn die Taste gerade herunter gedrückt wird
- **on_key_up(self, keys)** - Wenn die Taste wieder losgelassen wird
- **on_mouse_left(self, mouse_pos)** - Wenn die linke Maustaste gedrückt wird
- **on_mouse_right(self, mouse_pos)** - Wenn die rechte Maustaste gedrückt wird
- **on_mouse_motion(self, mouse_pos)** - Wenn die Maus bewegt wird (Wird bei Bewegung immer wieder aufgerufen)
- **act()** - Unabhängig von Events wird diese Methode immer wieder aufgerufen

Beispiel:

```
from miniworldmaker import *


class MyBoard(ProcessingBoard):

    def on_setup(self):
        self.color = (255, 255, 255, 50)
        self.fill((0,0,0,255))

    def act(self):
        Ellipse(self.get_mouse_position(), 80, 80, 1, self.color)

    def on_mouse_left(self, mouse_pos):
        self.color = (200, 100, 100, 50)

    def on_mouse_right(self, mouse_pos):
        self.color = (255, 255, 255, 50)


my_board = MyBoard()
my_board.show()
```

variables

Defining Variables

We have already used variables in the previous examples.

You can define new variables by writing:

```
self.variable name = value
```

- The **self** always refers to the current object. For example, if you have created several circles, **self** means that the variable name to *this* circle and to no other.
- A variable is a **Name** for an object. An object can be a number, a word, a geometric shape or much more. By giving the object a name, you can access it and change it.

Consider the following example:

```
from miniworldmaker import *


class MyBoard(ProcessingBoard):

    def on_setup(self):
        self.circle1 = Circle((40, 40), 60, 0, color = (255, 0, 0, 100))
        self.circle2 = Circle((80, 100), 60, 0, color = (0, 255, 0, 100))

    def on_mouse_left(self, mouse_pos):
        self.circle1.x = 150

my_board = MyBoard(400, 400)
my_board.show()
```

A board of the type MyBoard has two circles. By giving the circles **names**. (namely self.circle1 and self.circle2) you can also access the circles elsewhere.

Here the x-coordinate of the first circle is set to 150.

The Random Function

The Random function allows you to assign random values to things. First you have to randomly import the library at the beginning of your file:

```
import random
```

Then a single command is sufficient for the first one.

```
random.randint(0, 5)
```

This creates a random number between 0 and 5

The following program lets a circle jump to a random position:

```
from miniworldmaker import *
import random


class MyBoard(ProcessingBoard):

    def on_setup(self):
        self.circle1 = Circle((40, 40), 60, 0, color=(255, 0, 0, 255)))

    def on_mouse_left(self, mouse_pos):
        self.circle1.x = random.randint(0, 260)
        self.circle1.y = random.randint(0, 200)


my_board = MyBoard(260, 200)
my_board.show()
```

Lists and loops

lists

Create lists

A list contains several objects without you having to give each one a new name.

Example:

```
list = [0, 1, 2, 3, 4]
```

The list contains the numbers 0-4.

You can also create lists by first creating an empty list and then adding numbers one after the other:

```
list2 = []
list2.append(5)
list2.append(6)
list2.append(7)
```

This list contains the numbers 5, 6 and 7.

In the same way, a list can also contain objects of any kind.

```
cliste = []
cliste.append(Circle((40, 40), 60, 0, color=(255, 0, 0, 100))))
```

This adds a circle to a list.

Accessing list items

The list elements can be accessed with a **index**:

```
list2 = []
list2.append(5)
list2.append(6)
list2.append(7)
print(list2[0], list2[1])
```

Spends 5 6. The 0th list element is 6, the 1st list element is 6.

Loops

With the help of loops you can repeat things. For example, if you want to create 50 circles instead of 5, the easiest way to do this is with a loop:

```
class MyBoard(ProcessingBoard):

    def on_setup(self):
        self.fill((255, 255, 255, 255)))
        for i in range(50):
            Circle((random.randint(0,260), random.randint(0,200)), 10, 0, color=(255,␣
↪0, 0, 100))
```

The program creates 50 circles at random position.

With the help of lists you can also move all circles at the same time.

```
class MyBoard(ProcessingBoard):

    def on_setup(self):
        self.fill((255, 255, 255, 255)))
        self.lst = []
        for i in range(50):
            self.lst.append(Circle((random.randint(0, 800), random.randint(0, 600)),␣
↪random.randint(10, 20), 0, color=(255, 0, 0, 100)))

    def act(self):
        for circle in self.lst:
            circle.y-=random.randint(0,2)
```

Listen und Schleifen

Listen

Listen erstellen

Eine Liste enthält mehrere Objekte, ohne dass Sie jedem einen neuen Namen geben müssen.

Beispiel:

```
Liste = [0, 1, 2, 2, 3, 4]
```

Die Liste enthält die Zahlen 0-4.

Sie können Listen auch erstellen, indem Sie zuerst eine leere Liste erstellen und dann nacheinander Zahlen hinzufügen:

```
list2 = []
list2.append(5)
list2.append(6)
list2.append(7)
```

Diese Liste enthält die Zahlen 5, 6 und 7.

Ebenso kann eine Liste auch Objekte jeglicher Art enthalten.

```
cliste = []
cliste.append(Circle((40, 40), 60, 0, 0, color=(255, 0, 0, 0, 100))))))
```

Dadurch wird einer Liste ein Kreis hinzugefügt.

Zugriff auf Listenelemente

Auf die Listenelemente kann mit einem **Index** zugegriffen werden:

```
list2 = []
list2.append(5)
list2.append(6)
list2.append(7)
drucken(list2[0], list2[1])
```

Gibt 5 6 aus, das 0. Listenelement ist 6, das 1. Listenelement ist 6.

Schleifen

Mit Hilfe von Schleifen können Sie Dinge wiederholen. Wenn Sie z.B. 50 Kreise statt 5 erstellen möchten, der einfachste Weg, dies zu tun, ist mit einer Schleife:

```
Klasse MyBoard(ProcessingBoard):

    def on_setup(self):
        self.fill((255, 255, 255, 255, 255, 255))))
        für i im Bereich (50):
            Circle((random.randint(0,260), random.randint(0,200)), 10, 0, 0,␣
↪color=(255, 0, 0, 0, 100)))
```

Das Programm erzeugt 50 Kreise an einer beliebigen Position.

Mit Hilfe von Listen können Sie auch alle Kreise gleichzeitig bewegen.

```
Klasse MyBoard(ProcessingBoard):

    def on_setup(self):
        self.fill((255, 255, 255, 255, 255, 255))))
        self.lst = []
        für i im Bereich (50):
            self.lst.append(Circle((random.randint(0, 800), random.randint(0, 600)),␣
↪random.randint(10, 20), 0, color=(255, 0, 0, 0, 100))))

    def act(self):
        für Kreis in self.lst:
            circle.y-=random.randint(0,2)
```

Physik

-tbd-

2.2.2 Processing_Tutorial

Installation

Install the framework:

```
pip install miniworldmaker
```

First Steps

First copy the following supporting program into your code editor:

```
from miniworldmaker import *

class MyBoard(ProcessingBoard):
    def on_setup(self):
        pass

my_board = MyBoard(600, 400)
my_board.show()
```

This framework programme does the following:

- In the upper part of the source code, a separate class (i.e. a blueprint) is created for your own miniworld. It is a special ProcessingBoard.
- In the lower part, a new window with width 600 and height 400 is created from this blueprint.

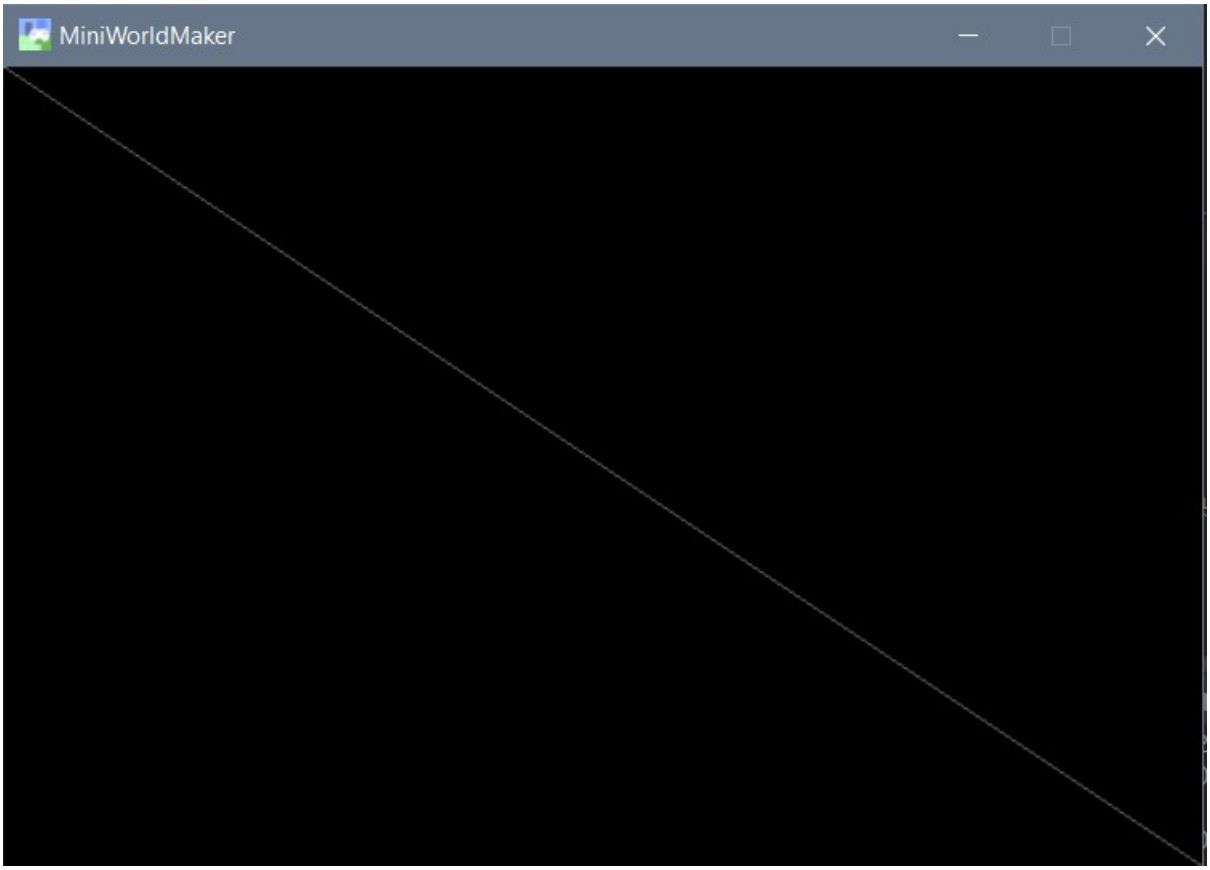
The first image

You can create images by adding graphic objects to the setup() method:

This works like this:

```
from miniworldmaker import *


class MyBoard(ProcessingBoard):
    def on_setup(self):
        self.fill((0, 0, 0, 255)))
        Line((0, 0), (600, 400), color=(100, 100, 100, 255))


my_board = MyBoard(600, 400)
my_board.show()
```

The background is first colored completely black. A diagonal is then drawn across the image.



Drawing the basic shapes

The syntax for the most important commands is as follows

- Line: **Line((x, y), (x, y), thickness)**
- Circle: **Circle((x,y), radius, thickness)**
- Ellipse: **Ellipse((x, y), width, height, thickness)
- Polygon: **Polygon([(x1, y1), (x2, y2), (x3, y3), . . .], thickness)

The **thickness** parameter is optional and specifies the thickness of the lines. If you set this to 0, then the corresponding object is filled in and drawn.

Example:

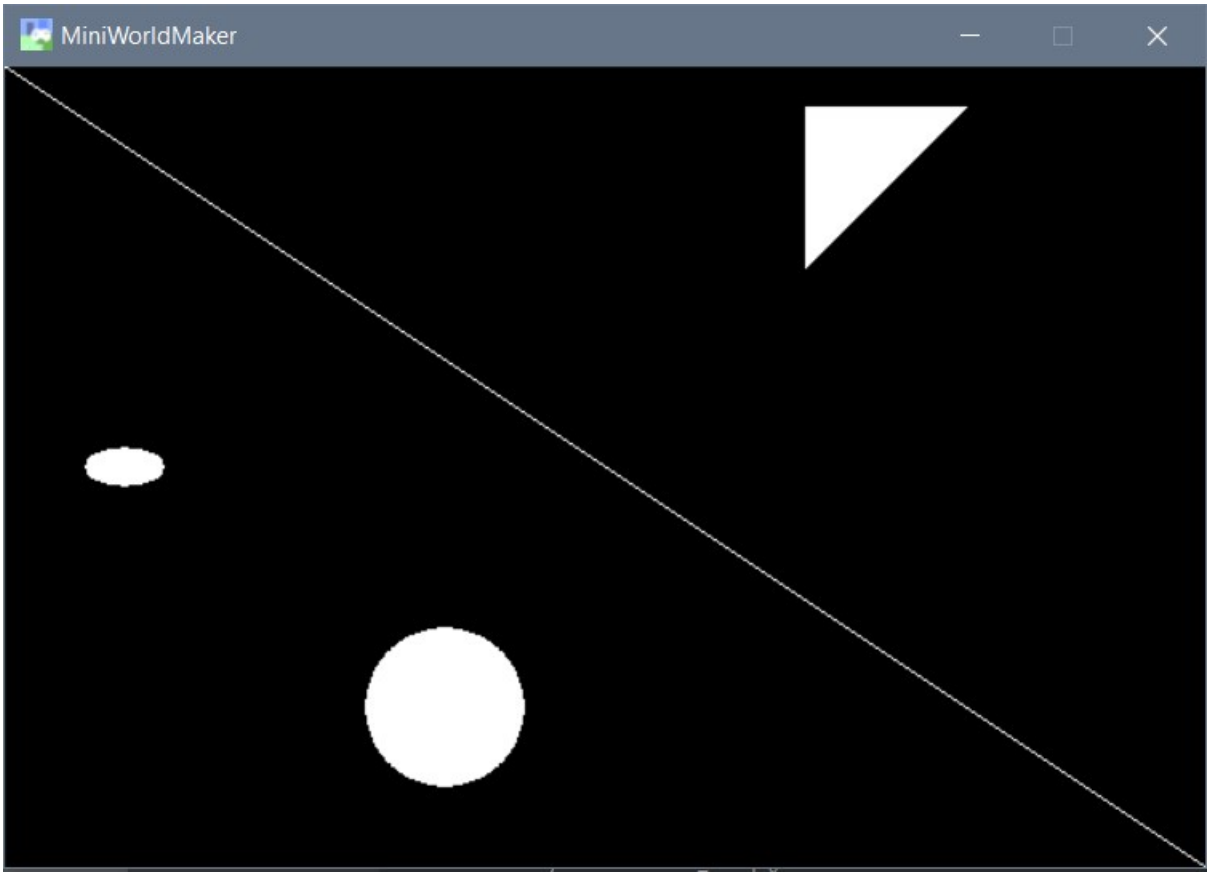
```
from miniworldmaker import *


class MyBoard(ProcessingBoard):
    def on_setup(self):
        self.fill((0, 0, 0, 255)))
        Line((0, 0), (600, 400))
        Circle((200, 300), 40, 0)
        Ellipse((60, 200), 40, 20, 0)
        Polygon([(400, 100), (480, 20), (400, 20)], 0)
```

(continues on next page)

(continued from previous page)

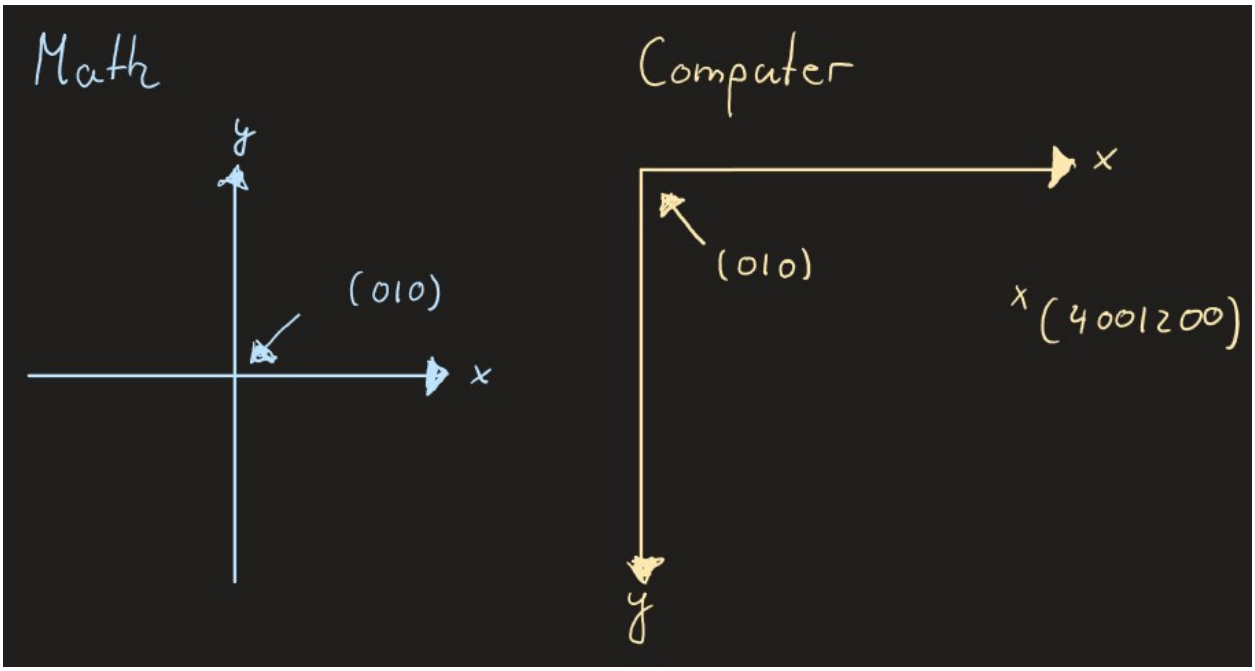
```
my_board = MyBoard(600, 400)
my_board.show()
```

The code results in the following image:



The coordinate system

The computer usually uses a slightly different coordinate system than in mathematics lessons.



The upper left corner is the origin (0|0). Therefore, large y-values mean that a point is further down.

Each object has the following properties

- object.x - The x-coordinate of the object
- object.y - The y coordinate of the object.
- object.position - The position of the object as a 2-tuple, e.g. (400, 200)

You can also change the rotation with

- object.direction

Colors

Each object can be colored with an additional parameter.

A color is a 4-tuple: (r, g, b, alpha) . Each of these 4 parameters is a number between 0 and 255.

The first 3 parameters indicate the proportion of red, green and blue in the image. The last parameter indicates the transparency of the color. A value of 0 means that the object is absolutely transparent (and therefore invisible). A value of 255 means that the object is not is transparent.

In the following example, an average transparency value was selected so that the circles below are still visible:

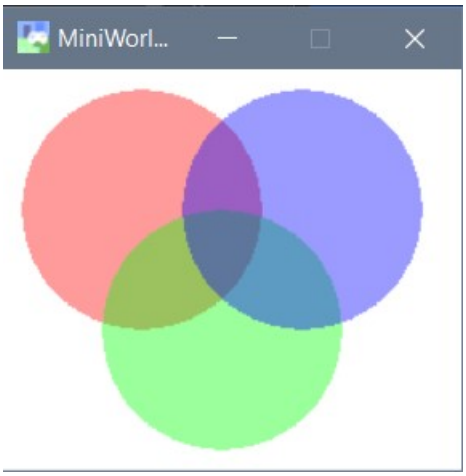
```
from miniworldmaker import *


class MyBoard(ProcessingBoard):

    def on_setup(self):
        self.fill((255, 255, 255, 255)))
        self.circle1 = Circle((40, 40), 60, 0, color = (255, 0, 0, 100))
        self.circle2 = Circle((80, 100), 60, 0, color = (0, 255, 0, 100))
        self.circle3 = Circle((120, 40), 60, 0, color = (0, 0, 255, 100))
```

(continues on next page)

(continued from previous page)

```
my_board = MyBoard(230, 200)
my_board.show()
```

Beispiel:



Animations

You can animate Objects by adding commands to the act() method. This method is called once for each frame.

Example:

```
from miniworldmaker import *


class MyBoard(ProcessingBoard):

    def on_setup(self):
        self.fill((0, 0, 0, 255)))
        self.circle1 = Circle((20, 400), 20, 1)
        self.circle2 = Circle((110, 410), 20, 1)
        self.circle3 = Circle((180, 420), 19, 1)
        self.circle4 = Circle((210, 500), 21, 1)
        self.circle5 = Circle((350, 400), 20, 1)
        # self.l1 is an object variable. You can access the variable in all methods␣
↪of the class.

    def act(self):
        self.circle1.y -= 1
        self.circle2.y -= 2
        self.circle3.y -= 3
        self.circle4.y -= 1
        self.circle5.y -= 2


my_board = MyBoard(400, 400)
my_board.show()
```

This is the program:

Events

Events are events to which the program can react. An event can be, for example:

- A mouse button was pressed or the mouse was moved.
- A key was pressed.
- The board has been rebuilt.

Events automatically call up special functions. If you **overwrite these functions**, you can control the behavior of the program on the events.

You can overwrite the following functions:

on_setup() - When creating the board **on_key_pressed(self, keys)** - When a key is pressed (is called again and again), if you keep the button pressed) **on_key_down(self, keys)** - When the key is just pressed down **on_key_up(self, keys)** - When the key is released again **on_mouse_left(self, mouse_pos)** - When left mouse button is pressed **on_mouse_right(self, mouse_pos)** - When right mouse button is pressed **on_mouse_motion(self, mouse_pos)** - When the mouse is moved (Is called again and again during movement) **act()** - This method is called again and again independent of events.

Example:

```
from miniworldmaker import *


class MyBoard(ProcessingBoard):

    def on_setup(self):
        self.color = (255, 255, 255, 50)
        self.fill((0,0,0,255)))

    def act(self):
        Ellipse(self.get_mouse_position(), 80, 80, 1, self.color)

    def on_mouse_left(self, mouse_pos):
        self.color = (200, 100, 100, 50)

    def on_mouse_right(self, mouse_pos):
        self.color = (255, 255, 255, 50)


my_board = MyBoard()
my_board.show()
```

Translated with www.DeepL.com/Translator

variables == == == == =

Defining Variables

We have already used variables in the previous examples.

You can define new variables by writing:

```
self.variable
name = value
```

*The ** self ** always refers to the current object. For example, if you have created several circles, ** self ** means that the variable name to * this * circle and to no other. *A variable is a ** Name ** for an object.An object can be a number, a word, a geometric shape or much more.By giving the object a name, you can access it and change it.

Consider the following example:

```
from miniworldmaker import *


class MyBoard(ProcessingBoard):

    def on_setup(self):
        self.circle1 = Circle((40, 40), 60, 0, color=(255, 0, 0, 100))
        self.circle2 = Circle((80, 100), 60, 0, color=(0, 255, 0, 100))

    def on_mouse_left(self, mouse_pos):
        self.circle1.x = 150


my_board = MyBoard(400, 400)
my_board.show()
```

A board of the type MyBoard has two circles.By giving the circles ** names **. (namely self.circle1 and self.circle2) you can also access the circles elsewhere.

Here the x - coordinate of the first circle is set to 150.

The Random Function

The Random function allows you to assign random values to things. First you have to randomly import the

library at the beginning of your file:

```
import random
```

Then a single command is sufficient for the first one.

```
random.randint(0, 5)
```

This creates a random number between 0 and 5

The following program lets a circle jump to a random position:

```
from miniworldmaker import *
import random


class MyBoard(ProcessingBoard):

    def on_setup(self):
        self.circle1 = Circle((40, 40), 60, 0, color=(255, 0, 0, 255)))

        def on_mouse_left(self, mouse_pos):
            self.circle1.x = random.randint(0, 260)
            self.circle1.y = random.randint(0, 200)

    my_board = MyBoard(260, 200)
    my_board.show()

    ```
```

Listen und Schleifen

Listen

Listen erstellen

Eine Liste enthält mehrere Objekte, ohne dass Sie jedem einen neuen Namen geben müssen.

Beispiel:

```
Liste = [0, 1, 2, 2, 3, 4]
```

Die Liste enthält die Zahlen 0-4.

Sie können Listen auch erstellen, indem Sie zuerst eine leere Liste erstellen und dann nacheinander Zahlen hinzufügen:

```
list2 = []
list2.append(5)
list2.append(6)
list2.append(7)
```

Diese Liste enthält die Zahlen 5, 6 und 7.

Ebenso kann eine Liste auch Objekte jeglicher Art enthalten.

```
cliste = []
cliste.append(Circle((40, 40), 60, 0, 0, color=(255, 0, 0, 0, 100))))))
```

Dadurch wird einer Liste ein Kreis hinzugefügt.
```

Zugriff auf Listenelemente

Auf die Listenelemente kann mit einem **Index** zugegriffen werden:

```
list2 = []
list2.append(5)
list2.append(6)
list2.append(7)
drucken(list2[0], list2[1])
```

Gibt 5 6 aus, das 0. Listenelement ist 6, das 1. Listenelement ist 6.

Schleifen

Mit Hilfe von Schleifen können Sie Dinge wiederholen. Wenn Sie z.B. 50 Kreise statt 5 erstellen möchten, der einfachste Weg, dies zu tun, ist mit einer Schleife:

```
Klasse MyBoard(ProcessingBoard):

    def on_setup(self):
        self.fill((255, 255, 255, 255, 255, 255))))
        für i im Bereich (50):
            Circle((random.randint(0,260), random.randint(0,200)), 10, 0, 0,␣
↪color=(255, 0, 0, 0, 100)))
```

Das Programm erzeugt 50 Kreise an einer beliebigen Position.

Mit Hilfe von Listen können Sie auch alle Kreise gleichzeitig bewegen.

```
Klasse MyBoard(ProcessingBoard):

    def on_setup(self):
        self.fill((255, 255, 255, 255, 255, 255))))
        self.lst = []
        für i im Bereich (50):
            self.lst.append(Circle((random.randint(0, 800), random.randint(0, 600)),␣
↪random.randint(10, 20), 0, color=(255, 0, 0, 0, 100))))

    def act(self):
        für Kreis in self.lst:
            circle.y-=random.randint(0,2)
```

Listen and Schleifen

Hören

Listen erstellen

A list contains several objects, without that the any an new name to be.

Beispiel:

```
Liste = [0, 1, 2, 2, 2, 3, 4]
```

Die Liste enthält die Zahlen 0-4.

The can be listen even to erstellen, to you's only a leere list erstellen and then nacheinander Zahlen hinzufügen:

```
list2 = []
list2.append(5)
list2.append(6)
list2.append(7)
```

Diese Liste enthält die Zahlen 5, 6 und 7.

Ebenso kann eine Liste von Objekten jeglicher Art erstellt werden.

```
cliste = []
cliste.append(Circle((40, 40), 60, 0, 0, 0, 0, color=(255, 0, 0, 0, 0, 0,␣
↪100)))))))))))
```

Dadurch wird eine Liste eines Kreis hinzugefügt.

Zugriff auf Listenelemente

Auf die Listenelemente kann mit einem **Index** zugegriffen werden:

```
list2 = []
list2.append(5)
list2.append(6)
list2.append(7)
drucken(list2[0], list2[1])
```

Gibt 5 6 aus, das 0. Listenelement ist 6, das 1. Listenelement ist 6.

Schleifen

With help of Schleifen can the Dinge wiederholen. Wenn Sie z.B. 50 Kreise statt 5 erstellen möchten, Der einfachste Weg, der zu tun ist, ist mit einer Schleife:

```
Klasse MyBoard(ProcessingBoard):

    def on_setup(self):
        self.fill((255, 255, 255, 255, 255, 255, 255, 255))))))
        for i im Bereich (50):
            Circle((random.randint(0,260), random.randint(0,200)), 10, 0, 0, 0, 0,␣
↪color=(255, 0, 0, 0, 0, 0, 100)))))
```

Das Programm erzeugt 50 Kreise an einer beliebigen Position.

Mit Hilfe von Listen können Sie auch alle Kreise gleichzeitig bewegen.

```
Klasse MyBoard(ProcessingBoard):

    def on_setup(self):
        self.fill((255, 255, 255, 255, 255, 255, 255, 255))))))
        self.lst = []
        for i im Bereich (50):
            self.lst.append(Circle((random.randint(0, 800), random.randint(0, 600)),␣
↪random.randint(10, 20), 0, 0, color=(255, 0, 0, 0, 0, 0, 0, 100)))))))))

    def act(self):
        für Kreis in self.lst:
            circle.y-=random.randint(0,2)
```

physics

MiniWorldmaker has an integrated physics environment.

To physically simulate an object, you must overwrite the method **setup_physics()**.

Example:

```
class Paddle(Rectangle):
    def setup(self):
        self.size = (10, 80)
        self.costume.is_rotatable = False

    def setup_physics(self):
        self.physics.stable = True
        self.physics.can_move = True
        self.physics.mass = "inf"
        self.physics.friction = 0
        self.physics.gravity = False
        self.physics.elasticity = 1
```

If the method is implemented, the physics engine is initialized before executing the setup() method. Once the engine is initialized, you can "push" objects. This works like this:

```
class Ball(Circle):

    def on_setup(self):
        self.direction = 30
        self.physics.impulse_in_direction(300)
```

or like this:

```
class Bird(Actor):

    def on_setup(self):
        ...
        self.physics.velocity_x = 600
        self.physics.velocity_y = - self.board.arrow.direction * 50
```

CHAPTER 3

API

3.1 Boards

3.1.1 Board

PixelBoard

TiledBoard

Processing Board

3.2 BoardPositions, Lines and Rectangles

3.2.1 BoardPosition

3.2.2 BoardRect

3.3 Tokens

3.3.1 Token

3.3.2 TextToken

3.3.3 NumberToken

3.3.4 Shape

Point

Circle

Ellipse

Line

Rectangle

Polygon

3.4 Costumes and Backgrounds

The Costume and Background classes are child classes of the Appearance class.

The Appearance class contains all the logic common to both, e.g. scaling and rotating images. The child classes contain the actions that are specific to these classes (e.g. certain overlays).

All actions performed on the images can be found in the class ImageRenderer

3.4.1 Appearance

3.4.2 Background

3.4.3 Costume

3.5 Physics

CHAPTER 4

Examples

These are some links to examples. You can find all examples in github

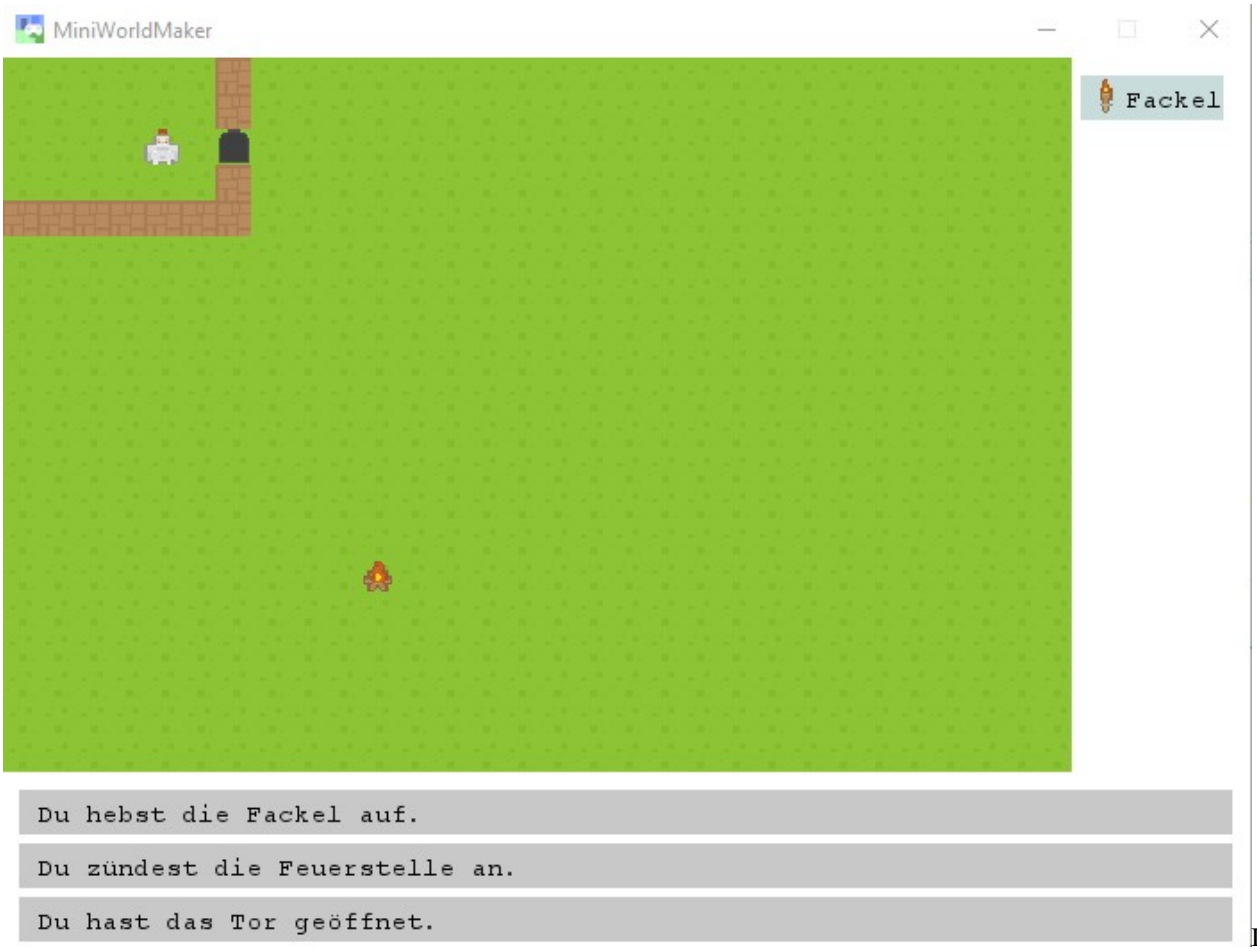
4.1 Basics

- Basic Framework - Basic concepts for the creation of boards and actors. Image - Basic Framework

4.2 Tiled Boards

These examples will show you how to use tiled boards. Tiled boards are boards where the players move on square cells.

- Basic Movement: Basic Movement, Reaction to events.
- The Crash: Collision detection, Creating and Removing Actors in Runtime.
- Rpg: This example shows you how to interact with the world.



Image

- RPG

.. raw:: html

4.3 Pixel Boards

These examples show you how tu use boards with pixel-precise placement

- Roboanimation: Shows you how to animate an actor
- Roboracing: Keyboard controlled movement.
- Roboracing 2: Keyboard controlled movement.
- Asteroids: Collision detection

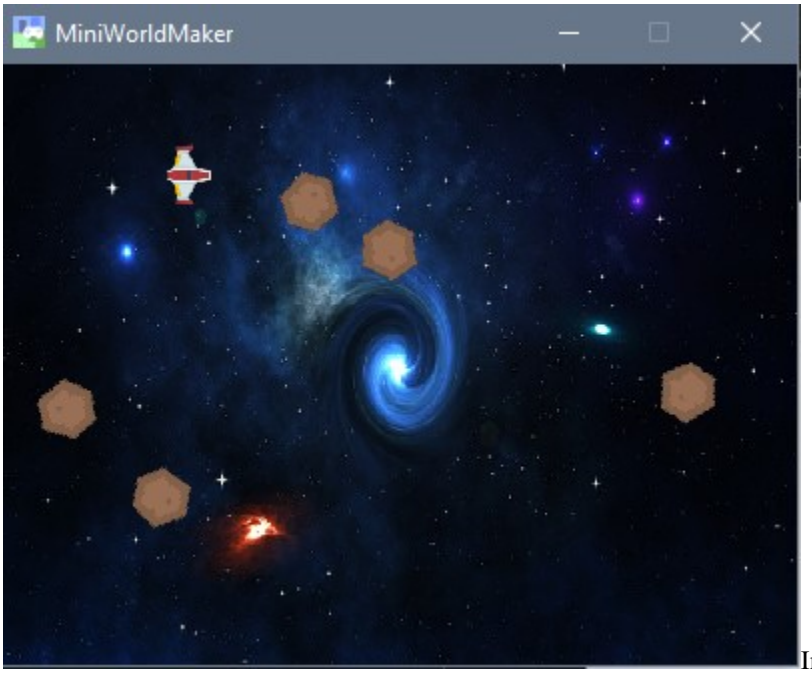


Image - Asteroids

CHAPTER 5

Credits

- Greenfoot: The model for this framework.
- Deepl: Assistant for translating pages.
- Kenny Assets: Most of the images in the example-code are based on kenny assets.

CHAPTER 6

Impressum and Contact

- Impressum
- Twitter: @a_siebel
- Mail: a.siebel@cws-usingen.de