# A Simple JavaScript Bomberman Clone in PIXI.js

Author: Milan Vlachovský

## Contents

## 1. Introduction

The goal of this project was to create a simple Bomberman clone using web technologies. The project focuses on demonstrating the basic principles of how the game works while ensuring comfortable controls and a good user experience. The project was conceived as a school semester assignment, so certain aspects that would be important in real production are simplified or omitted here (a proper WSGI server, ...).

## 2. Game Description

This is a simplified version of Bomberman in which the player controls a character on a rectangular playing field. The player's goal is to destroy all enemies and obstacles on the field using bombs that can be placed. The player has 1 bomb available to place on the field at any given moment. A bomb has a timer and, after a set time, explodes and destroys all entities in its vicinity. However, the blast cannot spread through indestructible walls nearby. The player has 0 to 3 spare lives, and if hit by an enemy or an explosion, they lose one life. Two game modes are available in this work:

- *Normal* mode
  - The game ends if the player loses all lives or completes all levels (a level can end once all enemy entities and all destructible walls are destroyed).
- *Endless* mode
  - The game ends when the player loses all lives. Until then, the player tries to reach the highest possible level, score as many points as possible, and do it in the shortest time.

The game is controlled via the keyboard. The player moves with the arrow keys and places bombs using the spacebar. The game is displayed in the browser window and can be played on any device with a web browser and a keyboard. The game includes sound effects. It also features a leaderboard of the best players stored on the server, and you can enter it after the game ends. The choice of the number of lives can be understood as the game's difficulty. All textures and sounds used come from freely available sources and are used non-commercially.

## 3. System Architecture

The project is divided into a frontend part, which includes client-side code written in JavaScript using the PIXI.js library and code written in React serving as a landing page; and a backend part, which consists of a web server (Nginx), the document database MongoDB, and a Python script using the Flask library for asynchronous (AJAX) processing of POST and GET requests to communicate with the DB. When connecting to the server, static files (HTML, CSS, JS) are sent to the client, and the entire game runs on the client side. The backend is used only for storing scores and retrieving the leaderboard. The static files sent to the client are preprocessed by the Parcel bundler.

### Requirements

- Node.js/npm
- Python 3.10 or higher
- MongoDB
- Nginx

## 4. Setup Guide

To build the static files, just run the script:

- Linux/MacOS:

  ```
  npm run build
  ```

- Windows (PowerShell):

  ```
  npm run build-win
  ```

Using Parcel, this creates a *dist/* folder with the resulting files (or *npm run build-dev*, which creates files with source maps for debugging). These files need to be served by a web server (e.g., the aforementioned Nginx). To fetch and store information in the DB via AJAX calls, you need to run a Flask server (a primitive, native Python server; not suitable for real production because it is not a full WSGI server, but sufficient for the purposes of this project). For proper functionality, a MongoDB server must be running with default port settings. To start the Flask server, install the dependencies from `requirements.txt`, run `db_init.py` to initialize the DB, and then `ajax_handler.py` to start the Flask server. This script will listen on port 5000 and process requests from the frontend.

For local testing, you can use the built-in `http-server` via the scripts:

- Linux/MacOS:

  ```
  npm run start
  ```

- Windows (PowerShell):

```
npm run start-win
```

### Step-by-step installation

Follow these steps to install and set up the project:

1. Keep the project in a local directory with read, write, and execute permissions.
2. Install *js* dependencies with

   ```
   npm install
   ```

   from the project root, and Python dependencies with

   ```
   pip install -r requirements.txt
   ```

   in the *backend/* folder.
3. Build the static files with:

   ```
   npm run build
   ```

   Or:

   ```
   npm run build-dev
   ```

   The files will be saved in the *dist/* folder.
4. Start the MongoDB server on the default port.
5. Start the Flask server with `python ajax_handler.py` in the *backend/* folder.
6. Start the web server (e.g., Nginx) and set the path to the static files to the *dist/* folder (the main files are `index.html` and `game.html`). Make sure the server also uses a reverse proxy to forward requests to the Flask server (port 5000).
7. Open a web browser and enter the server address.

## 5. Project Structure

### Root directory

- *audio/* — Sound files for effects and music.

- *backend/* — Flask app scripts and other backend files.
- *css/* — Frontend styles.
- *dist/* — Build output.
- *fonts/* — Fonts used in the game.
- *img/* — Images used in the game.
- *img_template/* — Temporary images used during development (not part of the final build).
- *js/* — JavaScript files for the frontend.
  - *js/constants/* — Constants used in the game.
  - *js/graphic_elems/* — Graphic elements used in the game.
- *node_modules/* — NPM dependencies.

- *package.json* — NPM configuration.
- *scripts/* — Helper scripts for building and running.
- *index.html* — Entry point for the game.

## Backend (folder *backend/*)

- `db.json`: JSON file to initialize the database so the game runs correctly on first launch.
- `db_init.py`: Python script for database initialization.
- `ajax_handler.py`: Python script that processes AJAX requests using Flask.
- `requirements.txt`: Python dependencies.

## Frontend (folder *js/*)

> Files are ordered by their execution priority. Only the main files are shown, from which the structure and functioning of the game are evident. A more detailed description of individual methods and classes is provided in the code documentation or in **jsdoc.html**. All methods are documented using JSDoc.

- **app.js**: Entry point for the game.
  - The game is initialized here.
  - An instance of the PIXI application is created.
  - Methods for scaling and adapting to window size and keyboard controls are connected.
  - All required assets are loaded (images, sounds, ...).
  - A game instance is created, represented by the `Game` class from the `game.js` module.
  - A ticker provided by PIXI.js is attached to the `update(delta)` method from **game.js**, ensuring regular calls to `update(delta)` with the current time difference between rendered frames.
    - The `update(delta)` method updates the game state and serves as the main game loop.

- **loader.js**: Module that handles asset loading.

- **sound_manager.js**: Module that manages sounds in the game.
  - Contains the `SoundManager` class, which holds information about sounds used in the game.
  - Each sound is played via a method named `playSoundName()`.
  - An instance of `SoundManager` is used by the `Game` and `GameSession` classes to play sounds in the game.

- **game.js**: Module containing the `Game` class, representing the game.
    - The `Game` class stores information about the current game state in the `gameState` variable (an instance of the `GameState` class defined in `game_state.js`).
    - The main method is `update(delta)`, which updates the game state.
    - The `update(delta)` method is called from `app.js` and ensures regular calls to `update(delta)` with the current time difference between rendered frames.
    - It works on the principle of a state machine, where different updates are performed depending on the game state (a switch statement is used for branching).
    - Depending on the game state, methods are called to update individual parts of the game:
        - `handleMainMenuUpdate()`: Updates the main menu based on pressed keys and selected options.
        - `handleGameSessionUpdate(delta)`: Initializes/updates the ongoing game session.
        - `handleSettingsUpdate()`: Updates the settings based on pressed keys and selected options.
        - `handleLeaderboardUpdate()`: Updates the leaderboard view.
        - `handleGameEndScreen()`: Updates the post-game screen.
    - All these methods work with two main objects:
        - `screenContent`: Contains all logical objects to be rendered to the screen (in the case of menus, the individual options; in the case of a game session, the arena, player, enemies, ...).
        - `drawingManager`: This is an instance of one of the modules `drawing_manager_menus.js` if rendering a menu, or `game_session.js` if rendering a game session.
            - For menus, methods for rendering the entire view are called directly based on user input updates. Redrawing occurs whenever the menu state changes or due to window resizing.
            - For a game session, a method is continuously called to update the session, which manages rendering of the arena, player, enemies, ... and ensures interaction among them, including collisions and user input processing. Redrawing happens at the level of game objects created with PIXI.js, not at the level of the entire view (for optimization and snappy rendering).
            - > The author's original idea was that instances from the drawing_manager module would be used only for rendering to the screen, but it turned out that for the game session (due to the complexity of collision detection without access to the game canvas, etc.) it was more reasonable to let the session itself manage rendering and interactions, because detection and movement were tightly coupled to PIXI objects and it would be unclear to pass updates among different modules.
            - Although JavaScript does not natively support interfaces, based on the structure designed by the author, an instance assigned to `drawingManager` is expected to always contain the methods:
                - `redraw()` to redraw the arena after a window size change.
                - `cleanUp()` to clean the drawing surface when switching to a different view.

        - All states that display menus use the `draw()` method, which renders `screenContent` to the screen.
        - The game session state uses the `update(delta)` method, which updates the game session and, if necessary, also redraws the arena.

- **game_session.js**: Module containing the `GameSession` class, representing a game session.
    - The `GameSession` class stores information about the game session's state in the `gameSessionState` variable (an instance of the `GameSessionState` class defined in `game_session_state.js`).
    - The `start()` method must be called first to initialize the session.
    - The main method is `update(delta)`, which updates the session state.
    - The `update(delta)` method is called from **game.js** and ensures regular redrawing and updates of game logic with the current time difference between frames.
    - It works on the principle of a state machine, where different updates are performed depending on the session state (a switch statement is used for branching).
    - Depending on the session state, methods are called to update parts of the game:
        - `handleGameSessionInProgressUpdate(delta)`: Updates an ongoing session. The sequence of updates is:
            1. Update game time.
            2. Update the HUD using `updateStats(delta)`.
            3. Update game entities using `updateEntities(delta)`. This method returns information about any score gained, which must be processed.
                1. First, gather information about all entities on the field (player, enemies, bombs, power-ups, ...).
                2. Then update the player entity (collisions with entities, then movement).
                3. Then update enemies (collisions with entities, then movement).
                4. Then update destructible walls (collisions with entities).
                5. Then update bombs (planting bombs, explosion timer).
                6. Then update bomb explosions (creating explosions, explosion duration timer).
                7. Finally, check whether escape doors can appear (if the conditions are met).

            - > Updates occur only when the arena is not currently being scaled (due to a window size change).

        - `handleGameSessionPlayerHitUpdate(delta)`: Updates the session after the player is hit by an enemy.
            - Ensures player blinking on hit and updates player lives.
            - If the player loses all lives, the session state changes to `GAME_SESSION_STATE_GAME_END`.
            - If the player still has lives, the session state changes to `GAME_SESSION_STATE_LEVEL_INFO_SCREEN`.
        - `handleGameSessionLevelInfoScreen(delta)`: Updates the level info screen.
        - `handleGameSessionPausedUpdate()`: Updates the paused screen.
        - `handleGameSessionLeavePromptUpdate(delta)`: Updates the leave-game prompt screen.
        - `handleGameSessionGameEndUpdate(delta)`: Properly ends the game session.

        - Most of these methods work with the `screenContent` variable, which contains all logical objects to be drawn to the screen (for a session: the arena, player, enemies, HUD, prompts, ...).

- **arena.js**: Module containing the `Arena` class, representing the playing field.
    - The `Arena` class holds information about the field (represented by a 2D array) relative to the screen.
    - Includes methods for drawing and redrawing the field.
    - Includes methods for retrieving information about the field (get the 2D index based on coordinates, get field coordinates based on a 2D index, ...).
    - Includes a method to check collision with the field (the calculation method is passed as a parameter).
    - It also includes:
        - `draw()`: Draw the field.
        - `redraw()`: Redraw the field.
        - `cleanUp()`: Clean up the field.

- **entity.js**: Module containing the `Entity` class, representing a game entity.
    - The `Entity` class holds information about the entity relative to the field.
    - Includes methods for drawing the entity when created and updating the entity (collision detection and movement).
    - All game entities (player, enemies, bombs, explosions, ...) inherit from `Entity` and override `update(delta)` as needed.
    - Includes methods:
        - `spawn(x, y)`: Create the entity on the field at the given coordinates.
        - `update(delta)`: Update the entity.
        - `redraw()`: Redraw the entity (on window size change).
        - `moveToTop()`: Bring the entity to the foreground.
        - `remove()`: Remove the entity from the field.

- **levels_config.js**: Module containing level configuration for *normal* mode.

> Most elements are positioned/scaled based on predefined ratio constants (relative to screen width/height or a given entity). All these constants are either defined at the beginning of the module, at the beginning of the class, or are passed as parameters to methods.

## 6. Testing

The application was tested only by manual testing. The testing focused on verifying the game's functionality and the correct behavior of the game logic.

## 7. Algorithmic and Implementation Details

The way game elements are drawn in the game session went through major changes during development. In the original version, rendering and entity movement were handled externally. As development progressed, it turned out to be better to let entities manage their own rendering and movement within their instances, because the differences in update approaches based on entity type could be addressed by inheriting from the parent, default `Entity` object. Before sending the version to testers, the code underwent more extensive changes.

Otherwise, in the author's opinion, the code follows standard practices for topics such as state machines, rendering graphical primitives to the canvas, pre-scaling, hit testing, movement based on delta time, etc. A section of code worth mentioning—because it is a custom algorithm—is enemy movement from **enemy.js**. The way the enemy moves is as follows:

```
1. The enemy checks whether it has a target set.
2. If it doesn't have a target, it tries to find one as follows:
    1. With a certain probability (based on enemy difficulty), it chooses whether its target will be the player or a random point on the field.
    2. With a certain probability (based on enemy difficulty), it chooses how to compute the path to the target.
        - At higher difficulty, a BFS algorithm is more likely to be used to find the shortest path to the target.
        - At lower difficulty, a DFS algorithm is more likely to be used to find the shortest path to the target (a variant with random branch selection).
    3. The computed path to the target consists of a list of 2D field indices.
3. If it has a target, it tries to reach the next point on the path to the target by simply adding/subtracting from its coordinates (distance is computed in line with the player's speed).
4. It continues trying to reach the target until it is within range of the target.
5. Once it is within range of the target, the path is discarded and the algorithm repeats from step 1.
```