
game module - Billard@ISEM

Release 0.1

Mathis Wolter

Jan 13, 2026

CONTENTS:

1	Game Module - Billard@ISEM	3
1.1	Installation	3
1.1.1	.env fields	3
1.2	Documentation	4
2	API	5
2.1	Game	5
2.1.1	Game	5
2.1.2	Elo	8
2.1.2.1	Elo	9
2.1.3	GameEngine	9
2.1.3.1	GameEngine	9
2.1.4	GameImage	10
2.1.4.1	Game.GameImage.ie	10
2.1.4.2	BilliardBall	10
2.1.4.3	GameImage	12
2.1.4.4	Trickshot	18
2.1.5	GameRules	20
2.1.5.1	GameRules	20
2.1.6	GlobalAPI	21
2.1.6.1	GlobalAPI	21
2.1.7	Player	22
2.1.7.1	Player	22
2.1.8	gamemodes	23
2.1.8.1	GameMode	25
2.1.8.1.1	GameMode	25
2.1.8.2	GameModel	31
2.1.8.2.1	Game	31
2.1.8.3	api_utils	32
2.1.8.3.1	API	32
2.1.8.3.2	ApiAuth	33
2.1.8.4	common_utils	34
2.1.8.4.1	Game.gamemodes.common_utils.ball_distance	34
2.1.8.4.2	Game.gamemodes.common_utils.check_positions	35
2.1.8.4.3	Game.gamemodes.common_utils.classify_region	35
2.1.8.4.4	Game.gamemodes.common_utils.coord_to_vec	35
2.1.8.4.5	Game.gamemodes.common_utils.coords_report	35
2.1.8.4.6	Game.gamemodes.common_utils.get_border_intercept	35
2.1.8.4.7	Game.gamemodes.common_utils.is_close_enough	35
2.1.8.4.8	Game.gamemodes.common_utils.load_challenge_files	35

2.1.8.4.9	Game.gamemodes.common_utils.load_coordinate_file	35
2.1.8.4.10	Game.gamemodes.common_utils.metric_distance	36
2.1.8.4.11	Game.gamemodes.common_utils.metric_distance_closest	36
2.1.8.4.12	Game.gamemodes.common_utils.project_line	36
2.1.8.4.13	Game.gamemodes.common_utils.project_on_segment	36
2.1.8.4.14	Game.gamemodes.common_utils.vec_to_coord	36
2.1.8.5	distance	36
2.1.8.5.1	Distance	37
2.1.8.6	dummy	38
2.1.8.6.1	Dummy	38
2.1.8.7	final_competition	39
2.1.8.7.1	FinalCompetition	39
2.1.8.8	kp2	40
2.1.8.8.1	KP2	40
2.1.8.9	local_game	42
2.1.8.9.1	LocalGame	42
2.1.8.10	longest_break	43
2.1.8.10.1	LongestBreak	43
2.1.8.11	online_game	45
2.1.8.11.1	OnlineGame	45
2.1.8.12	precision	46
2.1.8.12.1	Precision	46
2.1.8.13	single_break	47
2.1.8.13.1	Break	47
3	To Do	49
	Python Module Index	51
	Index	53

The Game Module is a part of the [Billard@ISEM](#) system, ultimately aiming at teaching students relevant mechanical design principles as well as provide a research environment for agile product development methods.

GAME MODULE - BILLARD@ISEM

The repository provides the game module for the [Billard@ISEM](#) system.

1.1 Installation

- Clone the repository: `git clone https://github.com/ISEM-TUHH/billard-game-module.git`
- Modify `config.json` and/or `test_config.json` as well as add a `.env` file as specified later on
- From inside the repository folder create and install all relevant python dependencies: `python -m venv venv`
&& `source venv/bin/activate` && `pip install -r requirements.txt`
- Download the font used for generating images for the beamer and paste it into the `fonts` folder.
 - We use `Minecraft-Regular.otf` from publicly available Minecraft font collections.
- For testing the system, run `python main.py`.
 - This starts the server using the configuration from `test_config.json`
 - Open the specified webpage and check that everything works
- To run in the goal environment, set the environment variable `PROD_OR_TEST` (`export PROD_OR_TEST=PROD`) and restart the server
 - Now the configuration from `config.json` is used
- Add a `systemd` service so the server automatically runs when the device boots up
 - See the exemplary `job.service` file (you can change the name, exemplary name)
 - Move the `job.service` into `/etc/systemd/system/` (most likely a `sudo` operation)
 - Reload the daemon, enable and start the service: `sudo systemctl reload-daemon && systemctl enable job.service && systemctl start job.service`
 - Check if the server is running as intended: `sudo journalctl -u job.service`
 - If successfully started, the server should now be accessible under the address/port provided in `config.json`.
- A lot of functionalities require the other modules to also be up and running (mostly Game, Camera and Beamer modules). See <https://github.com/ISEM-TUHH> for the respective modules.

1.1.1 .env fields

The following fields are needed to be set in a `.env` file in the root directory:

```
# DOWNLOAD SECTION: authentication for /download
USER=XXX
PASSWORD=XXX

# TABLE: authentication for the table to communicate with the global API (for online
↪games)
TID=XXX
TAUTH=XXX

# API: connection details for the global API
ADDRESS=https://XXX.XXX.XXX.XXX
PORT=XXX
```

To be able to communicate with the global API to play online games you need to be registered with us. Write us an e-mail if you want to get registered. We will provide you with a TID and TAUTH

Important: The API is not publicly available yet, we are working on it.

1.2 Documentation

Documentation is generated using sphinx. A prebuild PDF can be found in docs/latex. To generate the documentation, use

```
make html
```

in the docs directory for a html output or `make latexpdf` for a PDF. This requires an installation of LaTeX on the system.

<i>Game</i>

2.1 Game

Classes

<i>Game</i> ([config, test_config, storage_folder, ...])	Implements central game scheduling functions
--	--

2.1.1 Game

class `Game.Game`(*config*='config/config.json', *test_config*='config/test_config.json', *storage_folder*='storage',
 download_folder='gamemodes/resources', *template_folder*='templates')

Bases: `Module`

Implements central game scheduling functions

Bundles all modules APIs and provides the central website.

In a default environment, this module runs in test mode (specified in parent class `billard_base_module.Module`). To start in production mode, set the environment variable `PROD_OR_TEST=PROD`.

base_image

Supplies the definition for a default `GameImage` that gets displayed when the game website is opened, but no gamemode is selected.

Type
list

current_dir

Absolute path of this file.

Type
str

storage

concat of current_dir and storage_folder (absolute path)

Type

str

camera

interface object to the remote camera module, configured from passed config

Type

Camera

beamer

like camera, but for the beamer module.

Type

Beamer

gameimage

the current displayed image. Most of the times this gets pushed to the beamer module.

Type

GameImage

GAMEMODES

mapping gamemode names to GameMode objects. Gets used to pipe input from the client website (posted on *gamemodecontroller* endpoint) to the gamemode specified in the posted data.

Type

dict

api

nested dictionary of all API endpoints.

Type

dict

__init__ (*config*='config/config.json', *test_config*='config/test_config.json', *storage_folder*='storage', *download_folder*='gamemodes/resources', *template_folder*='templates')

Initializes the Game object. Provide relative paths (to this file).

Parameters

- **config** (*str*, *optional*) – path to a configuration *.json* to be used in production mode
- **test_config** (*str*, *optional*) – path to a configuration *.json* to be used in test mode
- **storage_folder** (*str*, *optional*) – path to a folder where some storage files are kept. Not really relevant, only players.json gets read from there.
- **download_folder** (*str*, *optional*) – relative path to a folder where resources that should be available to download from endpoint */download* after authentication
- **template_folder** (*str*, *optional*) – path to the folder containing the jinja2 templates.

Methods

__init__ (*args, **kwargs)	Initialize self.
-----------------------------------	------------------

Attributes



beamer_correct_coords()

This method receives manual entries from a website (if the camera was incorrect) and forwards them to the beamer

beamer_make_gameimage(*coords=None, shots=False*)

Build an image with GameImage and push it to the beamer

Handle situations like showing gameresults and instructions for modes.

beamer_off()

Method to send to and display a black image on the beamer from the beamer module.

beamer_push_image(*img*)

Method to post an image to the beamer module to be displayed on the beamer.

Parameters

img (*cv2-image*) – image to be posted. Will get stretched to fullscreen on the beamer.

camera_save_image()

Deprecated, will raise AssertionError

Save the current image and (if passed) coordinates of the balls for AI training later on

forward_coords()

Collect the coordinates from the camera module and forward it to the client

This prevents the user from having to directly connect to the camera module. Also updates the beamer with the newly received coordinates and overlays them on the current GameImage object (self.gameimage).

Todo

- Experiment with timings between turning off the beamer and taking the image (especially with playing a sound simultaneously on the beamer)

gamemode_controller()

This method receives the posted data to the common gamemode API endpoint (*/gamemodecontroller*) - preprocess: select gamemode - passes it to the selected gamemode object (entrance(...) method, most of the times inherited from GameMode) - postprocess: update gameimage, send to beamer, read signal in output (handle), return output

gamemode_socket_handler(*json_data*)

This function forwards events on the “gamemode-socket” socket to the specified gamemode.

A gamemode must have a GameMode.SOCKETS dictionary matching the current state to a message handler like {"init": self.handler}

get_ball_image()

Get the image of a certain ball by number.

Send the number/name of the ball as request argument (*/ballimagenumber?n=4*)

Todo

- change endpoint to template like `/ballimage/<number>` instead of current system with args.

get_gamemode_website(*mode*)

A main level gamemode should have a website. If a gamemode exists but has no `GameMode.index_args()` implemented, return a 404 error.

index()

Renders and returns the index.html website (gamemode selection)

list_available_gamemodes()

Based on the `self.GAMEMODES` dict, generates a subset of gamemodes that have an `gamemode.index_args()` method.

render_template_camera(*file*, *kwargs*)**

Oftentimes we need to render a website that contains the camera livestream. We dont want to write it statically nor do we want to pass it everytime

save_players()

Writes the current `self.players` list to `players.json`

take_image()

Turn of the beamer and take an image.

view_csv(*file*)

Renders a single csv file as a html table and shows it. CSV files must not have an index and must be separated by tabs (`\t`). If the file does not exist or does not end in `.csv`, returns status 404 or 403

Modules

<i>Elo</i>	
<i>GameEngine</i>	
<i>GameImage</i>	
<i>GameRules</i>	
<i>GlobalAPI</i>	
<i>Player</i>	
<i>gamemodes</i>	This module provides the gamemodes, the parent class of all gamemodes and common utilities for ball interactions/analyses (<code>common_utils</code>) and the global API (<code>api_utils</code>).

===

2.1.2 Elo

Classes

<i>Elo</i> ([K])	Simple implementation of the elo rating system originally developed by Arpad Elo.
------------------	---

===

2.1.2.1 Elo

class Game.Elo.Elo(*K=30*)

Bases: object

Simple implementation of the elo rating system originally developed by Arpad Elo.

Parameters

K (*int*) – constant multiplier/penalty

__init__ (*K=30*)

Methods

<code>__init__</code> (<i>[K]</i>)	
<code>match</code> (<i>players[, winner]</i>)	Players are two dict-objects with the key "elo".
<code>propability</code> (<i>elo1, elo2</i>)	Returns the propability of player with elo1 of winning against player with elo2

match (*players, winner=0*)

Players are two dict-objects with the key "elo". Updates their elo value based on the outcome

Parameters

- **players** – list of the two player dictionaries
- **winner** – winner's index in the player list. 0.5 if draw.

propability (*elo1, elo2*)

Returns the propability of player with elo1 of winning against player with elo2

2.1.3 GameEngine

Classes

<code>GameEngine</code> (<i>[size, ballDiameter]</i>)	This class implements basic physics based calculations for determining good hits.
---	--

2.1.3.1 GameEngine

class Game.GameEngine.GameEngine(*size=(2230, 1115), ballDiameter=57*)

Bases: object

This class implements basic physics based calculations for determining good hits.

The holes get evenly spaced out along the edges of the defined size

__init__ (*size=(2230, 1115), ballDiameter=57*)

Methods

<code>__init__([size, ballDiameter])</code>	
<code>getShots(data[, group])</code>	Calculate all possible first-level (simple) shots.

Attributes

<code>group_map</code>

getShots(data, group='open')

Calculate all possible first-level (simple) shots. Returns for each ball in the specified group (except white) the coordinate of the white ball and where it should travel to, the coordinate of the ball it will hit and hole this ball will go to. This requires the white ball to exist, else it will return empty.

Calcs based on <https://www.real-world-physics-problems.com/physics-of-billiards.html>

Parameters

- **data** (*dict*) – coordinates of the balls in the typical coordinate format
- **group** (*optional str*) – only consider hits for balls in the group. Can be “open” (all balls), “half”/“striped”, “full”/“solid”, “eight”/“black”

2.1.4 GameImage

Functions

<code>ie(dic, key, default)</code>	If key in dic returns value in dic else default
------------------------------------	---

2.1.4.1 Game.GameImage.ie

`Game.GameImage.ie(dic, key, default)`

If key in dic returns value in dic else default

Classes

<code>BilliardBall(n)</code>	Class to store each Billiard Balls graphical attributes
<code>GameImage([definition, size, phys, img_cache])</code>	Class to generate an image from different game modes and other information
<code>Trickshot(definition[, size])</code>	Deprecated.

2.1.4.2 BilliardBall

class Game.GameImage.BilliardBall(*n*)

Bases: object

Class to store each Billiard Balls graphical attributes

Each ball is initiated with its number with number 16 being the white ball. Number 1-7 are full/solid, 8 is black/eight, 9-15 are half/stripped.

Special balls are dummy (17), correct (18) and incorrect (19).

colors

list of all colors (hexcodes strings) ordered

Type

list

Parameters

n (*int*, *str*) – name of the ball

__init__(*n*)

Methods

<code>__init__(n)</code>	
<code>getColor(n)</code>	Get the hexcode of the color of the ball based on the n value (number or str keys)
<code>getGroup(n)</code>	Get the group of the color of the ball based on the n value (number or str keys).
<code>getImg(d[, square])</code>	Get the image of the ball with its number, half or full and color.

Attributes

<code>colors</code>
<code>unique_map</code>
<code>white</code>

classmethod `getColor(n)`

Get the hexcode of the color of the ball based on the n value (number or str keys)

Parameters

n (*int*, *str*) – number or name of the ball

classmethod `getGroup(n)`

Get the group of the color of the ball based on the n value (number or str keys). Everything over 16 is dummy.

Parameters

n (*int*, *str*) – number or name of the ball

getImg(*d*, *square=False*)

Get the image of the ball with its number, half or full and color.

Parameters

- **d** (*int*) – Diameter of the returned image (side of square) in px
- **square** (*optional, bool*) – should the ball image be generated as a square? This is useful when you are simultaneously displaying ball positions on the beamer and inferring ball positions using the camera module. Displayed square balls are less likely to get picket up as balls compared to circular balls.

Returns

PIL.Image object

2.1.4.3 GameImage

```
class Game.GameImage.GameImage(definition=[], size=(2230, 1115), phys=1, img_cache={'feedback-form-qr':  
    'https://www.youtube.com/watch?v=XfELJU1mRMg', 'isem-logo':  
    'static/images/ISEM-only.png', 'isem-logo-big':  
    'static/images/isem_logo_big.png'})
```

Bases: object

Class to generate an image from different game modes and other information

Each image starts with a black background, as this is “neutral” on a beamer.

The image gets build using a *definition* (list). It consists of *parts* (dicts) which each can look like

```
{  
    "type": "text",  
    "text": "This is a text element, that will show up at the top and bottom of ↵  
↵ the image."  
}
```

The available types and specific fields are listed in the documentation of each *part*. Each *part* gets assigned (if not defined) a reference (field *ref*), under which it can be changed using `self.update_definition`. Some *part* types are static and will get reassigned instead of added with a different reference if they are input to `GameImage.update_definition`, unless a specific reference has been assigned.

definition

list of *parts*. The first element in the list will get drawn first, with later parts getting drawn above of it (rudimentary layering)

Parameters

- **definition** (*list*) – *definition* list as specified in the GameImage class documentation
- **size** (*tuple<int, int>*) – width, height of the goal image in pixels
- **phys** (*float*) – how many meters are 1000 pixels? (equivalent to how many mm are 1 pixel). Not really in use.
- **img_cache** (*dict*) – Images that can be used by using in their key as reference, e.g., in a part like `{"type": "text", "text": "Cached Image", "subimg": "key-from-img_cache"}` the *subimg* would be loaded from the file, url (QR-code) or PIL.Image object mapped to the key. Files get loaded upon init and the loaded cache gets copied when the object gets copied (`GameImage.copy`), minimizing file operations.

Todo

- Move the GameImage system to its entirely own module/project as an abstraction layer on top of PIL itself.
- Add GIF/video features. Maybe this would need the GameImage to be generated on the Beamer module itself.
- Add GameImage slots to minimize compute needs

```
__init__(definition=[], size=(2230, 1115), phys=1, img_cache={'feedback-form-qr':
    'https://www.youtube.com/watch?v=XfELJU1mRMg', 'isem-logo': 'static/images/ISEM-only.png',
    'isem-logo-big': 'static/images/isem_logo_big.png'})
```

Methods

<code>__init__([definition, size, phys, img_cache])</code>	
<code>arrow(start, end[, offset])</code>	Arrow pointing from start to end
<code>arrow_bottom(bottom, orientation, length[, ...])</code>	Draws an arrow by specifying the bottom point.
<code>bullseye([center, r])</code>	Draws a bullseye around the center with 3 rings, with each ring having a width of r
<code>centralImage(img)</code>	Draw an (PIL) image in the center of the image.
<code>copy()</code>	Generate a copy of the image.
<code>drawBallConnections(shots[, offset, ...])</code>	Takes the output of GameEngine.getShots and draws arrows accordingly.
<code>drawBreak([ball])</code>	Draw the basic triangle for a break and optionally the white ball for break
<code>draw_from_dict(definition[, draw])</code>	Draw the image (GameImage.img) from a list<dictionary> specifying the subparts.
<code>getImageCV2()</code>	Exports the GameImage.img (PIL.Image) as a RGB cv2 compatible np.ndarray
<code>instructionText(text[, subimg])</code>	Place text on top and flipped on the bottom of the image.
<code>line(c1, c2[, color, width])</code>	Draws a line between two points.
<code>nameTeamText(name, team)</code>	Put the name and team on the right side of the image, oriented outwards.
<code>placeAllBalls(coords)</code>	Place all balls mentioned in the data dict on the canvas self.img
<code>placeBall(pos, n[, d])</code>	Place a ball on the canvas self.img based on its number
<code>rectangle(c1, c2[, outline, width, fill])</code>	Draws a rectangle between two points as opposite corners.
<code>redraw()</code>	Draw the GameImage.img again using the definition already available as GameImage.definition
<code>rm_definition(ref)</code>	Remove an element from the definition by ref
<code>update_definition(val[, ref, subfield, ...])</code>	Adds a new part to the image definition or update an existing part.
<code>update_text(text)</code>	Specific wrapper for GameImage.update_definition to only update the text part

Attributes

static

These *part* types are static and will get reassigned instead of added with a different reference if they are input to `GameImage.update_definition`, unless a specific reference has been assigned.

arrow(*start, end, offset=0, **kwargs*)

Arrow pointing from start to end

Part description: `{"type": "arrow", "start": [123, 123], "end": {"x": 123, "y": 123}, "offset": 35}` + optional fields of `GameImage.arrow_bottom`

Parameters

- **start** (*list-like*) – Starting point (bottom)
- **end** (*list-like*) – End point (head)
- **offset** (*int, optional*) – Distance from the bottom and head to where the arrow actually start. Useful for pointing arrows between billard balls. Defaults to 0.

arrow_bottom(*bottom, orientation, length, head_width=50, line_width=10, color='white', **kwargs*)

Draws and arrow by specifying the bottom point. *bottom* as `np.array([x, y])`, *orientation* in degrees relative to +x-axis in mathematical positive direction

Part description: `{"type": "arrow_bottom", "bottom": [123, 123], "orientation": 3.14159, "length": 50, "head_width": 50, "line_width": 10, "color": "#123456"}`

Parameters

- **bottom** (*list-like*) – Starting point of the arrow (not the head)
- **orientation** (*float*) – Orientation in radians from the +x axis ($\pi/2$ would be vertical, 0 to the right)
- **length** (*float*) – Length of the arrow
- **head_width** (*int, optional*) – Width of the arrow head. Defaults to 50.
- **line_width** (*int, optional*) – Thickness of the line. Defaults to 10.
- **color** (*str, optional*) – Color as keyword or hex. Defaults to "white".

bullseye(*center=None, r=30*)

Draws a bullseye around the center with 3 rings, with each ring having a width of *r*

Part description: `{"type": "bullseye", "center": [123, 123], "r": 30}`

Parameters

- **center** (*list-like, optional*) – Center of the bullseye. Defaults to None.
- **r** (*int, optional*) – Thickness of each of the three rings. Defaults to 30.

Todo

- change *r* to be the total radius not this weird ring-thickness.

- ideally also add a number of rings parameter

centralImage(*img*)

Draw an (PIL) image in the center of the image. Currently the image gets inserted at its original size, so resizing must be done before.

Part description: `{“type”: “central-image”, “img”: “isem-logo-big”}`

Parameters

img (*path*, *PIL.Image*, *GameImage.img_cache* key) – Image the gets placed in the center

Todo

- add sizing options

copy()

Generate a copy of the image.

Returns

copy of this object

Return type

GameImage

drawBallConnections(*shots*, *offset=35*, *line_width=6*, *head_width=12*, *kwargs*)**

Takes the output of *GameEngine.getShots* and draws arrows accordingly. Each shot path gets the color of the ball it is supposed to sink (except for the black ball, which would not be visible)

Part description: `{“type”: “possible_shots”, “shots”: [shots dict]} + optional fields from GameImage.arrow and GameImage.arrow_bottom`

Parameters

- **shots** (*dict*) – output of *GameEngine.getShots* like `{“1”: {“hole”: {“x”: 123, “y”: 123}, “ball”: ..., “end-white”: ..., “white”: ...}, ...}`
- **kwargs** (*optional*) – arguments passed to *GameImage.arrow()*

drawBreak(*ball=True*)

Draw the basic triangle for a break and optionally the white ball for break

Part description: `{“type”: “break”, “ball”: True}`

Parameters

ball (*bool*) – Should the starting point of the break be drawn?

Todo

- verify the position of the starting point and break triangle

draw_from_dict(*definition*, *draw=True*)

Draw the image (*GameImage.img*) from a list<dictionary> specifying the subparts. Execution flow is from top to bottom, so lower (later) parts are on a higher image layer.

Parameters

- **definition** (*list*) – *definition* list as specified in the class documentation. If an element of the list is *None*, all *parts* in the passed *definition* list will be treated using `GameImage.update_definition`.
- **draw** (*bool*, *optional*) – Choose if the image should directly be rendered ore only the definition updated. The improvement over just reassigning `GameImage.definition` is the creation of references.

Returns

definition list with assigned references (field *ref*, only for *parts* that had none before)

Return type

list

Todo

- Change match-case statement to dict mapping type to methods

getImageCV2()

Exports the `GameImage.img` (`PIL.Image`) as a RGB cv2 compatible `np.ndarray`

Returns

RGB image

Return type

`np.ndarray`

instructionText(text, subimg=None)

Place text on top and flipped on the bottom of the image. Font size is chosen automatically.

Part description: `{"type": "text", "text": "Hello World", "subimg": "isem-logo"}`

Images can be passed as PIL image, path or `GameImage.img_cache` key. Text can be emphasized by encapsulating in ...

Todo

- Finalize emphasised formatting (color blind friendly). Maybe generalize text rendering?

Parameters

- **text** (*str*) – Text to be placed
- **subimg** (*optional PIL Image or relative path to image*) – Image to be placed directly under the text

line(c1, c2, color='white', width=3)

Draws a line between two points.

Part description: `{"type": "line", "c1": [123, 123], "c2": {"x": 123, "y": 123}, "color": "#123456", "width": 5}`

Parameters

- **c1** (*dict, iterable*) – Coordinates of one end (e.g. as `{"x": 12, "y": 34}`)

- **c2** (*dict*, *iterable*) – Coordinates of the other end
- **color** (*str*, *optional*) – Color of the line, hexcode (#112233) or keyword. Defaults to “white”.
- **width** (*int*, *optional*) – Width of the line in px. Defaults to 3.

nameTeamText(*name*, *team*)

Put the name and team on the right side of the image, oriented outwards.

Part description: {“type”: “team”, “name”: “Jon Doe”, “team”: “ISEM”}

Parameters

- **name** (*str*) – Player name
- **team** (*str*) – Team name

placeAllBalls(*coords*)

Place all balls mentioned in the data dict on the canvas self.img

Part description: {“type”: “line”, “coords”: [*coords dict*]}

Parameters

data (*dict*<*tuple*<*float*>> or *list*<*dict*>) – dictionary matching the number of a ball (key) to its position (*tuple* x,y) from the upper left in mm. Can also just be the output of `Camera.get_coords (.../v1/getcoords)`

placeBall(*pos*, *n*, *d=None*)

Place a ball on the canvas self.img based on its number

Parameters

- **pos** (*tuple*<*int*>) – (x,y) center of the ball in mm from the top left
- **n** (*int* or *str*) – number of the ball
- **d** (*optional int*) – diameter of the ball in pixels, automatically determined to be accurate according to self.phys and

rectangle(*c1*, *c2*, *outline='white'*, *width=5*, *fill=None*)

Draws a rectangle between two points as opposite corners.

Part description: {“type”: “rectangle”, “c1”: [123, 123], “c2”: {“x”: 123, “y”: 123}, “outline”: “#123456”, “width”: 5, “fill”: None}

Parameters

- **c1** (*iterable*) – Coordinates of one end (x, y)
- **c2** (*iterable*) – Coordinates of the other end
- **outline** (*str*, *optional*) – Color of the line, hexcode (#112233) or keyword. Defaults to “white”.
- **width** (*int*, *optional*) – Width of the border in px. Defaults to 5.
- **fill** (*str*, *optional*) – Color of the line, hexcode (#112233) or keyword. Defaults to None.

redraw()

Draw the GameImage.img again using the definition already available as GameImage.definition

rm_definition(ref)

Remove an element from the definition by ref

static

These *part* types are static and will get reassigned instead of added with a different reference if they are input to GameImage.update_definition, unless a specific reference has been assigned.

update_definition(val, ref=None, subfield=None, new_instance=False, layer=-1, remove=False)

Adds a new part to the image definition or update an existing part.

If ref is None or no current entry has the same ref, actually adds a new part. Otherwise it finds a part with the same reference (either passed as argument *ref* or in *val* as field *ref*). If subfield != None, looks for the key in a part with given ref and only updates that.

Parameters

- **val** (*dict*, *any*) – If *subfield* is *None*, this gets treated as an entire *part*, otherwise it is the new value of the field *subfield* in the part with reference *ref*. If it has the field “*remove*”: *True* and a set *ref* field, the part with that *ref* will be removed, calls GameImage.rm_definition.
- **ref** (*str*, *optional*) – reference of a part to update. Defaults to *None*.
- **subfield** (*str*, *optional*) – Key of the field in an existing part to update. Defaults to *None*.
- **new_instance** (*bool*, *optional*) – Unused. Defaults to *False*.
- **layer** (*int*, *optional*) – At what layer should this be inserted? Defaults to -1 for the last element (highest layer)
- **remove** (*bool*, *optional*) – Remove the element that has the given *ref*, calls GameImage.rm_definition. Defaults to *False*.

update_text(text)

Specific wrapper for GameImage.update_definition to only update the text part

2.1.4.4 Trickshot

class Game.GameImage.Trickshot(*definition*, *size*=(2230, 1115))

Bases: object

Deprecated.

The trickshot challenge this belongs to has been superseded by the Longst Break challenge, since that can be played more reliably.

Class to draw a trickshot with all its belongings onto the gameimage based on the json/dict defining the trickshot (see subfolder trickshots).

Parameters

- **definition** (*dict*) – dict describing the trickshot, most likely loaded from a trickshots/*.json
- **size** (*optional tuple (width, height)*) – the size of the image in mm = pixel. Sample down later.

__init__(*definition*, *size*=(2230, 1115))

Methods

<code>__init__(definition[, size])</code>	
<code>drawCue([linewidth, indicatorDistance, ...])</code>	Draws the suggested position for the cueue including an indicator of the power level of the hit.
<code>drawPolygons([linewidth])</code>	Directy draws all polygons from definition on the objects image
<code>getHitHints([d])</code>	Hit hints are hints on where to hit the ball (shown from the back) to get a desired trajectory like backspin.
<code>getTrickshotImage()</code>	Draws everything from the definition and returns a cv2 image.
<code>getTrickshotImageCV2()</code>	
<code>placeBalls([anonymize, diameter])</code>	Place all the ball from the definition onto the canvas
<code>placeHitHints()</code>	Generate and place the cue hints on the main image

drawCue(*linewidth=20, indicatorDistance=180, indicatorLength=200*)

Draws the suggested position for the cueue including an indicator of the power level of the hit.

Parameters

- **linewidth** (*int, optional*) – Width of the cue and indicator. Defaults to 20.
- **indicatorDistance** (*int, optional*) – Distance from the indicator to the cueue. Defaults to 180.
- **indicatorLength** (*int, optional*) – Length of the indicator. Defaults to 200.

drawPolygons(*linewidth=10*)

Directy draws all polygons from definition on the objects image

getHitHints(*d=100*)

Hit hints are hints on where to hit the ball (shown from the back) to get a desired trajectory like backspin.

Information on the hit hint gets pulled from Trickshot.d[“hit”][“x” and “y”] (d = definition from init).

Parameters

d (*int, optional*) – Diameter of the ball that should be displayed. Defaults to 100.

Returns

`_description_`

Return type

`_type_`

getTrickshotImage()

Draws everything from the definition and returns a cv2 image.

The background gets set to black, but this is not a problem as thisis “off” on a beamer.

placeBalls(*anonymize=False, diameter=57*)

Place all the ball from the definition onto the canvas

Parameters

anonymize (*optional bool*) – if true, every ball that is not white will be replaced by the same color, as their number is rarely important for trickshot

placeHitHints()

Generate and place the cue hints on the main image

The hit hint (generated by Trickshot.getHitHints) gets placed four times, on each edge once.

2.1.5 GameRules

Classes

<i>GameRules()</i>	This class implements rules for 8 ball pool, determining the state of the game based on passed coordinates.
--------------------	---

2.1.5.1 GameRules

class Game.GameRules.GameRules

Bases: object

This class implements rules for 8 ball pool, determining the state of the game based on passed coordinates.

The decision on who can break is handled by doing a randomized precision shot.

The state variable (string) determines the image/mode to be projected on the table by the GameImage object

`__init__()`

Methods

<i><code>__init__()</code></i>	
<i><code>change_player()</code></i>	
<i><code>determine_start(data1)</code></i>	Determine of the players should play the break by having them do a precision shot to a random point on the screen
<i><code>enter_round(data)</code></i>	Register a round and determine the next steps
<i><code>filterData(group, data[, white])</code></i>	Filter only the balls of the current player's group to be displayed/shots to be calculated
<i><code>get_current_player_name([theother])</code></i>	
<i><code>get_current_players_ball_coords()</code></i>	
<i><code>get_display_string()</code></i>	Compiles a nice string to be displayed on the beamer
<i><code>get_distance(coords1, coords2)</code></i>	Returns the distance (2 norm) between two coordinates (ball coord dicts)
<i><code>get_remaining_balls(data)</code></i>	Filters the data by groups and counts the ball per group (full, half, eight)
<i><code>handle_win()</code></i>	This handles the win of a player, updating their elo etc.
<i><code>init_start(player1, player2)</code></i>	Actually starting the game

`determine_start(data1)`

Determine of the players should play the break by having them do a precision shot to a random point on the screen

Parameters

data1 – dict like {"player1": {"name": "ball-5", "x": 123, "y": 456, ...}} with x and y in mm

enter_round(data)

Register a round and determine the next steps

Parameters

data (*dict*) – coords dictionary of balls

filterData(group, data, white=True)

Filter only the balls of the current player's group to be displayed/shots to be calculated

Parameters

- **group** (*str*) – the group of the balls to highlight ("full", "half", "eight")
- **data** (*dict*) – coords from the camera
- **white** (*optional bool*) – include the white ball when filtering the balls

get_display_string()

Compiles a nice string to be displayed on the beamer

get_distance(coords1, coords2)

Returns the distance (2 norm) between two coordinates (ball coord dicts)

get_remaining_balls(data)

Filters the data by groups and counts the ball per group (full, half, eight)

handle_win()

This handles the win of a player, updating their elo etc. NOT IMPLEMENTED YET!

init_start(player1, player2)

Actually starting the game

This initialises the process to decide who will break. :param player1, player2: dict with name, team, elo (from database)

2.1.6 GlobalAPI

Classes

<i>GlobalAPI</i> (configpath)	Interface class with the API provided by the main global server
-------------------------------	---

2.1.6.1 GlobalAPI

class Game.GlobalAPI.**GlobalAPI**(*configpath*)

Bases: object

Interface class with the API provided by the main global server

__init__(*configpath*)

Methods

<code>__init__(configpath)</code>	
<code>endpoint(endpoint)</code>	Returns full address from endpoint passed as '/a/b' or 'a/b'
<code>get(endpoint)</code>	Only send information relevant for auth
<code>is_server_online()</code>	
<code>post(endpoint, json)</code>	

endpoint(*endpoint*)

Returns full address from endpoint passed as '/a/b' or 'a/b'

get(*endpoint*)

Only send information relevant for auth

2.1.7 Player

Classes

<code>Player(json_data)</code>	Implementing a the player class and storing all relevant values
--------------------------------	---

2.1.7.1 Player

class `Game.Player.Player(json_data)`

Bases: `object`

Implementing a the player class and storing all relevant values

CURRENTLY NOT IN USE

Parameters

json_data (*dict*) – parsed version of the player entry in players.json

`__init__(json_data)`

Methods

<code>__init__(json_data)</code>	
<code>make_id()</code>	Concat name and team and stores it as id
<code>to_dict()</code>	

make_id()

Concat name and team and stores it as id

2.1.8 gamemodes

This module provides the gamemodes, the parent class of all gamemodes and common utilities for ball interactions/analyses (common_utils) and the global API (api_utils).

In the directory, the common folder individual gamemode resources is provided as *gamemodes/resources*. All files in this folder can be accessed by the `Game.view_csv` method/endpoint, if they end in `.csv`.

Each gamemode is designed as a python class with a *dict* defining the game flow called *GameMode.TREE*. An example for the *TREE* is copied from `distance.Distance`:

```
self.TREE = {
    "init": [
        lambda inp: self.check_starting_positions(inp, starting_positions=[self.starting_
        ↪point_raw]), # function to call on input: self.check_starting_points is provided by
        ↪the parent class GameMode. A lambda expression is used to unify the input to only take
        ↪in one argument.
        {"True": "strike", "False": "init"}, # based on the output, what should state
        ↪should be assumed next?
        lambda: [
            # GameImage.definition that should be shown when this is the state. As
            ↪lambda expression to be evaluated on call (dynamically change values from self.
            ↪[attribute] like self.starting_point). See GameImage documentation for details.
            {
                "type": "text",
                "text": "Distance Challenge: Place the ball on the starting point",
                "subimg": "isem-logo"
            },
            {
                "type": "balls", # starting point
                "coords": self.starting_point,
                "ref": "balls-base" # manually set the reference so it does not get
            ↪overwritten when displaying other balls
            }
        ],
        ["Start"] # describe name of button and special inputs that should be shown on
        ↪the client. Gamemodes that are parts of the KP2 gamemode (KP2.GAMEMODES) have their
        ↪inputs be dynamically generated based on this field. See GameMode.build_HTML for more
        ↪information.
    ],
    "strike": [
        self.calculate_score, #
        {"True": "finished"}, # no other outputs possible
        lambda: [ # as soon as the state (self.state) is "strike", this gets assigned to
        ↪Game.gameimage and displayed on the beamer
        {
            "type": "text",
            "text": "Distance Challenge: Strike the ball!",
            "subimg": "isem-logo"
        },
        {
            "type": "balls", # starting point
            "coords": self.starting_point,
            "ref": "balls-base"
        },
    ],
}
```

(continues on next page)

(continued from previous page)

```

        {
            "type": "arrow_bottom",
            "length": 300,
            "orientation": 180,
            "bottom": {"x": self.starting_point["white"]["x"] - 100, "y": self.
↪starting_point["white"]["y"]},
            "ref": "arrow-tooltip"
        }
    ],
    ["Measure", #IDs of this button gets set to [self.gamemode_name]-[state/step_
↪(here `strike`)]-submit
        { # special input fields needed in this step:
            "type": "number", # input fields are required to be changed at least once_
↪for the main button (here: Measure) to become selectable. `types` are just html input_
↪types.
            "name": "collisions", # generates a number input field which will be_
↪available in inp["collisions"]
            "placeholder": "Collisions", # IDs of input get set to [self.gamemode_name]-
↪[state/step (here `strike`)]-[name]
        }]
    ],
    # a "finished" action is automatically added by the parent class GameMode if not_
↪already specified.
}

```

The input *inp* that is piped into is a standardized *dict* send by the client, consisting of at least the fields

```

inp = {
    "gmode": "distance", # name of the gamemode this input should be passed to (names as_
↪set as keys in Game.GAMEMODES)
    "action": "game", # action for what this input should be used for:
        # game: pass it to the normal GameMode.entrance function, will normally get_
↪passed to the specified function in GameMode.TREE[GameMode.state][0]
        # settings: pass it to the GameMode.settings function (this is specified/decided_
↪in GameMode.entrance)
        # show: pass it to the GameMode.show function, which returns a current GameImage_
↪object without modifying anything of the GameMode (this is specified in Game.gamemode_
↪controller)
    # all other fields are specific to the gamemode selected
}

```

The client website sends this *json*-data to the API endpoint */gamemodecontroller* of the Game module. For testing, this can also be done by API testing systems like postman or a simple *curl* request.

Gamemodes need to be registered with the main game module (*Game/__init__.py*)

Todo

- move common_utils to another module, since it can cause problems with circular imports.

Modules

GameMode
GameModel
api_utils
common_utils
distance
dummy
final_competition
kp2
local_game
longest_break
online_game
precision
single_break

2.1.8.1 GameMode**Classes**

<i>GameMode()</i>	The parent class to all gamemodes.
-------------------	------------------------------------

2.1.8.1.1 GameMode

class Game.gamemodes.GameMode.**GameMode**

Bases: object

The parent class to all gamemodes. Provides common implementations like show and history management

state

the current state of the GameMode. Usually starts with *init* and ends with *finished*, after which it resets.

Type

str

TREE

mapping of the current state to actions that should be taken when the next user input arrives. The example below is from a Distance object:

```

self.TREE = {
    "init": [
        lambda inp: self.check_starting_positions(inp, starting_positions=[self.
↪starting_point_raw]), # function to call on input: self.check_starting_points_

```

(continues on next page)

(continued from previous page)

```

↪ is provided by the parent class GameMode. A lambda expression is used to
↪ unify the input to only take in one argument.
    {"True": "strike", "False": "init"}, # based on the output of the
↪ previous function, what should state should be assumed next?
    lambda: [
        # GameImage.definition that should be shown when this is the state.
↪ As lambda expression to be evaluated on call (dynamically change values from
↪ self.[attribute] like self.starting_point). See GameImage documentation for
↪ details.
        {
            "type": "text",
            "text": "Distance Challenge: Place the ball on the starting
↪ point",
            "subimg": "isem-logo"
        },
        {
            "type": "balls", # starting point
            "coords": self.starting_point,
            "ref": "balls-base" # manually set the reference so it does not
↪ get overwritten when displaying other balls
        }
    ],
    ["Start"] # describe name of button and special inputs that should be
↪ shown on the client. Gamemodes that are parts of the KP2 gamemode (KP2.
↪ GAMEMODES) have their inputs be dynamically generated based on this field.
↪ See GameMode.build_HTML for more information.
    ],
    "strike": [
        self.calculate_score, #
        {"True": "finished"}, # no other outputs possible
        lambda: [ # as soon as the state (self.state) is "strike", this gets
↪ assigned to Game.gameimage and displayed on the beamer
            {
                "type": "text",
                "text": "Distance Challenge: Strike the ball!",
                "subimg": "isem-logo"
            },
            {
                "type": "balls", # starting point
                "coords": self.starting_point,
                "ref": "balls-base"
            },
            {
                "type": "arrow_bottom",
                "length": 300,
                "orientation": 180,
                "bottom": {"x": self.starting_point["white"]["x"] - 100, "y":
↪ self.starting_point["white"]["y"]},
                "ref": "arrow-tooltip"
            }
        ],
    ["Measure", #IDs of this button gets set to [self.gamemode_name]-[state/

```

(continues on next page)

(continued from previous page)

```

↪step (here `strike`)]-submit
    { # special input fields needed in this step:
      "type": "number", # input fields are required to be changed at
↪least once for the main button (here: Measure) to become selectable. `types`
↪are just html input types.
      "name": "collisions", # generates a number input field which will
↪be available in inp["collisions"]
      "placeholder": "Collisions", # IDs of input get set to [self.
↪gamemode_name]-[state/step (here `strike`)]-[name]
    }]
  ],
  # a "finished" action is automatically added by the parent class GameMode
↪if not already specified.
}

```

Type
dict

__init__()

Methods

<code>__init__()</code>	
<code>build_HTML()</code>	Based on self.TREE, build the input fields for this gamemode flow.
<code>check_starting_positions(inp[, ...])</code>	Checks if the ball(s) is/are placed on the starting point(s).
<code>entrance(inp)</code>	This is the central function of every gamemode (can be overloaded).
<code>get_history()</code>	Gets the history table from self.history_file.
<code>history([history, add, get_semester])</code>	Get the player/team rankings.
<code>reset([keep_settings, inplace])</code>	Reset the gamemode object but keep changes made to the settings.
<code>save_history(history)</code>	Writes a passed dataframe to self.history_file (Path object) as csv (tab-separated, without an index).
<code>save_json_history(history)</code>	Appends a passed dict to a list of dicts in self.json_history_file.
<code>show([inp])</code>	When (re)entering the gamemode, determine what gameimage to show.
<code>socket_event(json_data)</code>	Currently not in use.

build_HTML()

Based on self.TREE, build the input fields for this gamemode flow. Returns the HTML content inside of the fieldset container as well as the name of the gamemode

Inputs that get rendered are defined by the fourth (starting at 0: 3rd) entry in a self.TREE value field. An example is

```

["Measure", #IDs of this button gets set to [self.gamemode_name]-[state/step
↪(here `strike`)]-submit
    { # special input fields needed in this step:

```

(continues on next page)

(continued from previous page)

```

    "type": "number", # input fields are required to be changed at least once,
    ↪ for the main button (here: Measure) to become selectable. `types` are just
    ↪ html input types.
    "name": "collisions", # generates a number input field which will be
    ↪ available in inp["collisions"]
    "placeholder": "Collisions", # IDs of input get set to [self.gamemode_name]-
    ↪ [state/step (here `strike`)]-[name]
  }]

```

1. The first field “Measure” defines a checkbox with the text “Measure”. Clicking on it starts the interaction flow to get/update the coordinates. If this field is *None*, there will be no checkbox displayed.
2. **The second field defines an arbitrary html input element. All key-value pairs should be valid attributes of a html input element (see <https://developer.mozilla.org/en-US/docs/Web/HTML/Reference/Elements/input>).**
 1. There can be an arbitrary amount of special input fields like this
3. Input buttons and checkboxes trigger a js-event to collect the inputs of all other specified inputs in this step/state and send their value as [name]: [value] pairs in a json-object to the specified gamemode’s entrance method. The [value] of checkbox inputs gets translated to true/false.

Returns

html-code of all steps joined with `<hr>` and the name of the gamemode (self.gamemode_name)

Return type

str, str

check_starting_positions(inp, starting_positions=[], tolerance=50, update_gameimage=None)

Checks if the ball(s) is/are placed on the starting point(s). Returns True if it is correct, False if not. Local returns contains “gameimage-updates” with balls to signal misplacements. Forward returns contains the message what is wrong or correct. See common_utils.check_positions for more details. For starting positions determined after init, use a (lambda) function to provide the starting coords.

Parameters

- **inp** (dict) – input from the client, must contain key “coordinates”
- **starting_positions** (dict/list/function) – the positions of the balls at the start. For syntax see common_utils.check_positions.
- **tolerance** (int) – max radial distance a ball can have from a position before it is determined misplaced. Given in the same scale as the coordinates, which usually are mm (1px = 1mm).
- **update_gameimage** (GameImage) – if set, also updates the passed GameImage objects definition (call by reference) according to the local_returns[“gameimage-updates”]. Only updates the text field. Useful when the GameImage displayed is set in self.TREE as self.gameimage.definition.

Returns

within_tolerance: True if all balls are placed correctly, False otherwise || local_returns: special stuff to handle on the server. Mostly contains “gameimage-updates”: [{},...] to update the gameimage | forward_returns: message that gets send to the client

Return type

bool, dict, dict

entrance(*inp*)

This is the central function of every gamemode (can be overloaded). This method is called by Game.gamemode_controller, which also handles the return values.

In normal gamemodes (which do not overload this), the action is determined by the structure of GameMode.TREE

Parameters

inp (*dict*) – json data input from the client. Usually posted to the endpoint */gamemodecontroller*.

Returns

the output that gets send to the client as a response (must contain the fields “signal”, which is usually the current/new self.state value). The returned GameImage object is the new image that should get shown as a response to the request. The final string is the name of a sound file (just the file name, without path or ending) on the Beamer module that should get played (if it is None, nothing will get played).

Return type

dict, *GameImage*, (str | None)

get_history()

Gets the history table from self.history_file. If the file does not exist, returns an empty dataframe

Returns

history or empty dataframe

Return type

pd.DataFrame

history(*history=None, add=None, get_semester=None*)

Get the player/team rankings.

Returns a dictionary that can be used to generate html code for showing the list and podium. History files are organized inside the *resources* directory, always starting with the stem of the gamemode filename (kp2.py -> *kp2_...*). History file must be name [gamemode]_history.csv. If adding to the history, set add to a dictionary containing all fields you want to save (must contain *player*, *team* and *score*). A timestamp automatically gets added.

When a new entry was added, the history gets saved.

See Game/static/history_handler.js or Game/templates/base.html for the usage of this output in JavaScript or jinja2 templates.

Parameters

- **history** (*pd.DataFrame* | *None*) – A table containin a history entry in each row. It must at least contain the columns *player*, *team* and *score*. If it is None, the table gets loaded from the .csv (tsv) path specified in *GameMode.history_file*. If this file does not exist (yet), creates an empty pd.DataFrame.
- **add** (*pd.DataFrame* | *json* | *None*) – A history entry with the same field as the history should have to get added to the history. If it is unset (None), nothing gets added. If it is set, the returned dict has a field *single_new_index* of the row number where the new entry has been inserted (in the returned sorted table, in the saved table it will be appended at the end).
- **get_semester** (*int* | *str* | *None*) – If set (not None), the returned tables will only contain entries with the same value in the *semester* column. Requires the *semester* column to exist, as in the KP2 mode.

Returns

a dictionary with the fields *single_table* (list of lists with *player*, *team*, *score* in each row, if existing also *semester* and *attestation*), *single_columns* (list of the name of the columns in *single_table*), *team_table* (list of lists always containing the *team* and *score*, which is averaged across all members of the team.). All tables are sorted with the highest score on top. If a new entry was added, the field *single_new_index* (int) also exists with the index of the new entry in the sorted *single_table*

Return type

dict

reset(*keep_settings=True*, *inplace=False*, ***kwargs*)

Reset the gamemode object but keep changes made to the settings. If reset inplace, overwrites itself without returning itself. With *inplace=True*, it returns valid values useful for usage as *lambda inp: self.reset(inplace=True)* in *GameMode.TREE*, as used in e.g. the default *finished* step from *GameMode.__init__*.

Returns

the reinitialized gamemode OR (False, {}, {}) as “neutral” output for calls in *GameMode.entrance* (if *inplace=True*)

Return type

GameMode or (bool, dict, dict)

save_history(*history*)

Writes a passed dataframe to *self.history_file* (Path object) as csv (tab-separated, without an index). Also writes a .xlsx file with the same name, just different extension.

Parameters

history (*pd.DataFrame*) – The dataframe to get saved. Usually contains the history of all rounds of the specific gamemode, with nested data being normalized (see *pd.json_normalize*)

save_json_history(*history*)

Appends a passed dict to a list of dicts in *self.json_history_file*.

Parameters

history (*dict*) – arbitrary dictionary, usually a dump of all information of the current round.

show(*inp={}*)

When (re)entering the gamemode, determine what gameimage to show.

If the gamemode contains children gamemodes, they must be organized in a dictionary named *self.GAMEMODES* with the key *self.SUBSELECTOR* in the input *inp* selecting the child gamemode by key of the dictionary.

Returns

a copy of the *GameImage* object of this *GameMode* object or a child

Return type

GameImage

socket_event(*json_data*)

Currently not in use. Using *self.SOCKETS* handle the socket event. Pipes the input into the function registered in *self.SOCKETS[self.state]*.

2.1.8.2 GameModel

Classes

<code>Game(player1, player2)</code>	This is the general game simulation for a typical game of billiards
-------------------------------------	---

2.1.8.2.1 Game

class `Game.gamemodes.GameModel.Game(player1, player2)`

Bases: `object`

This is the general game simulation for a typical game of billiards

`__init__(player1, player2)`

Methods

<code>__init__(player1, player2)</code>	
<code>change_player()</code>	Based on who is the current player (player1 or player2), this switches it.
<code>evaluate_break(coordinates1, coordinates2)</code>	Chooses the player that can break based on two precision shots.
<code>evaluate_play(auth, coordinates)</code>	Based on common billard rules, control what should happen next.
<code>generate_shootout()</code>	This generates two ball positions (white and dummy).

`change_player()`

Based on who is the current player (player1 or player2), this switches it. The other player will now be the active player (accessible as `self.active_player`) and the previously active player will now be inactive (`self.inactive_player`).

`evaluate_break(coordinates1, coordinates2)`

Chooses the player that can break based on two precision shots.

To decide who of the players should play the break, both players play a precision shot between two semi-random points (in `self.break_points`). The player that is closer to the blue ball wins and can play the break.

Parameters

- **`coordinates1`** (*dict*) – Coordinates of the shot of the first player
- **`coordinates2`** (*dict*) – Coordinates of the shot of the second player

Returns

report with important messages like `{“winner”: self.active_player, “distance1”: distance of the first player, “distance2”: ... }`

Return type

`dict`

evaluate_play(*auth, coordinates*)

Based on common billard rules, control what should happen next.

This method receives the coordinates after a hit and evaluates whether the player can play again, has made a foul or won/lost the game.

Parameters

- **auth** (*str*) – 32-byte token, must match the current *self.active_player["token"]*, otherwise this is an illegal request (only the active player should be able to hand in a hit)
- **coordinates** (*dict*) – Coordinates of the balls of the current position on the billard table.

Returns

message (like a log), a list matching a new GameImage.definition (mostly used for iterating using GameImage.update_definition) and an action list (which is currently not used, could be used for arbitrary outputs in the future)

Return type

str, list, list

generate_shootout()

This generates two ball positions (white and dummy). The player that gets closer to the dummy position starting from the white position wins the break

2.1.8.3 api_utils

Classes

<i>API</i> (env)	Provides a common interface for communicating with the global API
<i>ApiAuth</i> (TID, TAUTH, ADDRESS, PORT[, PID, ...])	Provides a common data interface for transferring authentication data

===

2.1.8.3.1 API

class Game.gamemodes.api_utils.**API**(env)

Bases: object

Provides a common interface for communicating with the global API

__init__(env)

Methods

__init__ (env)	
<i>add_player</i> (name, team)	Add a player of a certain team.
<i>evaluate_break</i> (coordinates)	Send the coordinates to the server and return the decision ("won" / "lost")
<i>evaluate_play</i> (coordinates)	Send the coordinates to the server and return the decision ("go_on" / "change" / "lost" / "won")

continues on next page

Table 30 – continued from previous page

<code>get_all_tables()</code>	Load all table names/locations/ID as a list from the API
<code>get_players_from_team(table_id)</code>	Load all players that are on a certain table
<code>post(endpoint, data)</code>	
<code>start_game(player, team[, opponent_table])</code>	Posts the player and team as well as the opponent table (if existing).

add_player(*name, team*)

Add a player of a certain team. Automatically adds id, origin_table

evaluate_break(*coordinates*)

Send the coordinates to the server and return the decision (“won” / “lost”)

evaluate_play(*coordinates*)

Send the coordinates to the server and return the decision (“go_on” / “change” / “lost” / “won”)

get_all_tables()

Load all table names/locations/ID as a list from the API

get_players_from_team(*table_id*)

Load all players that are on a certain table

start_game(*player, team, opponent_table=None*)

Posts the player and team as well as the opponent table (if existing). If it is not set, just look

2.1.8.3.2 ApiAuth

```
class Game.gamemodes.api_utils.ApiAuth(TID: str, TAUTH: str, ADDRESS: str, PORT: str, PID: str = "",
                                         PAUTH: str = "", GID: str = "")
```

Bases: object

Provides a common data interface for transferring authentication data

__init__(*TID: str, TAUTH: str, ADDRESS: str, PORT: str, PID: str = "", PAUTH: str = "", GID: str = ""*) → None

Methods

<code>__init__</code> (<i>TID, TAUTH, ADDRESS, PORT[, PID, ...]</i>)	
<code>json</code> ()	Return all set fields (= not default value) as a dict

Attributes

GID
PAUTH
PID
TID

continues on next page

Table 32 – continued from previous page

TAUTH
ADDRESS
PORT

json()

Return all set fields (= not default value) as a dict

2.1.8.4 common_utils**Functions**

<i>ball_distance</i> (coords, b1, b2)	From a coords dict, calculates the distance between two given balls
<i>check_positions</i> (real, goal[, tolerance])	Check if each ball of two sets of balls are close enough to each other (tolerance in mm).
<i>classify_region</i> (coord, region_img, translator)	Based on a loaded region image (bw image of the size of the playing field with a color for each region), classifies the region of the coords given.
<i>coord_to_vec</i> (coord)	Transforms a {"x": 0, "y": 0} dict to a np.ndarray [x, y]
<i>coords_report</i> (coordsNew, coordsOld)	Find the differences between new and old coordinates and report on sunken balls.
<i>get_border_intercept</i> (coordStart, coordPass)	Determines a point somewhere outside the playing field on a line drawn from a starting point through a passing point.
<i>is_close_enough</i> (c1, c2[, tolerance])	Are the two coordinates close enough too each other? tolerance is given in mm (as are coordinates)
<i>load_challenge_files</i> (glob_path[, ...])	looks for all files matching the glob pattern passed (see Python's glob module) and loads them with load_coordinate_file.
<i>load_coordinate_file</i> (path)	load a coordinate file (like longest break challenges) from a file.
<i>metric_distance</i> (c1, c2)	Metric distance between two coordinates each containing {"x": 0, "y": 0}
<i>metric_distance_closest</i> (coords, c2)	Calculates the distance between one point and all points in the coords.
<i>project_line</i> (coord, corner1, corner2)	Draws a mathematical line segment between the two corners and finds the closest point to the passed coord on the line.
<i>project_on_segment</i> (coord, corner1, corner2)	Based on two ends of a line segment, this returns a True if the passed coord projects onto the line segment (between the points) and False otherwise.
<i>vec_to_coord</i> (vec)	Transforms a np.ndarry [x,y] into a dict {"x": 0, "y": 0}

2.1.8.4.1 Game.gamemodes.common_utils.ball_distance

`Game.gamemodes.common_utils.ball_distance(coords, b1, b2)`

From a coords dict, calculates the distance between two given balls

2.1.8.4.2 Game.gamemodes.common_utils.check_positions

Game.gamemodes.common_utils.**check_positions**(*real*, *goal*, *tolerance=50*)

Check if each ball of two sets of balls are close enough to each other (tolerance in mm). If the balls are in the format {"white": {...}, "1": {...}}, it checks if each ball is on the corresponding goal ball with the same name. If the balls are just grouped (half, full, white, eight in the "group" field), check if they are within tolerance of a goal ball with the same group. If there are no identifications, just check if every ball is on any goal ball position.

Returns

bool (if matches), str (message why it failed), dict (coords of real balls with "incorrect"/"correct" name to display. dict just for compatability, dict keys are unimportant)

2.1.8.4.3 Game.gamemodes.common_utils.classify_region

Game.gamemodes.common_utils.**classify_region**(*coord*, *region_img*, *translator*)

Based on a loaded region image (bw image of the size of the playing field with a color for each region), classifies the region of the coords given. Needs a translator dictionary (mapping color to region name)

2.1.8.4.4 Game.gamemodes.common_utils.coord_to_vec

Game.gamemodes.common_utils.**coord_to_vec**(*coord*)

Transforms a {"x": 0, "y": 0} dict to a np.ndarray [x, y]

2.1.8.4.5 Game.gamemodes.common_utils.coords_report

Game.gamemodes.common_utils.**coords_report**(*coordsNew*, *coordsOld*)

Find the differences between new and old coordinates and report on sunken balls.

2.1.8.4.6 Game.gamemodes.common_utils.get_border_intercept

Game.gamemodes.common_utils.**get_border_intercept**(*coordStart*, *coordPass*, *size=(2230, 1115)*)

Determines a point somewhere outside the playing field on a line drawn from a starting point through a passing point. Used for drawing.

2.1.8.4.7 Game.gamemodes.common_utils.is_close_enough

Game.gamemodes.common_utils.**is_close_enough**(*c1*, *c2*, *tolerance=50*)

Are the two coordinates close enough too each other? tolerance is given in mm (as are coordinates)

2.1.8.4.8 Game.gamemodes.common_utils.load_challenge_files

Game.gamemodes.common_utils.**load_challenge_files**(*glob_path*, *sort_difficulty=False*)

looks for all files matching the glob pattern passed (see Python's glob module) and loads them with load_coordinate_file. Use this to load challenge files.

Parameters

sort_diffulty (*bool*) – decide if the returned list should be sorted by the difficulty field (easiest come first). Otherwise, sorts alphabetically by name to achieve consistency.

2.1.8.4.9 Game.gamemodes.common_utils.load_coordinate_file

Game.gamemodes.common_utils.**load_coordinate_file**(*path*)

load a coordinate file (like longest break challenges) from a file. See coordinate file reference for details on file formatting.

2.1.8.4.10 Game.gamemodes.common_utils.metric_distance

Game.gamemodes.common_utils.**metric_distance**(c1, c2)

Metric distance between two coordinates each containing {"x": 0, "y": 0}

2.1.8.4.11 Game.gamemodes.common_utils.metric_distance_closest

Game.gamemodes.common_utils.**metric_distance_closest**(coords, c2)

Calculates the distance between one point and all points in the coords. Returns the closest coord and distance.

2.1.8.4.12 Game.gamemodes.common_utils.project_line

Game.gamemodes.common_utils.**project_line**(coord, corner1, corner2)

Draws a mathematical line segment between the two corners and finds the closest point to the passed coord on the line. If the projection would be outside the line segment, it returns the closest corner.

Parameters

- **coord** (*dict*/*np.array*) – coordinates, either a dict like {"x": 123, "y": 456} or np.array([123, 456]). All must have the same type.
- **corner1** (*dict*/*np.array*) – coordinates, either a dict like {"x": 123, "y": 456} or np.array([123, 456]). All must have the same type.
- **corner2** (*dict*/*np.array*) – coordinates, either a dict like {"x": 123, "y": 456} or np.array([123, 456]). All must have the same type.

2.1.8.4.13 Game.gamemodes.common_utils.project_on_segment

Game.gamemodes.common_utils.**project_on_segment**(coord, corner1, corner2)

Based on two ends of a line segment, this returns a True if the passed coord projects onto the line segment (between the points) and False otherwise. Additionally, returns the distance of the point to the line segment (if the first is True, else int(1e9))

Parameters

- **coord** (*dict*/*np.array*) – coordinates, either a dict like {"x": 123, "y": 456} or np.array([123, 456]). All must have the same type.
- **corner1** (*dict*/*np.array*) – coordinates, either a dict like {"x": 123, "y": 456} or np.array([123, 456]). All must have the same type.
- **corner2** (*dict*/*np.array*) – coordinates, either a dict like {"x": 123, "y": 456} or np.array([123, 456]). All must have the same type.

2.1.8.4.14 Game.gamemodes.common_utils.vec_to_coord

Game.gamemodes.common_utils.**vec_to_coord**(vec)

Transforms a np.ndarry [x,y] into a dict {"x": 0, "y": 0}

2.1.8.5 distance

Classes

Distance([w, h])

The goal is to cover the highest possible distance (including bounces of the walls) on the playing field along the horizontal axis

2.1.8.5.1 Distance

class Game.gamemodes.distance.**Distance**(w=2230, h=1115)

Bases: *GameMode*

The goal is to cover the highest possible distance (including bounces of the walls) on the playing field along the horizontal axis

__init__(w=2230, h=1115)

Methods

<i>__init__</i> ([w, h])	
<i>build_HTML</i> ()	Based on self.TREE, build the input fields for this gamemode flow.
<i>calculate_score</i> (inp)	Horizontal distance in mm is the score
<i>check_starting_positions</i> (inp[, ...])	Checks if the ball(s) is/are placed on the starting point(s).
<i>entrance</i> (inp)	This is the central function of every gamemode (can be overloaded).
<i>entrance_old</i> (inp)	init -> register start -> [shoot and enter # of border collisions, user clicks enter] -> calculate distance (finish)
<i>get_history</i> ()	Gets the history table from self.history_file.
<i>history</i> ([history, add, get_semester])	Get the player/team rankings.
<i>place_ball</i> (inp)	Checks if the ball is placed on the starting point.
<i>reset</i> ([keep_settings, inplace])	Reset the gamemode object but keep changes made to the settings.
<i>save_history</i> (history)	Writes a passed dataframe to self.history_file (Path object) as csv (tab-separated, without an index).
<i>save_json_history</i> (history)	Appends a passed dict to a list of dicts in self.json_history_file.
<i>show</i> ([inp])	When (re)entering the gamemode, determine what gameimage to show.
<i>socket_event</i> (json_data)	Currently not in use.

calculate_score(inp)

Horizontal distance in mm is the score

entrance_old(inp)

init -> register start -> [shoot and enter # of border collisions, user clicks enter] -> calculate distance (finish)

place_ball(inp)

Checks if the ball is placed on the starting point. Returns True if it is correct, False if not.

2.1.8.6 dummy

Classes

<i>Dummy()</i>	This gamemode only serves as a dummy mode to test the system
----------------	--

2.1.8.6.1 Dummy

class `Game.gamemodes.dummy.Dummy`

Bases: *GameMode*

This gamemode only serves as a dummy mode to test the system

`__init__()`

Methods

<code>__init__()</code>	
<code>build_HTML()</code>	Based on self.TREE, build the input fields for this gamemode flow.
<code>check_starting_positions(inp[, ...])</code>	Checks if the ball(s) is/are placed on the starting point(s).
<code>entrance(inp)</code>	This is the central function of every gamemode (can be overloaded).
<code>get_history()</code>	Gets the history table from self.history_file.
<code>history([history, add, get_semester])</code>	Get the player/team rankings.
<code>reset([keep_settings, inplace])</code>	Reset the gamemode object but keep changes made to the settings.
<code>save_history(history)</code>	Writes a passed dataframe to self.history_file (Path object) as csv (tab-separated, without an index).
<code>save_json_history(history)</code>	Appends a passed dict to a list of dicts in self.json_history_file.
<code>show([inp])</code>	When (re)entering the gamemode, determine what gameimage to show.
<code>socket_event(json_data)</code>	Currently not in use.

`entrance(inp)`

This is the central function of every gamemode (can be overloaded). This method is called by `Game.gamemode_controller`, which also handles the return values.

In normal gamemodes (which do not overload this), the action is determined by the structure of `GameMode.TREE`

Parameters

`inp (dict)` – json data input from the client. Usually posted to the endpoint `/gamemodecontroller`.

Returns

the output that gets send to the client as a response (must contain the fields “signal”, which is usually the current/new self.state value). The returned `GameImage` object is the new image that should get shown as a response to the request. The final string is the name of a sound file

(just the file name, without path or ending) on the Beamer module that should get played (if it is None, nothing will get played).

Return type

dict, *GameImage*, (str | None)

2.1.8.7 final_competition

Classes

FinalCompetition()

This is an abstraction of the KP2 mode.

2.1.8.7.1 FinalCompetition

class Game.gamemodes.final_competition.**FinalCompetition**

Bases: *KP2*

This is an abstraction of the KP2 mode. It essentially is the same, only using a different number of challenges and allows for the input of previously collected bonus points. It is used in the final competition of the KP2/MDP2 project.

__init__()

Methods

__init__()	
build_HTML()	Based on self.TREE, build the input fields for this gamemode flow.
check_starting_positions(inp[, ...])	Checks if the ball(s) is/are placed on the starting point(s).
entrance(inp)	Main entrance point.
get_history()	Gets the history table from self.history_file.
get_score()	Determine the score based on the scores of the individual played gamemodes.
hand_in()	Calculates the total score, score breakdown and saves the history.
history([history, add, get_semester])	Get the player/team rankings.
index_args()	Generate a dictionary of keyword arguments that get supplied to a jinja html template of a gamemode with the same name (e.g. precision -> precision.html) in the template directory.
mystery_distance_4b([session])	Fantastic Four: Hit 4 borders in (at least) one distance shot, get 350p
mystery_distance_500([session_hist, ...])	And I would walk 500 miles...: Try to be the closest to 500cm in the distance shot (session wide).
mystery_longestbreak_3solid([session])	One's company, two's a crowd and three's a party: Sink three solids in at least one round of the longest break

continues on next page

Table 39 – continued from previous page

<code>mystery_longestbreak_break([session])</code>	Not catchin' a break: Sink at least one solid in every round of the longest break
<code>mystery_precision_2bull([session])</code>	(2): Hit the bullseye at least twice during the precision challenge (<15mm)
<code>mystery_precision_600([session])</code>	It all adds up: The accumulated distance from the bullseye over all precision shots must not be larger than 600mm, earn 350p
<code>reset([keep_settings, inplace])</code>	Reset the gamemode object but keep changes made to the settings.
<code>save_history(history)</code>	Writes a passed dataframe to <code>self.history_file</code> (Path object) as csv (tab-separated, without an index).
<code>save_json_history(history)</code>	Appends a passed dict to a list of dicts in <code>self.json_history_file</code> .
<code>settings(inp)</code>	Handle game configuration tasks
<code>show([inp])</code>	When (re)entering the gamemode, determine what gameimage to show.
<code>socket_event(json_data)</code>	Currently not in use.

===

2.1.8.8 kp2

Classes

<code>KP2([occurences, gm_name])</code>	This class builds upon lower level gamemodes to deliver the entire user experience for the KP2 events.
---	--

===

2.1.8.8.1 KP2

```
class Game.gamemodes.kp2.KP2(occurences={'break': 1, 'distance': 5, 'longest_break': 5, 'precision': 5},
                               gm_name='KP2')
```

Bases: [GameMode](#)

This class builds upon lower level gamemodes to deliver the entire user experience for the KP2 events. The teaching unit is also known as MPD2.

```
__init__(occurences={'break': 1, 'distance': 5, 'longest_break': 5, 'precision': 5}, gm_name='KP2')
```

Methods

<code>__init__([occurences, gm_name])</code>	
<code>build_HTML()</code>	Based on <code>self.TREE</code> , build the input fields for this gamemode flow.
<code>check_starting_positions(inp[, ...])</code>	Checks if the ball(s) is/are placed on the starting point(s).
<code>entrance(inp)</code>	Main entrance point.
<code>get_history()</code>	Gets the history table from <code>self.history_file</code> .

continues on next page

Table 41 – continued from previous page

<code>get_score()</code>	Determine the score based on the scores of the individual played gamemodes.
<code>hand_in()</code>	Calculates the total score, score breakdown and saves the history.
<code>history([history, add, get_semester])</code>	Get the player/team rankings.
<code>index_args()</code>	Generate a dictionary of keyword arguments that get supplied to a jinja html template of a gamemode with the same name (e.g. precision -> precision.html) in the template directory.
<code>mystery_distance_4b([session])</code>	Fantastic Four: Hit 4 borders in (at least) one distance shot, get 350p
<code>mystery_distance_500([session_hist, ...])</code>	And I would walk 500 miles...: Try to be the closest to 500cm in the distance shot (session wide).
<code>mystery_longestbreak_3solid([session])</code>	One's company, two's a crowd and three's a party: Sink three solids in at least one round of the longest break
<code>mystery_longestbreak_break([session])</code>	Not catchin' a break: Sink at least one solid in every round of the longest break
<code>mystery_precision_2bull([session])</code>	(2): Hit the bullseye at least twice during the precision challenge (<15mm)
<code>mystery_precision_600([session])</code>	It all adds up: The accumulated distance from the bullseye over all precision shots must not be larger than 600mm, earn 350p
<code>reset([keep_settings, inplace])</code>	Reset the gamemode object but keep changes made to the settings.
<code>save_history(history)</code>	Writes a passed dataframe to self.history_file (Path object) as csv (tab-separated, without an index).
<code>save_json_history(history)</code>	Appends a passed dict to a list of dicts in self.json_history_file.
<code>settings(inp)</code>	Handle game configuration tasks
<code>show([inp])</code>	When (re)entering the gamemode, determine what gameimage to show.
<code>socket_event(json_data)</code>	Currently not in use.

entrance(inp)

Main entrance point. Mainly forwards to subgamemodes. inp must contain 'kp2_activity': 'precision' e.g.

get_score()

Determine the score based on the scores of the individual played gamemodes. Edit here to manipulate the scoring function (weights).

hand_in()

Calculates the total score, score breakdown and saves the history. Returns with signal="finished".

index_args()

Generate a dictionary of keyword arguments that get supplied to a jinja html template of a gamemode with the same name (e.g. precision -> precision.html) in the template directory

mystery_distance_4b(session=0, **kwargs)

Fantastic Four: Hit 4 borders in (at least) one distance shot, get 350p

mystery_distance_500(session_hist=0, history_addons=0, **kwargs)

And I would walk 500 miles...: Try to be the closest to 500cm in the distance shot (session wide). The closest team in the session gets 350p

THIS is implemented in the KP2.score, as it manipulates the history.

mystery_longestbreak_3solid(*session=0*, ***kwargs*)

One's company, two's a crowd and three's a party: Sink three solids in at least one round of the longest break

mystery_longestbreak_break(*session=0*, ***kwargs*)

Not catchin' a break: Sink at least one solid in every round of the longest break

mystery_precision_2bull(*session=0*, ***kwargs*)

(2): Hit the bullseye at least twice during the precision challenge (<15mm)

mystery_precision_600(*session=0*, ***kwargs*)

It all adds up: The accumulated distance from the bullseye over all precision shots must not be larger than 600mm, earn 350p

settings(*inp*)

Handle game configuration tasks

2.1.8.9 local_game

Classes

<i>LocalGame</i> (<i>env</i> , <i>settings</i>)	A local game that essentially simulates both sides of a online game
---	---

2.1.8.9.1 LocalGame

class Game.gamemodes.local_game.**LocalGame**(*env=None*, *settings=None*)

Bases: *GameMode*

A local game that essentially simulates both sides of a online game

__init__(*env=None*, *settings=None*)

Methods

<i>__init__</i> (<i>env</i> , <i>settings</i>)	
<i>build_HTML</i> ()	Based on self.TREE, build the input fields for this gamemode flow.
<i>check_starting_positions</i> (<i>inp</i> [, ...])	Checks if the ball(s) is/are placed on the starting point(s).
<i>entrance</i> (<i>inp</i>)	This is the central function of every gamemode (can be overloaded).
<i>get_history</i> ()	Gets the history table from self.history_file.
<i>history</i> (<i>[history, add, get_semester]</i>)	Get the player/team rankings.
<i>index_args</i> ()	
<i>play</i> (<i>inp</i>)	After the break is decided, enter the play loop

continues on next page

Table 43 – continued from previous page

<code>reset([keep_settings, inplace])</code>	Reset the gamemode object but keep changes made to the settings.
<code>save_history(history)</code>	Writes a passed dataframe to <code>self.history_file</code> (Path object) as csv (tab-separated, without an index).
<code>save_json_history(history)</code>	Appends a passed dict to a list of dicts in <code>self.json_history_file</code> .
<code>setup_shootout(inp)</code>	Upon sending the player and team names, save the game config and lookup the player
<code>shootout(inp)</code>	This gets called to two times (once per player): decides who will start
<code>show([inp])</code>	When (re)entering the gamemode, determine what gameimage to show.
<code>socket_event(json_data)</code>	Currently not in use.
<code>update_elo()</code>	Loads the history file, looks for the players (based on name/team) or adds them with <code>elo=1000</code> , and then updates them based on the outcome (assumes that the active player won.)

play(inp)

After the break is decided, enter the play loop

setup_shootout(inp)

Upon sending the player and team names, save the game config and lookup the player

shootout(inp)

This gets called to two times (once per player): decides who will start

update_elo()

Loads the history file, looks for the players (based on name/team) or adds them with `elo=1000`, and then updates them based on the outcome (assumes that the active player won.)

2.1.8.10 longest_break**Classes**

<code>LongestBreak([data_storage, can_discard, ...])</code>	The goal is to get the longest possible streak of sinking balls with each hit.
---	--

2.1.8.10.1 LongestBreak

class `Game.gamemodes.longest_break.LongestBreak`(*data_storage=None, can_discard=True, scored=2, tries=5, round_tracker=None, can_play_on=True*)

Bases: `GameMode`

The goal is to get the longest possible streak of sinking balls with each hit.

__init__(*data_storage=None, can_discard=True, scored=2, tries=5, round_tracker=None, can_play_on=True*)

Methods

<code>__init__([data_storage, can_discard, ...])</code>	
<code>build_HTML()</code>	Based on self.TREE, build the input fields for this gamemode flow.
<code>check_starting_positions(inp[, ...])</code>	Checks if the ball(s) is/are placed on the starting point(s).
<code>decide_keep(inp)</code>	After the first hit, the players have to decide if they want to continue this round or reset and try again
<code>determine_hit(inp)</code>	Determines if a finished hit is valid: if a ball was sunk, stay in this state and add to score tracker, if no ball was sunk, progress to state finished
<code>entrance(inp)</code>	This is the central function of every gamemode (can be overloaded).
<code>get_history()</code>	Gets the history table from self.history_file.
<code>get_score()</code>	
<code>history([history, add, get_semester])</code>	Get the player/team rankings.
<code>reset([keep_settings, inplace])</code>	Reset the gamemode object but keep changes made to the settings.
<code>save_history(history)</code>	Writes a passed dataframe to self.history_file (Path object) as csv (tab-separated, without an index).
<code>save_json_history(history)</code>	Appends a passed dict to a list of dicts in self.json_history_file.
<code>setup_next_round()</code>	The longest break challenge has its own method of determining rounds: there are 5 tries, but only two can be not discarded.
<code>show([inp])</code>	When (re)entering the gamemode, determine what gameimage to show.
<code>socket_event(json_data)</code>	Currently not in use.
<code>start_game(inp)</code>	Sets up default values and the gameimage on the beamer, selects stored challenges
<code>start_geometry()</code>	Provides the basic starting image.

decide_keep(inp)

After the first hit, the players have to decide if they want to continue this round or reset and try again

determine_hit(inp)

Determines if a finished hit is valid: if a ball was sunk, stay in this state and add to score tracker, if no ball was sunk, progress to state finished

setup_next_round()

The longest break challenge has its own method of determining rounds: there are 5 tries, but only two can be not discarded. A discard is defined by `new_score=-1`. This method decides, if the next round of `longest_break` can be discarded or not even played

Returns

can be discarded bool: can be played

Return type

bool

start_game(inp)

Sets up default values and the gameimage on the beamer, selects stored challenges

start_geometry()

Provides the basic starting image. Also defines the regions according to distance defined here

2.1.8.11 online_game**Classes**

<i>OnlineGame</i> (env)	Play a game of billiards online against another table in the network!
-------------------------	---

2.1.8.11.1 OnlineGame

class Game.gamemodes.online_game.**OnlineGame**(env)

Bases: *GameMode*

Play a game of billiards online against another table in the network!

Parameters

env (*dict*) – loaded and parsed .env file with configuration and authentication for the global API interface

__init__(env)

Methods

<i>__init__</i> (env)	
build_HTML()	Based on self.TREE, build the input fields for this gamemode flow.
check_starting_positions(inp[, ...])	Checks if the ball(s) is/are placed on the starting point(s).
entrance(inp)	This is the central function of every gamemode (can be overloaded).
<i>evaluate_break</i> (inp)	Send the coordinates to the API to determine the smaller distance and who should start.
<i>evaluate_play</i> (inp)	Sends the coordinates to the game and handles the decision
get_history()	Gets the history table from self.history_file.
history([history, add, get_semester])	Get the player/team rankings.
<i>other_played</i> (inp)	If the other player played and control was transferred (mechanism is the same as On-lineGame.reply_received)
<i>reply_received</i> (inp)	If there was no open game found immediately, just wait.
<i>request_game</i> (inp)	Send a request to the API to open the game or (if selected) look for open games
reset([keep_settings, inplace])	Reset the gamemode object but keep changes made to the settings.

continues on next page

Table 47 – continued from previous page

<code>save_history(history)</code>	Writes a passed dataframe to <code>self.history_file</code> (Path object) as csv (tab-separated, without an index).
<code>save_json_history(history)</code>	Appends a passed dict to a list of dicts in <code>self.json_history_file</code> .
<code>show([inp])</code>	When (re)entering the gamemode, determine what gameimage to show.
<code>socket_event(json_data)</code>	Currently not in use.

`evaluate_break(inp)`

Send the coordinates to the API to determine the smaller distance and who should start.

`evaluate_play(inp)`

Sends the coordinates to the game and handles the decision

`other_played(inp)`

If the other player played and control was transfered (mechanism is the same as `OnlineGame.reply_received`)

`reply_received(inp)`

If there was no open game found immediately, just wait. When a game is found or interrupted, the global server will send an (empty) socket event to the client (connected on load of the page), which will call this endpoint to then connect to the global server with authentication to get the update.

`request_game(inp)`

Send a request to the API to open the game or (if selected) look for open games

2.1.8.12 precision

Classes

<code>Precision([data_storage, bullseye, settings])</code>	The goal is to get as close as possible to the projected bullseye.
--	--

2.1.8.12.1 Precision

```
class Game.gamemodes.precision.Precision(data_storage="", bullseye=[557, 557], settings={'difficulty': 0})
```

Bases: `GameMode`

The goal is to get as close as possible to the projected bullseye.

```
__init__(data_storage="", bullseye=[557, 557], settings={'difficulty': 0})
```

Methods

<code>__init__([data_storage, bullseye, settings])</code>	continues on next page
---	------------------------

Table 49 – continued from previous page

<code>build_HTML()</code>	Based on self.TREE, build the input fields for this gamemode flow.
<code>check_starting_positions(inp[, ...])</code>	Checks if the ball(s) is/are placed on the starting point(s).
<code>determine_precision(inp)</code>	Determines if a finished hit is valid: if a ball was sunk, stay in this state and add to score tracker, if no ball was sunk, progress to stafte finished
<code>entrance(inp)</code>	This is the central function of every gamemode (can be overloaded).
<code>get_history()</code>	Gets the history table from self.history_file.
<code>history([history, add, get_semester])</code>	Get the player/team rankings.
<code>reset([keep_settings, inplace])</code>	Reset the gamemode object but keep changes made to the settings.
<code>save_history(history)</code>	Writes a passed dataframe to self.history_file (Path object) as csv (tab-separated, without an index).
<code>save_json_history(history)</code>	Appends a passed dict to a list of dicts in self.json_history_file.
<code>settings(inp)</code>	Handle setting requests.
<code>show([inp])</code>	When (re)entering the gamemode, determine what gameimage to show.
<code>socket_event(json_data)</code>	Currently not in use.
<code>start_geometry([default_difficulty])</code>	Provides the basic starting image.

determine_precision(inp)

Determines if a finished hit is valid: if a ball was sunk, stay in this state and add to score tracker, if no ball was sunk, progress to stafte finished

settings(inp)

Handle setting requests. In this gamemode, this is only the difficulty (0 = hard, 1, 2=easy)

start_geometry(default_difficulty=1)

Provides the basic starting image. Also defines the regions according to distance defined here

2.1.8.13 single_break**Classes**

<code>Break()</code>	The goal is to sink as many balls as possible with a single hit
----------------------	---

2.1.8.13.1 Break**class Game.gamemodes.single_break.Break**

Bases: `GameMode`

The goal is to sink as many balls as possible with a single hit

`__init__()`

Methods

<code>__init__()</code>	
<code>build_HTML()</code>	Based on self.TREE, build the input fields for this gamemode flow.
<code>check_starting_positions(inp[, ...])</code>	Checks if the ball(s) is/are placed on the starting point(s).
<code>count(inp)</code>	
<code>entrance(inp)</code>	This is the central function of every gamemode (can be overloaded).
<code>entrance_old(inp)</code>	
<code>get_history()</code>	Gets the history table from self.history_file.
<code>history([history, add, get_semester])</code>	Get the player/team rankings.
<code>reset([keep_settings, inplace])</code>	Reset the gamemode object but keep changes made to the settings.
<code>save_history(history)</code>	Writes a passed dataframe to self.history_file (Path object) as csv (tab-separated, without an index).
<code>save_json_history(history)</code>	Appends a passed dict to a list of dicts in self.json_history_file.
<code>show([inp])</code>	When (re)entering the gamemode, determine what gameimage to show.
<code>socket_event(json_data)</code>	Currently not in use.

TO DO

Todo

- move common_utils to another module, since it can cause problems with circular imports.

(The [original entry](#) is located in /home/ma4096/Documents/TUHH_Cloud/BA/software/modules/billard-game-module/Game/gamemodes/__init__.py:docstring of Game.gamemodes, line 77.)

Todo

- Experiment with timings between turning off the beamer and taking the image (especially with playing a sound simultaneously on the beamer)

(The [original entry](#) is located in /home/ma4096/Documents/TUHH_Cloud/BA/software/modules/billard-game-module/Game/__init__.py:docstring of Game._camera_interface.forward_coords, line 5.)

Todo

- change endpoint to template like /ballimage/<number> instead of current system with args.

(The [original entry](#) is located in /home/ma4096/Documents/TUHH_Cloud/BA/software/modules/billard-game-module/Game/__init__.py:docstring of Game.Game.get_ball_image, line 5.)

Todo

- Move the GameImage system to its entirely own module/project as an abstraction layer on top of PIL itself.
- Add GIF/video features. Maybe this would need the GameImage to be generated on the Beamer module itself.
- Add GameImage slots to minimize compute needs

(The [original entry](#) is located in /home/ma4096/Documents/TUHH_Cloud/BA/software/modules/billard-game-module/Game/GameImage.py:docstring of Game.GameImage.GameImage, line 31.)

Todo

- change r to be the total radius not this weird ring-thickness.
- ideally also add a number of rings parameter

(The [original entry](#) is located in /home/ma4096/Documents/TUHH_Cloud/BA/software/modules/billard-game-module/Game/GameImage.py:docstring of Game.GameImage.GameImage.bullseye, line 10.)

Todo

- add sizing options

(The [original entry](#) is located in /home/ma4096/Documents/TUHH_Cloud/BA/software/modules/billard-game-module/Game/GameImage.py:docstring of Game.GameImage.GameImage.centralImage, line 8.)

Todo

- verify the position of the starting point and break triangle

(The [original entry](#) is located in /home/ma4096/Documents/TUHH_Cloud/BA/software/modules/billard-game-module/Game/GameImage.py:docstring of Game.GameImage.GameImage.drawBreak, line 8.)

Todo

- Change match-case statement to dict mapping type to methods

(The [original entry](#) is located in /home/ma4096/Documents/TUHH_Cloud/BA/software/modules/billard-game-module/Game/GameImage.py:docstring of Game.GameImage.GameImage.draw_from_dict, line 11.)

Todo

- Finalize emphasised formatting (color blind friendly). Maybe generalize text rendering?

(The [original entry](#) is located in /home/ma4096/Documents/TUHH_Cloud/BA/software/modules/billard-game-module/Game/GameImage.py:docstring of Game.GameImage.GameImage.instructionText, line 7.)

PYTHON MODULE INDEX

g

- Game, [5](#)
- Game.Elo, [8](#)
- Game.GameEngine, [9](#)
- Game.GameImage, [10](#)
- Game.gamemodes, [23](#)
- Game.gamemodes.api_utils, [32](#)
- Game.gamemodes.common_utils, [34](#)
- Game.gamemodes.distance, [36](#)
- Game.gamemodes.dummy, [38](#)
- Game.gamemodes.final_competition, [39](#)
- Game.gamemodes.GameMode, [25](#)
- Game.gamemodes.GameModel, [31](#)
- Game.gamemodes.kp2, [40](#)
- Game.gamemodes.local_game, [42](#)
- Game.gamemodes.longest_break, [43](#)
- Game.gamemodes.online_game, [45](#)
- Game.gamemodes.precision, [46](#)
- Game.gamemodes.single_break, [47](#)
- Game.GameRules, [20](#)
- Game.GlobalAPI, [21](#)
- Game.Player, [22](#)

Symbols

__init__() (Game.Elo.Elo method), 9
 __init__() (Game.Game method), 6
 __init__() (Game.GameEngine.GameEngine method), 9
 __init__() (Game.GameImage.BilliardBall method), 11
 __init__() (Game.GameImage.GameImage method), 13
 __init__() (Game.GameImage.Trickshot method), 18
 __init__() (Game.GameRules.GameRules method), 20
 __init__() (Game.GlobalAPI.GlobalAPI method), 21
 __init__() (Game.Player.Player method), 22
 __init__() (Game.gamemodes.GameMode.GameMode method), 27
 __init__() (Game.gamemodes.GameModel.GameModel method), 31
 __init__() (Game.gamemodes.api_utils.API method), 32
 __init__() (Game.gamemodes.api_utils.ApiAuth method), 33
 __init__() (Game.gamemodes.distance.Distance method), 37
 __init__() (Game.gamemodes.dummy.Dummy method), 38
 __init__() (Game.gamemodes.final_competition.FinalCompetition method), 39
 __init__() (Game.gamemodes.kp2.KP2 method), 40
 __init__() (Game.gamemodes.local_game.LocalGame method), 42
 __init__() (Game.gamemodes.longest_break.LongestBreak method), 43
 __init__() (Game.gamemodes.online_game.OnlineGame method), 45
 __init__() (Game.gamemodes.precision.Precision method), 46
 __init__() (Game.gamemodes.single_break.Break method), 47

A

add_player() (Game.gamemodes.api_utils.API method), 33

API (class in Game.gamemodes.api_utils), 32
 api (Game.Game attribute), 6
 ApiAuth (class in Game.gamemodes.api_utils), 33
 arrow() (Game.GameImage.GameImage method), 14
 arrow_bottom() (Game.GameImage.GameImage method), 14

B

ball_distance() (in module Game.gamemodes.common_utils), 34
 base_image (Game.Game attribute), 5
 beamer (Game.Game attribute), 6
 beamer_correct_coords() (Game.Game method), 7
 beamer_make_gameimage() (Game.Game method), 7
 beamer_off() (Game.Game method), 7
 beamer_push_image() (Game.Game method), 7
 BilliardBall (class in Game.GameImage), 10
 Break (class in Game.gamemodes.single_break), 47
 build_HTML() (Game.gamemodes.GameMode.GameMode method), 27
 bullseye() (Game.GameImage.GameImage method), 14

C

calculate_score() (Game.gamemodes.distance.Distance method), 37
 camera (Game.Game attribute), 6
 camera_save_image() (Game.Game method), 7
 centralImage() (Game.GameImage.GameImage method), 15
 change_player() (Game.gamemodes.GameModel.GameModel method), 31
 check_positions() (in module Game.gamemodes.common_utils), 35
 check_starting_positions() (Game.gamemodes.GameMode.GameMode method), 28
 classify_region() (in module Game.gamemodes.common_utils), 35
 colors (Game.GameImage.BilliardBall attribute), 11
 coord_to_vec() (in module Game.gamemodes.common_utils), 35

coords_report() (in module *Game.gamemodes.common_utils*), 35
 copy() (*Game.GameImage.GameImage* method), 15
 current_dir (*Game.Game* attribute), 5

D

decide_keep() (*Game.gamemodes.longest_break.LongestBreak* method), 44
 definition (*Game.GameImage.GameImage* attribute), 12
 determine_hit() (*Game.gamemodes.longest_break.LongestBreak* method), 44
 determine_precision() (*Game.gamemodes.precision.Precision* method), 47
 determine_start() (*Game.GameRules.GameRules* method), 20
 Distance (class in *Game.gamemodes.distance*), 37
 draw_from_dict() (*Game.GameImage.GameImage* method), 15
 drawBallConnections() (*Game.GameImage.GameImage* method), 15
 drawBreak() (*Game.GameImage.GameImage* method), 15
 drawCue() (*Game.GameImage.Trickshot* method), 19
 drawPolygons() (*Game.GameImage.Trickshot* method), 19
 Dummy (class in *Game.gamemodes.dummy*), 38

E

Elo (class in *Game.Elo*), 9
 endpoint() (*Game.GlobalAPI.GlobalAPI* method), 22
 enter_round() (*Game.GameRules.GameRules* method), 21
 entrance() (*Game.gamemodes.dummy.Dummy* method), 38
 entrance() (*Game.gamemodes.GameMode.GameMode* method), 29
 entrance() (*Game.gamemodes.kp2.KP2* method), 41
 entrance_old() (*Game.gamemodes.distance.Distance* method), 37
 evaluate_break() (*Game.gamemodes.api_utils.API* method), 33
 evaluate_break() (*Game.gamemodes.GameModel.GameModel* method), 31
 evaluate_break() (*Game.gamemodes.online_game.OnlineGame* method), 46
 evaluate_play() (*Game.gamemodes.api_utils.API* method), 33
 evaluate_play() (*Game.gamemodes.GameModel.GameModel* method), 31
 evaluate_play() (*Game.gamemodes.online_game.OnlineGame* method), 46

F

filterData() (*Game.GameRules.GameRules* method), 21
 FinalCompetition (class in *Game.gamemodes.final_competition*), 39
 forward_coords() (*Game.Game* method), 7

G

Game module, 5
 Game (class in *Game*), 5
 Game (class in *Game.gamemodes.GameModel*), 31
 Game.Elo module, 8
 Game.GameEngine module, 9
 Game.GameImage module, 10
 Game.gamemodes module, 23
 Game.gamemodes.api_utils module, 32
 Game.gamemodes.common_utils module, 34
 Game.gamemodes.distance module, 36
 Game.gamemodes.dummy module, 38
 Game.gamemodes.final_competition module, 39
 Game.gamemodes.GameMode module, 25
 Game.gamemodes.GameModel module, 31
 Game.gamemodes.kp2 module, 40
 Game.gamemodes.local_game module, 42
 Game.gamemodes.longest_break module, 43
 Game.gamemodes.online_game module, 45
 Game.gamemodes.precision module, 46
 Game.gamemodes.single_break module, 47
 Game.GameRules module, 20
 Game.GlobalAPI module, 21
 Game.Player module, 22
 GameEngine (class in *Game.GameEngine*), 9
 GameImage (class in *Game.GameImage*), 12

gameimage (*Game.Game* attribute), 6
 GameMode (*class in Game.gamemodes.GameMode*), 25
 gamemode_controller() (*Game.Game* method), 7
 gamemode_socket_handler() (*Game.Game* method), 7
 GAMEMODES (*Game.Game* attribute), 6
 GameRules (*class in Game.GameRules*), 20
 generate_shootout() (*Game.gamemodes.GameModel.Game* method), 32
 get() (*Game.GlobalAPI.GlobalAPI* method), 22
 get_all_tables() (*Game.gamemodes.api_utils.API* method), 33
 get_ball_image() (*Game.Game* method), 7
 get_border_intercept() (*in module Game.gamemodes.common_utils*), 35
 get_display_string() (*Game.GameRules.GameRules* method), 21
 get_distance() (*Game.GameRules.GameRules* method), 21
 get_gamemode_website() (*Game.Game* method), 8
 get_history() (*Game.gamemodes.GameMode.GameMode* method), 29
 get_players_from_team() (*Game.gamemodes.api_utils.API* method), 33
 get_remaining_balls() (*Game.GameRules.GameRules* method), 21
 get_score() (*Game.gamemodes.kp2.KP2* method), 41
 getColor() (*Game.GameImage.BilliardBall* class method), 11
 getGroup() (*Game.GameImage.BilliardBall* class method), 11
 getHitHints() (*Game.GameImage.Trickshot* method), 19
 getImageCV2() (*Game.GameImage.GameImage* method), 16
 getImg() (*Game.GameImage.BilliardBall* method), 11
 getShots() (*Game.GameEngine.GameEngine* method), 10
 getTrickshotImage() (*Game.GameImage.Trickshot* method), 19
 GlobalAPI (*class in Game.GlobalAPI*), 21

H

hand_in() (*Game.gamemodes.kp2.KP2* method), 41
 handle_win() (*Game.GameRules.GameRules* method), 21
 history() (*Game.gamemodes.GameMode.GameMode* method), 29

I

ie() (*in module Game.GameImage*), 10
 index() (*Game.Game* method), 8
 index_args() (*Game.gamemodes.kp2.KP2* method), 41
 init_start() (*Game.GameRules.GameRules* method), 21
 instructionText() (*Game.GameImage.GameImage* method), 16
 is_close_enough() (*in module Game.gamemodes.common_utils*), 35

J

json() (*Game.gamemodes.api_utils.ApiAuth* method), 34

K

KP2 (*class in Game.gamemodes.kp2*), 40

L

line() (*Game.GameImage.GameImage* method), 16
 list_available_gamemodes() (*Game.Game* method), 8
 load_challenge_files() (*in module Game.gamemodes.common_utils*), 35
 load_coordinate_file() (*in module Game.gamemodes.common_utils*), 35
 LocalGame (*class in Game.gamemodes.local_game*), 42
 LongestBreak (*class in Game.gamemodes.longest_break*), 43

M

make_id() (*Game.Player.Player* method), 22
 match() (*Game.Elo.Elo* method), 9
 metric_distance() (*in module Game.gamemodes.common_utils*), 36
 metric_distance_closest() (*in module Game.gamemodes.common_utils*), 36
 module
 Game, 5
 Game.Elo, 8
 Game.GameEngine, 9
 Game.GameImage, 10
 Game.gamemodes, 23
 Game.gamemodes.api_utils, 32
 Game.gamemodes.common_utils, 34
 Game.gamemodes.distance, 36
 Game.gamemodes.dummy, 38
 Game.gamemodes.final_competition, 39
 Game.gamemodes.GameMode, 25
 Game.gamemodes.GameModel, 31
 Game.gamemodes.kp2, 40
 Game.gamemodes.local_game, 42
 Game.gamemodes.longest_break, 43

Game.gamemodes.online_game, 45
Game.gamemodes.precision, 46
Game.gamemodes.single_break, 47
Game.GameRules, 20
Game.GlobalAPI, 21
Game.Player, 22
mystery_distance_4b() (Game.gamemodes.kp2.KP2 method), 41
mystery_distance_500() (Game.gamemodes.kp2.KP2 method), 41
mystery_longestbreak_3solid() (Game.gamemodes.kp2.KP2 method), 42
mystery_longestbreak_break() (Game.gamemodes.kp2.KP2 method), 42
mystery_precision_2bull() (Game.gamemodes.kp2.KP2 method), 42
mystery_precision_600() (Game.gamemodes.kp2.KP2 method), 42

N

nameTeamText() (Game.GameImage.GameImage method), 17

O

OnlineGame (class in Game.gamemodes.online_game), 45
other_played() (Game.gamemodes.online_game.OnlineGame method), 46

P

place_ball() (Game.gamemodes.distance.Distance method), 37
placeAllBalls() (Game.GameImage.GameImage method), 17
placeBall() (Game.GameImage.GameImage method), 17
placeBalls() (Game.GameImage.Trickshot method), 19
placeHitHints() (Game.GameImage.Trickshot method), 19
play() (Game.gamemodes.local_game.LocalGame method), 43
Player (class in Game.Player), 22
Precision (class in Game.gamemodes.precision), 46
project_line() (in module Game.gamemodes.common_utils), 36
project_on_segment() (in module Game.gamemodes.common_utils), 36
propability() (Game.Elo.Elo method), 9

R

rectangle() (Game.GameImage.GameImage method), 17

redraw() (Game.GameImage.GameImage method), 17
render_template_camera() (Game.Game method), 8
reply_received() (Game.gamemodes.online_game.OnlineGame method), 46
request_game() (Game.gamemodes.online_game.OnlineGame method), 46
reset() (Game.gamemodes.GameMode.GameMode method), 30
rm_definition() (Game.GameImage.GameImage method), 17

S

save_history() (Game.gamemodes.GameMode.GameMode method), 30
save_json_history() (Game.gamemodes.GameMode.GameMode method), 30
save_players() (Game.Game method), 8
settings() (Game.gamemodes.kp2.KP2 method), 42
settings() (Game.gamemodes.precision.Precision method), 47
setup_next_round() (Game.gamemodes.longest_break.LongestBreak method), 44
setup_shootout() (Game.gamemodes.local_game.LocalGame method), 43
shootout() (Game.gamemodes.local_game.LocalGame method), 43
show() (Game.gamemodes.GameMode.GameMode method), 30
socket_event() (Game.gamemodes.GameMode.GameMode method), 30
start_game() (Game.gamemodes.api_utils.API method), 33
start_game() (Game.gamemodes.longest_break.LongestBreak method), 44
start_geometry() (Game.gamemodes.longest_break.LongestBreak method), 44
start_geometry() (Game.gamemodes.precision.Precision method), 47
state (Game.gamemodes.GameMode.GameMode attribute), 25
static (Game.GameImage.GameImage attribute), 18
storage (Game.Game attribute), 5

T

take_image() (Game.Game method), 8
TREE (Game.gamemodes.GameMode.GameMode attribute), 25
Trickshot (class in Game.GameImage), 18

U

update_definition() (Game.GameImage.GameImage method), 18

`update_elo()` (*Game.gamemodes.local_game.LocalGame*
 method), [43](#)
`update_text()` (*Game.GameImage.GameImage*
 method), [18](#)

V

`vec_to_coord()` (*in* *module*
 Game.gamemodes.common_utils), [36](#)
`view_csv()` (*Game.Game* *method*), [8](#)