game module - Billard@ISEM

Release 0.1

Mathis Wolter

Feb 15, 2026

CONTENTS:

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.

Have a look at the project website: https://billard.isem-tuhh.de/ (German)

ISEM - Institute for Smart Engineering and Machine Elements at the Hamburg University of Technology.

CHAPTER

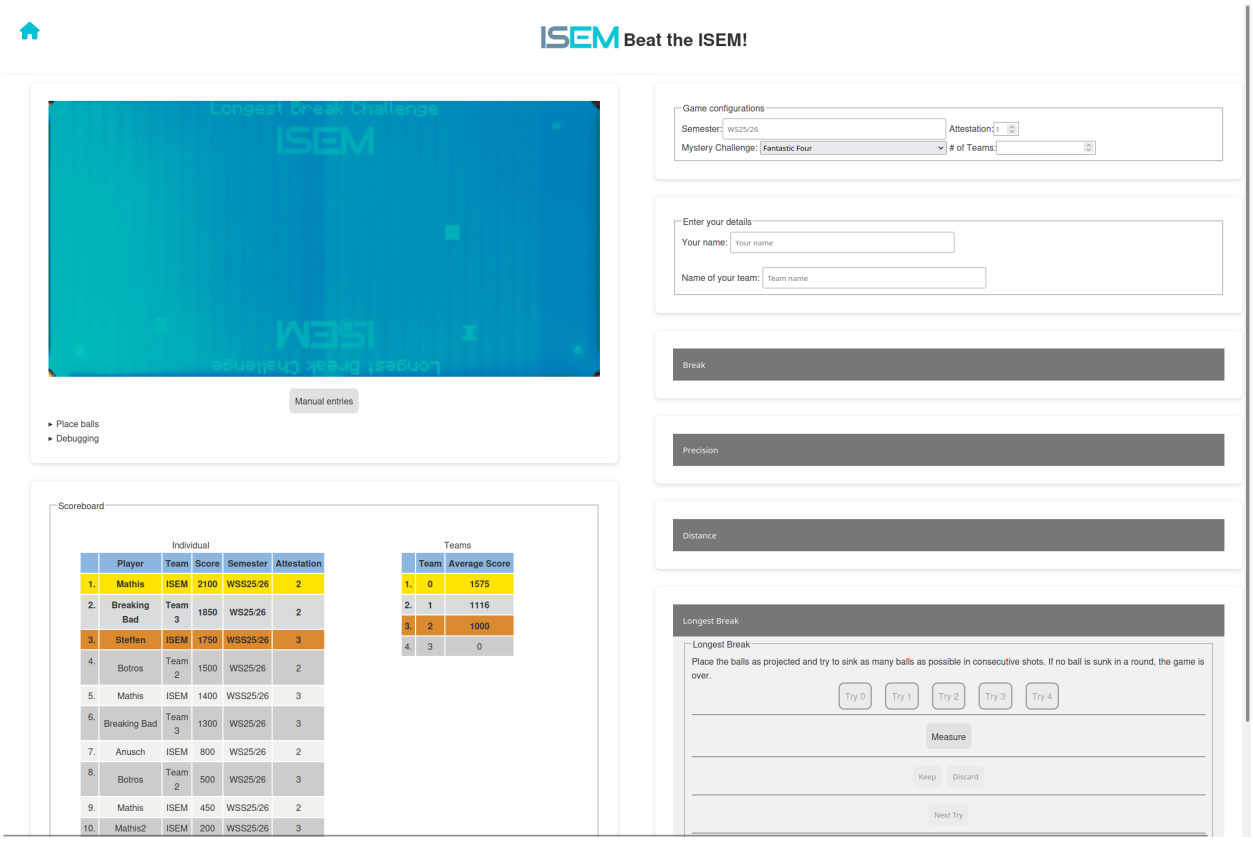
ONE

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 succesfully 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.



The website of the KP2 gamemode with the Longest Break challenge selected

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 prebuilt 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.

CHAPTER

TWO

USER MANUAL

As this system is mainly used in course contexts, users include students and lecturers. This manual is only concerned with interactions with websites of the Game Module. Other modules should usually not be visited by users, as debugging features are included on this page.

2.1 Index page

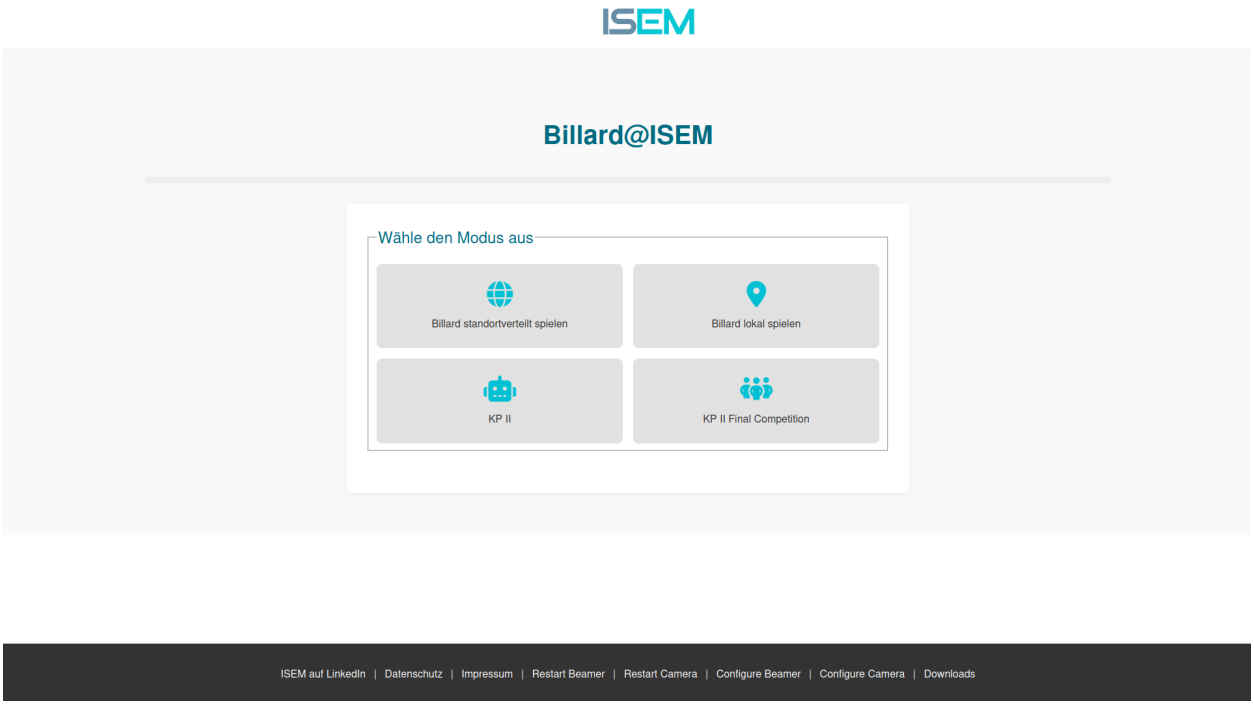


Fig. 1: The index page with four selectable games and a list of links in the footer.

After opening the website, the user is prompted with a selection of gamemodes. Usually, the user can directly select the wanted gamemode. For debugging purposes, the other modules (Camera and Beamer) are accessible in the footer, where they can be restarted or have their website opened. This will open in a new tab, when restarting the tab will not load (obviously, this indicates a successfull restart).

2.2 Game page

Opening or reloading a game site always resets the game progress. Be aware of that feature, especially in longer challenges like the KP2. Debugging like reloading the camera can be done without reloading the website!

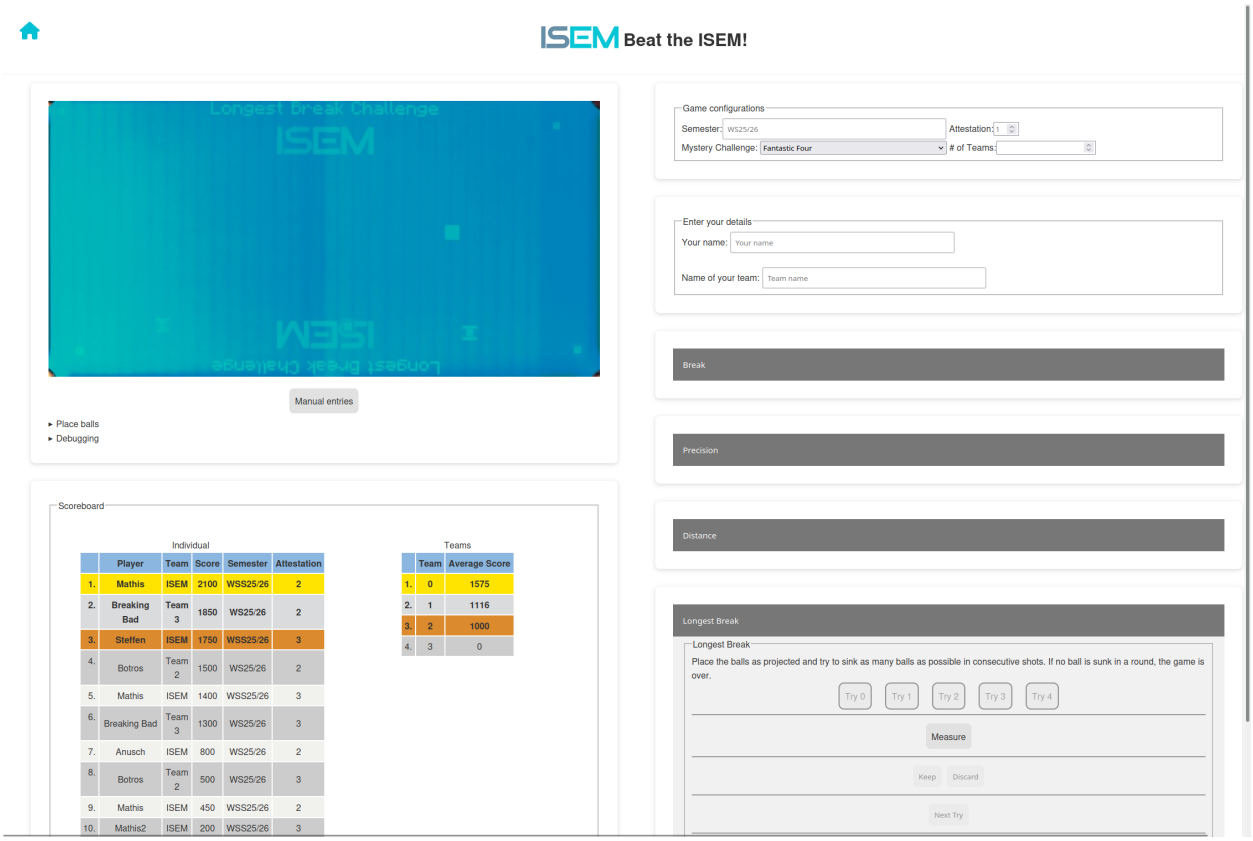


Fig. 2: An examplary games website (KP2) with the livestream (upper left), leaderboards (bottom left), options (upper right) and sub-gamemodes context menus (bottom right). The mode “Longest Break” is selected. The context menus “Place balls” and “Debugging” below the livestream are not open and the “Manual entries” option is not active.

The website of a game always follows the same structure. On the left half, the livestream with companioning menus, and the leaderboards are displayed. On the right, gamemode specific menus are shown. In this case, the KP2 mode has options and games (challenges), which are partially collapsed. By clicking on the challenge names, *their menu* opens.

Configurations can be made by entering data into the fields. All fields are mandatory, clicking onto anything else on the page applies the configuration. In the KP2 mode, this can be seen by the leaderboards only displaying results from the same semester as entered in the semester field.

The index page can be opened by clicking on the house icon in the upper left.

2.2.1 Challenge menu

A challenge’s menu contains all relevant steps to complete the challenge. The active step is enabled, with all others being greyed-out. By opening the challenge menu, the beamer displays the challenge on the billard table.

In this examplary menu, the number of collisions must also be entered. This is needed as the system can’t reliably track collisions with borders, especially at high speeds.

Clicking on a button (unless its a button for a decision, like “Next Try”) loads the coordinates of all balls on the table and displays them on the livestream window and on the table. As this system uses an artificial intelligence model, it

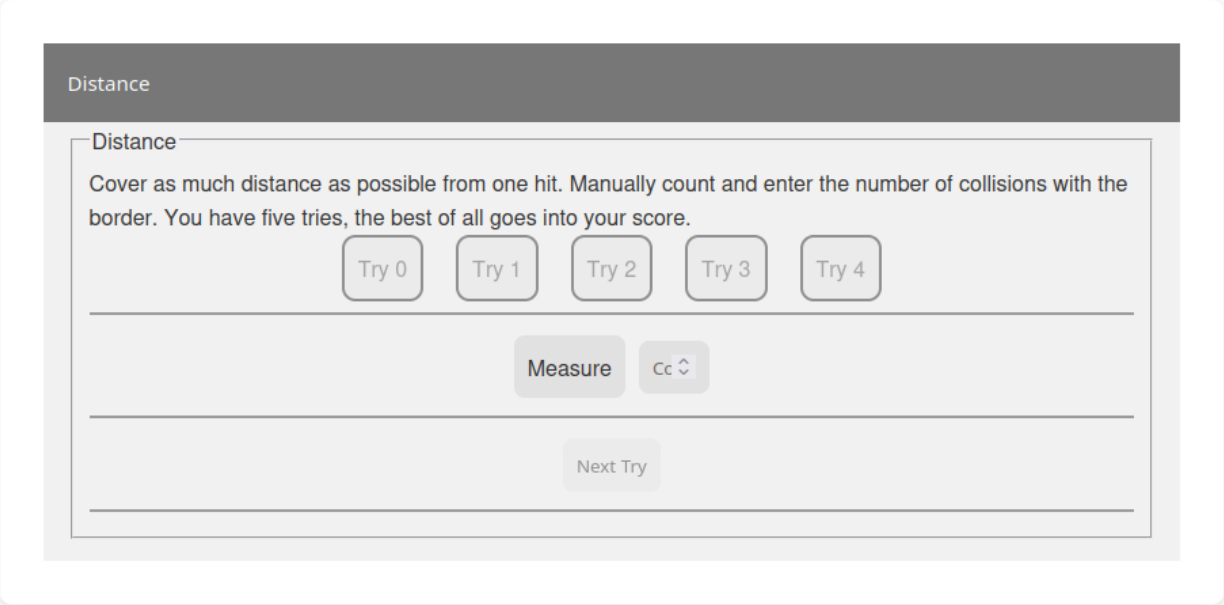


Fig. 3: The distance challenge context menu. A description of the challenge is given with instructions for entering data. As the user has five tries for the distance challenge in the KP2, there are five boxes where results will be entered into.

can make errors. If the user sees errors, they can correct them like shown *below*. Click on the button again to finally use the ball positions for the challenge the evaluate. This can not be reversed, so make sure that there are no mistakes in the ball positions.

2.2.2 Final submission and reset

In special gamemodes like KP2, containing challenges, the results need to be manually submitted using the "submit" button at the bottom. This is also possible when not all challenges have been done. In the case of KP2, a new entry appears in the scoreboard (highlighted in red) and a summary is shown just below the "submit" button.

Gamemodes can be reset after they are finished, but the safest way is to just reload the page.

2.2.3 Correcting wrong positions

If there are mistakes in the ball position, they can be corrected by opening the "Place balls" menu. For a quick glance, all balls that are currently detected are marked with **bold** text. By clicking onto the names of the balls, they are selected. Now clicking on the livestream will place the ball at the click position. Click on a detected ball to delete it. Changes are mirrored onto the beamer module.

Manipulating the detected coordinates activates "Manual entries" (blue if active). As long as this is active, the website will not load new coordinates from the camera but always use the positions that are currently displayed.

It is **very important** to deactivate the "Manual entries" when the balls are moved and next step should be measured.

2.2.4 Accessing collected data

In the footer of the *index page* a link to the downloads page is provided (*/download*). This page is login protected, where the password was set during the setup process. On that page, the histories of all gamemodes can be downloaded. Some gamemodes like KP2 are also provided as Excel files.

For every challenge, the finishing time and results are saved. Additional information on the configuration, scoring and played games are included. It does not save personal information besides player name (which has only ever been used

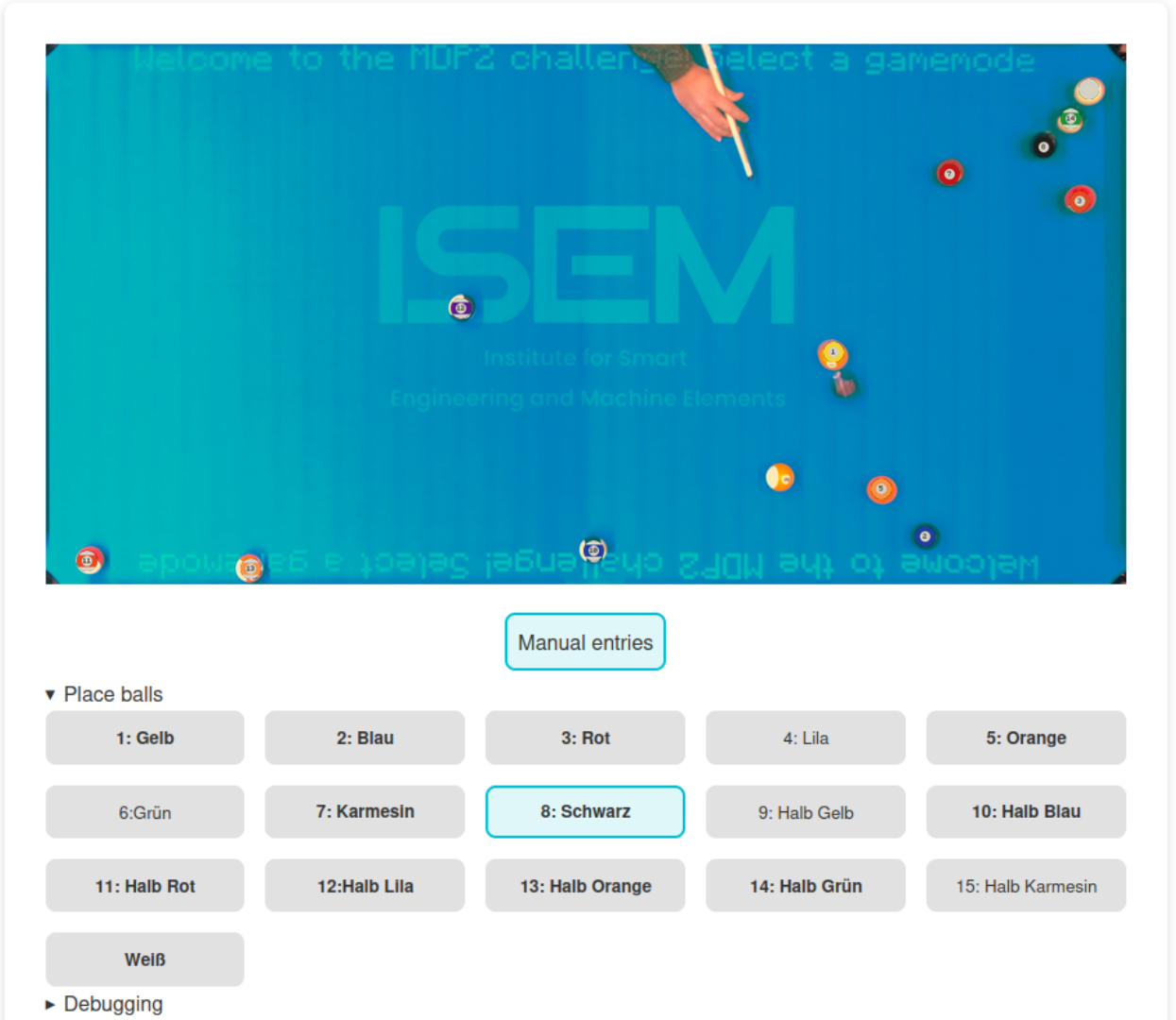


Fig. 4: The livestream with an expanded "Place balls" menu. Balls that are detected or placed can be seen on the livestream window and their names are **bold** in the menu. "Manual entries" are activated.

for “cool” team names), team name (mostly just a number) and temporal information.

CHAPTER

THREE

FRONTEND AND DATA EXCHANGE PROCESSES

As this documentation is built using Sphinx, which automatically only documents Python programs, the backend-frontend interaction needs to be manually documented.

This module implements a Model-View-Controller architecture for playing different gamemodes.

A gamemode is the model, consisting of different steps (states). It defines the process through its TREE attribute (see *GameMode documentation*). For every step/state, this dictionary defines - the function/method to call on submission, - the next state depending on the functions output, - the image to send to the beamer and - (often unused) the interface to get rendered on the frontend. A gamemode in this implementation can contain other gamemodes (see *KP2*), which should be treated in a way similar to the implementation in *KP2*.

The controller is implemented in the main *Game class* (*_gamemode_controller.py*) and is accessible via POST requests at */gamemodecontroller*. This is the main endpoint for interactions between the model and the view (client). It # receives the JSON data from the client, # passes it to the appropriate gamemode, # organizes the actions that should be taken from the return (new images, play sound), and # returns a serialized JSON object to the client.

3.1 Frontend scripts and functions

The view of the MVC is defined by the frontend loaded clientside. This includes the gamemode website (*/gamemode/[name]*) as well as a number of common JavaScript resources. In the following listing *.class* and *#id* represent CSS selectors.

From *base.html* (on which all gamemodes extend) the following resources are loaded:

- ***gamemode_controller.js*: Implements methods for communicating with the server.**
  # *sender(apiEndpoint, jsonData)*: POST the json data to the endpoint. Does not return. # *getCoordsAndConfirm(event, jsonData, fun=controller, final=...)*: This is called from some *click/ change* event listeners (mostly see *gamemode_retro.js*). If the *manipulatedFlag* is not raised, it requests new coordinates to be loaded from the Camera Module using *getCameraCoordinatesAsync*. This loads and displays the new coordinates. On a second invocation (tracked by setting *step_coordinate_process* property on the event target) or if *manipulatedFlag* is raised, it now passes the passed *jsonData* to the *fun* (a *controller function*). It runs the passed *final* function on the promise result (*.then* call) and returns that result as a resolved promise.
- ***coords-handler.js*: This is the main script for interacting with the livestream and manipulate coordinates.**
  # Adds a *click* event listener to *#reload-livestream* to reload the livestream (*#livestream* img) by cache busting. # *getCameraCoordinatesAsync*: if *manipulatedFlag* is not set, call */camera/coords* and place the returned coords using *placeAllBalls(coords)*. # *placeAllBalls(coords)*: based on coordinates in the backend format (name e.g *1* instead of *ball-1*), highlights the ball selectors (in *#ball-selector*) by adding *.ball-exists* class to all mentioned balls, removing it from all buttons previously. Also resets the *coordinates* JSON object and hides all placed ball images. Then places every mentioned ball using *placePointFromRealDim*, passing the coordinates and the id of the ball image to be displayed. # *placePointFromRealDim(xr, yr, id)*: rescales *x* and *y* coordinates to match the livestream size in pixels based on the size of the table. Calls *placePoint* to actually display the ball image. # *placePoint(x, y, id, realBall=true)*: places a ball image

(based on the passed id) on the livestream. If it is a *realBall*, the coordinates get added to *coordinates*, otherwise to *altCoordinates* (for non-ball objects, not really used at the moment). # *pxToReal(x)*: utility function to translate pixels to real length on the billard table. # Adds a *click* event listener to *.ball* to remove it from the *coordinates* JSON object, the livestream image and remove *.ball-exists* from its selector input (e.g. *#ball-selector input[value=ball-1]*). This raises the *manipulatedFlag*. # Adds a *click* event listener to the *#livestream* to place to *current_ball* on the livestream image. This raises the *manipulatedFlag*. # *sendCorrectedCoords*: POSTs the current coordinates in the backend format (e.g. *"1"* instead of *"ball-1"*) to */general/correctedcoords* using *coordinatesBackend* and *sender* function. # Adds a *click* event listener to *#manipulation-flag* checkbox to set *manipulatedFlag* according to its checked status. This can be used to lower the flag. # Adds *click* event listeners to every radio input in *#ball-selector*, setting the *current_ball* according to the selected radio. # *coordinatesBackend*: Returns *coordinates* transformed to the backend compatible format. # Adds debugging *click* event listeners to *#toggle-inference* (GET *camera_address* + */v1/togglequick*) and *#stop-livestream* (GET *camera_address* + */v1/stopgeneration*).

- *history_handler.js*: Provides *update_scoreboard* function, which builds a new HTML tables from past history data (see *GameMode history method*) and puts it into *#table-team* and *#table-single*. Also adds an event listener for *update_scoreboard*, passing the event *e.details* JSON to the *update_scoreboard* function.
- *update_gamemode.js*: Adds a *click* event listener to every *.collapsible* to open that object and close all others, setting *current_gamemode* to the *button.parentElement.id*. If the button is already expanded when clicked on, it collapses and *current_gamemode="base"*.
- ***gamemode_retro.js*: The main builder of event listeners to every gamemode on the page (*.mode*). For every mode, add:**
  # *change* event listener to *.setting* to send the inputs contained in the setting container to the server using *send_setting* # For every *.step* (a single state of the gamemode) add a *click* (button) or *change* (checkbox) event listener to the *input.submit-step* matches. This event calls a *controller function*, sending a CtS-JSON message to the server and executing the returned StC-JSON message. # This file also provides functions:
  - *getAllInputValues(container)*: returns a JSON of all input name - value pairs found inside the container
  - *activate_step(gamemode, new_step)*: adds class *deactivated* to all other steps of the gamemode.
  - *send_setting(setting_container, print=false, meta=false)*: gets all inputs from the container (*getAllInputValues*), adds basic CtS-JSON properties and sends to server using a *controller function*. If *meta=true*, it does not add a *kp2_activity="settings"* property.
- *game_configuration.js*: adds *focusout* event listener to every *.global-config* element to send all contained values (*getAllInputValues*) to the server using send_setting

3.2 Controller Functions

A controller function is the main interaction function between the client and the server. It adds organisational fields to the passed json data, sends it to the server (*/gamemodecontroller*), and takes actions based on the return, finally returning the json answer.

Each gamemodes website defines the controller function to use by loading it. The function gets set as the global *controller* variable to be included in other functions.

- ***vanillaController* (*vanillaController.js*): This is used by all gamemodes which are "monolithic", meaning not containing other gamemodes. It adds the field *gmode* set as *global_gamemode* to the passed json data. Actions can be taken on the following fields set in the returns:**
  # *notification* (*str*): shows a temporary (6s) alert in the upper left corner by calling *tempAlert* (defined in *base.html*). # *disable* (*list*): adds the *deactivated* class to all elements matching the CSS selectors passed in the list. # *enable* (*list*): like *disable*, but removing the *deactivated* class. # *emit* (*json*): emits events onto the *window*, defined as *{"[event-name]: {[optional data]}, ... }*. # *click* (*list*): emits a *click* event onto every element matching the CSS selectors in the list. # *log_to* (*str*): writes the *message* field value of the response to the first element that matches the CSS selector of this field. This removes the *message* field.

- ***kp2Controller(jsonData, set_activity=true)* (*kp2Controller.js*): This is used by all gamemodes that contain other gamemodes. If implemented with an *entrance* function "routing" like *KP2*, setting the field *kp2_activity* will pipe the data to the contained gamemode model.**
  - **This controller sets the fields**
    # *gmode* as *global_gamemode* (currently either *kp2* or *final_competition*) # *kp2_activity* as *current_gamemode*, but only if *set_activity* is *true*.
  - **This controller responds to the fields:**
    # *notification* (*str*): shows a temporary (6s) alert in the upper left corner by calling *tempAlert* (defined in *base.html*). # *disable* (*list*): adds the *deactivated* class to all elements matching the CSS selectors passed in the list. # *enable* (*list*): like *disable*, but removing the *deactivated* class.
  - Additionally, it adds a *gamemode_updated* event to the *document*, triggering *kp2Controller({"action": "show"})* which leads to the Game Module to send the newly opened gamemode's image to the Beamer Module.

3.3 StC and CtS JSON messages

This part describes the reference for Server-to-Client and Client-to-Server JSON messages.

Mandatory fields are highlighted in green, common fields with already implemented handlers are orange.

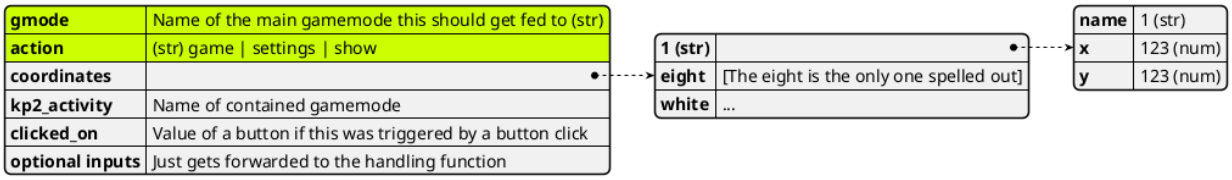


Fig. 1: This is an examplary message from the client to the server.

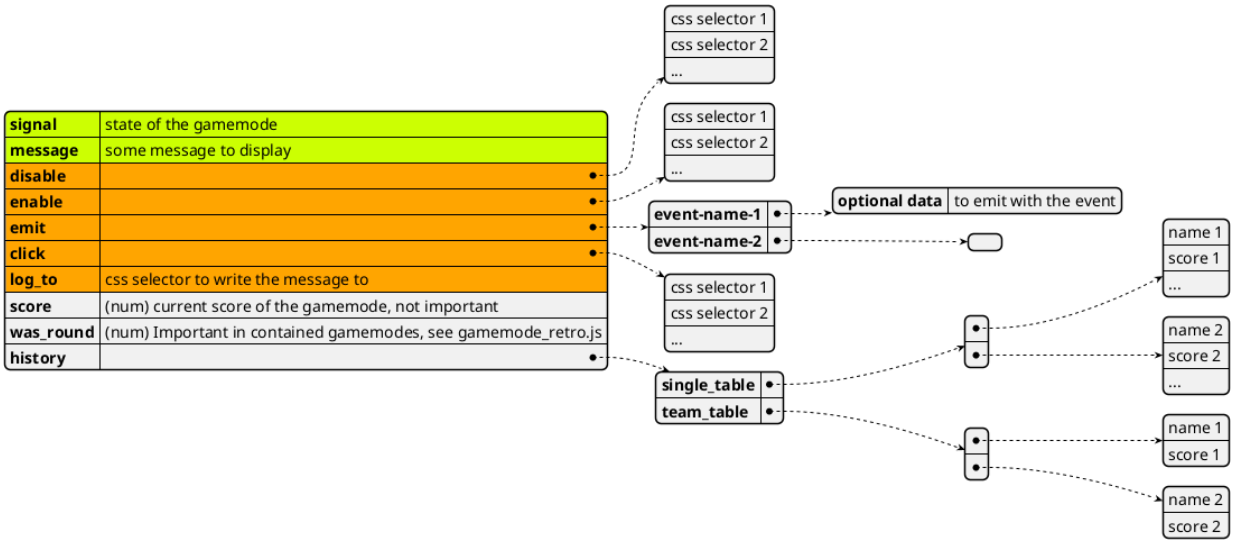


Fig. 2: This is an examplary message from the server to the client.

CHAPTER

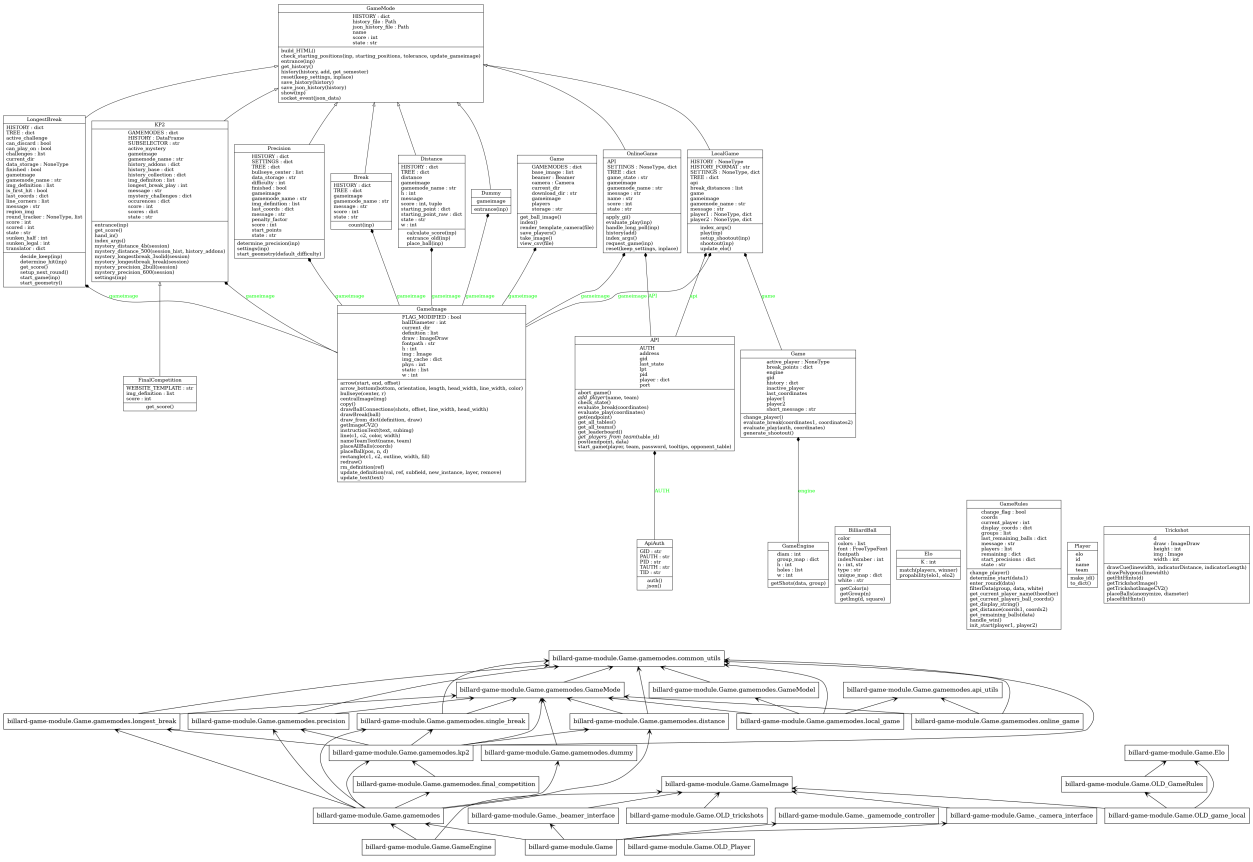
FOUR

API

4.1 UML Diagrams

Class diagram (inheritance) and dependence structure. The latter has partial errors, as every gamemode.[...] imports the GameImage class.

The diagrams are automatically generated using the Sphinx-Pyreverse plugin. If they are not readable in the PDF version of this documentation (too small), see the *docs/source/uml_images* directory for the full resolution images.



4.2 Game Module's API

Game	

4.2.1 Game

Game

Classes

Game([config, test_config, storage_folder, ...])	Implements central game scheduling functions

4.2.1.1 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 ( ). If the file does not exist or does not end in *.csv*, returns status 404 or 403

Modules

Elo	
GameEngine	
GameImage	
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).

4.2.1.2 Elo

Game.Elo

Classes

Elo([K])	Simple implementation of the elo rating system originally developed by Arpad Elo.

4.2.1.2.1 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

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

4.2.1.3 GameEngine

Game.GameEngine

Classes

GameEngine([size, ballDiameter])	This class implements basic physics based calculations for determining good hits.

4.2.1.3.1 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

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

Attributes

group_map

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"

4.2.1.4 GameImage

Game.GameImage

Functions

ie(dic, key, default)	If key in dic returns value in dic else default

4.2.1.4.1 Game.GameImage.ie

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

If key in dic returns value in dic else default

Classes

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

4.2.1.4.2 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/striped.

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

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

Attributes

colors
unique_map
white

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 infering 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

4.2.1.4.3 GameImage

class Game.GameImage.**GameImage**(*definition=[]*, *size=(2230, 1115)*, *phys=1*, *img_cache={'feedback-form-qr': 'Example QR Code', '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_chache** (*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 mimize compute needs

__init__(*definition=[]*, *size=(2230, 1115)*, *phys=1*, *img_cache={'feedback-form-qr': 'Example QR Code', 'isem-logo': 'static/images/ISEM-only.png', 'isem-logo-big': 'static/images/isem_logo_big.png'}*)

Methods

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

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

4.2.1.4.4 Trickshot

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

Bases: object

Deprecated.

The trickshot challenge this belongs to has been superseeded 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

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

4.2.1.5 gamemodes

Game.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␣
```

(continues on next page)

(continued from previous page)

```
↪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/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.
```

(continues on next page)

(continued from previous page)

```
            "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

continues on next page

Table 18 – continued from previous page

kp2
local_game
longest_break
online_game
precision
single_break

4.2.1.5.1 GameMode

Game.gamemodes.GameMode

Classes

GameMode()	The parent class to all gamemodes.

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␣
↪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.
            {
```

(continues on next page)

(continued from previous page)

```
                "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/
↪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]
        }]
```

(continues on next page)

(continued from previous page)

```
    ],
    # a "finished" action is automatically added by the parent class GameMode␣
↪if not already specified.
}
```

Type
dict

__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)	This is the central function of every gamemode (can be overloaded).
get_history()	Gets the history table from self.history_file.
history([history, add, get_semester])	Get the player/team rankings.
reset([keep_settings, inplace])	Reset the gamemode object but keep changes made to the settings.
save_history(history)	Writes a passed dataframe to self.history_file (Path object) as csv (tab-separated, without an index).
save_json_history(history)	Appends a passed dict to a list of dicts in self.json_history_file.
show([inp])	When (re)entering the gamemode, determine what gameimage to show.
socket_event(json_data)	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:
    "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].

4.2.1.5.2 GameModel

Game.gamemodes.GameModel

Classes

Game(player1, player2)	This is the general game simulation for a typical game of billiards

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

__init__(player1, player2)	
change_player()	Based on who is the current player (player1 or player2), this switches it.
evaluate_break(coordinates1, coordinates2)	Chooses the player that can break based on two precision shots.
evaluate_play(auth, coordinates)	Based on common billard rules, control what should happen next.
generate_shootout()	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 wether 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 arbtirary 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

4.2.1.5.3 api_utils

Game.gamemodes.api_utils

Classes

API(env)	Provides a common interface for communicating with the global API
ApiAuth(TID, TAUTH[, PID, PAUTH, GID])	Provides a common data interface for transfering authentication data

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)	
abort_game()	
add_player(name, team)	Add a player of a certain team.
check_state()	
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(endpoint)	Get json data from a certain authenticated endpoint
get_all_tables()	Load all table names/locations/ID as a list from the API
get_all_teams()	
get_leaderboard()	Get the leaderboard in a format matching GameMode.history's single_table

continues on next page

Table 24 – continued from previous page

get_players_from_team(table_id)	Load all players that are on a certain table
post(endpoint, data)	Post json data to a certain authenticated endpoint
start_game(player, team, password, tooltips, ...)	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(*endpoint*)

Get json data from a certain authenticated endpoint

Authenticated by the information stored in API.ApiAuth

Parameters
endpoint (*str*) – Endpoint of the api like "api/test" or "/api/test"

Raises
ConnectionError – If the endpoint is not available, raise this error

Returns
Returned json data including ("http": http status code) pair.

Return type
dict

get_all_tables()

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

get_leaderboard()

Get the leaderboard in a format matching GameMode.history's single_table

get_players_from_team(*table_id*)

Load all players that are on a certain table

post(*endpoint*, *data*)

Post json data to a certain authenticated endpoint

Authenticated by the information stored in API.ApiAuth

Parameters

- **endpoint** (*str*) – Endpoint of the api like "api/test" or "/api/test"
- **data** (*dict*) – Data to send to the server. Must be serializable.

Raises
ConnectionError – If the endpoint is not available, raise this error

Returns
Returned json data including ("http": http status code) pair.

Return type
dict

start_game(*player*, *team*, *password*, *tooltips*, *opponent_table*)

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

ApiAuth

class Game.gamemodes.api_utils.**ApiAuth**(*TID: str*, *TAUTH: str*, *PID: str = ''*, *PAUTH: str = ''*, *GID: str = ''*)

Bases: `object`

Provides a common data interface for transfering authentication data

__init__(*TID: str*, *TAUTH: str*, *PID: str = ''*, *PAUTH: str = ''*, *GID: str = ''*) → None

Methods

__init__(TID, TAUTH[, PID, PAUTH, GID])	
auth()	
json()	Return all set fields (= not default value) as a dict

Attributes

GID
PAUTH
PID
TID
TAUTH

json()

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

4.2.1.5.4 common_utils

Game.gamemodes.common_utils

Functions

ball_distance(coords, b1, b2)	From a coords dict, calculates the distance between two given balls
check_positions(real, goal[, tolerance])	Check if each ball of two sets of balls are close enough to each other (tolerance in mm).

continues on next page

Table 27 – continued from previous page

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.
coord_to_vec(coord)	Transforms a {"x": 0, "y": 0} dict to a np.ndarray [x, y]
coords_report(coordsNew, coordsOld)	Find the differences between new and old coordinates and report on sunken balls.
get_border_intercept(coordStart, coordPass)	Determines a point somewhere outside the playing field on a line drawn from a starting point through a passing point.
is_close_enough(c1, c2[, tolerance])	Are the two coordinates close enough too each other? tolerance is given in mm (as are coordinates)
load_challenge_files(glob_path[, ...])	looks for all files matching the glob pattern passed (see Python's glob module) and loads them with load_coordinate_file.
load_coordinate_file(path)	load a coordinate file (like longest break challenges) from a file.
metric_distance(c1, c2)	Metric distance between two coordinates each containing {"x": 0, "y": 0}
metric_distance_closest(coords, c2)	Calculates the distance between one point and all points in the coords.
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.
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.
vec_to_coord(vec)	Transforms a np.ndarry [x,y] into a dict {"x": 0, "y": 0}

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

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)

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)

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]

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.

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.

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)

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.

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.

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}

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.

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.

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.

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}

4.2.1.5.5 distance

Game.gamemodes.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

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*)

Fig. 1: The image projected when the Distance gamemode is active.

Methods

__init__([w, h])	
build_HTML()	Based on self.TREE, build the input fields for this gamemode flow.
calculate_score(inp)	Horizontal distance in mm is the score
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).
entrance_old(inp)	init -> register start -> [shoot and enter # of border collisions, user clicks enter] -> calculate distance (finish)
get_history()	Gets the history table from self.history_file.
history([history, add, get_semester])	Get the player/team rankings.
place_ball(inp)	Checks if the ball is placed on the starting point.
reset([keep_settings, inplace])	Reset the gamemode object but keep changes made to the settings.
save_history(history)	Writes a passed dataframe to self.history_file (Path object) as csv (tab-separated, without an index).
save_json_history(history)	Appends a passed dict to a list of dicts in self.json_history_file.
show([inp])	When (re)entering the gamemode, determine what gameimage to show.
socket_event(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.

4.2.1.5.6 dummy

Game.gamemodes.dummy

Classes

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

Dummy

class Game.gamemodes.dummy.**Dummy**

Bases: *GameMode*

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

__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)	This is the central function of every gamemode (can be overloaded).
get_history()	Gets the history table from self.history_file.
history([history, add, get_semester])	Get the player/team rankings.
reset([keep_settings, inplace])	Reset the gamemode object but keep changes made to the settings.
save_history(history)	Writes a passed dataframe to self.history_file (Path object) as csv (tab-separated, without an index).
save_json_history(history)	Appends a passed dict to a list of dicts in self.json_history_file.
show([inp])	When (re)entering the gamemode, determine what gameimage to show.
socket_event(json_data)	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)

4.2.1.5.7 final_competition

Game.gamemodes.final_competition

Classes

FinalCompetition()	This is an abstraction of the KP2 mode.

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 indiviual 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.

continues on next page

Table 33 – continued from previous page

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

get_score()

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

This is a clone of KP2.get_score, except that it does not account for Mystery Challenges or Passing

4.2.1.5.8 kp2

Game.gamemodes.kp2

Classes

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

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'*)

Fig. 2: The image projected when the KP2 gamemode is active, seen through the calibrated Camera Module.

Methods

__init__([occurences, gm_name])	
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 indi-viual 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
mystery_longestbreak_break([session])	Not catchin' a break: Sink at least one solid in every round of the longest break

continues on next page

Table 35 – continued from previous page

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

4.2.1.5.9 local_game

Game.gamemodes.local_game

Classes

LocalGame([env, settings])	A local game that essentially simulates both sides of a online game

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

__init__([env, settings])	
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).
get_history()	Gets the history table from self.history_file.
history([history, add, get_semester])	Get the player/team rankings.
index_args()	
play(inp)	After the break is decided, enter the play loop
reset([keep_settings, inplace])	Reset the gamemode object but keep changes made to the settings.
save_history(history)	Writes a passed dataframe to self.history_file (Path object) as csv (tab-separated, without an index).
save_json_history(history)	Appends a passed dict to a list of dicts in self.json_history_file.
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
show([inp])	When (re)entering the gamemode, determine what gameimage to show.
socket_event(json_data)	Currently not in use.
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.)

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.)

4.2.1.5.10 longest_break

Game.gamemodes.longest_break

Classes

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

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.

Fig. 3: The image projected when the Longest Break gamemode is active.

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

Methods

__init__([data_storage, can_discard, ...])	
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).
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
entrance(inp)	This is the central function of every gamemode (can be overloaded).
get_history()	Gets the history table from self.history_file.
get_score()	
history([history, add, get_semester])	Get the player/team rankings.
reset([keep_settings, inplace])	Reset the gamemode object but keep changes made to the settings.
save_history(history)	Writes a passed dataframe to self.history_file (Path object) as csv (tab-separated, without an index).
save_json_history(history)	Appends a passed dict to a list of dicts in self.json_history_file.
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.
show([inp])	When (re)entering the gamemode, determine what gameimage to show.
socket_event(json_data)	Currently not in use.
start_game(inp)	Sets up default values and the gameimage on the beamer, selects stored challenges
start_geometry()	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

4.2.1.5.11 online_game

Game.gamemodes.online_game

Classes

OnlineGame([env, settings])	Play a game of billiards online against another table in the network!

OnlineGame

class Game.gamemodes.online_game.**OnlineGame**(*env=None*, *settings=None*)

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=None*, *settings=None*)

Methods

__init__([env, settings])	
apply_gi()	
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).
evaluate_play(inp)	Sends the coordinates to the game and handles the decision
get_history()	Gets the history table from self.history_file.
handle_long_poll(inp)	This method gets called by run_long_poll from long_poll_online.js.
history([add])	Get the player/team rankings.
index_args()	

continues on next page

Table 41 – continued from previous page

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

Attributes

game_state	this tracks the state of the current game, kept up to date with the API

evaluate_play(*inp*)

Sends the coordinates to the game and handles the decision

game_state

this tracks the state of the current game, kept up to date with the API

handle_long_poll(*inp*)

This method gets called by run_long_poll from long_poll_online.js. Check if there is a new state available on the server. If that is the case and its this players turn, return to play mode. Alternatively, display the new gameimage and messages.

Parameters
inp (*_type_*) – _description_

history(*add=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

request_game(*inp*)

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

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)

4.2.1.5.12 precision

Game.gamemodes.precision

Classes

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

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}*)

Fig. 4: The image projected when the Precision gamemode is active.

Methods

__init__([data_storage, bullseye, settings])	
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).
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
entrance(inp)	This is the central function of every gamemode (can be overloaded).
get_history()	Gets the history table from self.history_file.
history([history, add, get_semester])	Get the player/team rankings.
reset([keep_settings, inplace])	Reset the gamemode object but keep changes made to the settings.
save_history(history)	Writes a passed dataframe to self.history_file (Path object) as csv (tab-separated, without an index).
save_json_history(history)	Appends a passed dict to a list of dicts in self.json_history_file.
settings(inp)	Handle setting requests.
show([inp])	When (re)entering the gamemode, determine what gameimage to show.
socket_event(json_data)	Currently not in use.
start_geometry([default_difficulty])	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

4.2.1.5.13 single_break

Game.gamemodes.single_break

Classes

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

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

__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).
count(inp)	
entrance(inp)	This is the central function of every gamemode (can be overloaded).
get_history()	Gets the history table from self.history_file.
history([history, add, get_semester])	Get the player/team rankings.
reset([keep_settings, inplace])	Reset the gamemode object but keep changes made to the settings.
save_history(history)	Writes a passed dataframe to self.history_file (Path object) as csv (tab-separated, without an index).
save_json_history(history)	Appends a passed dict to a list of dicts in self.json_history_file.
show([inp])	When (re)entering the gamemode, determine what gameimage to show.
socket_event(json_data)	Currently not in use.

CHAPTER

FIVE

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 mimize 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
- this may get detected as a ball. Maybe change innermost circle to a square?

(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

INDEX

Symbols

A

B

C

H

I

J

K

L

M

N

O

P

R

S

T

U

V