ChatGPT

Customizable Chess & Board Game Engines

We surveyed existing open-source engines and frameworks for supporting rich variant chess/board games. No single mature engine meets *all* requirements (hex/circular boards, square states, programmable triggers, web embedding), but several come close:

- **Fairy-Stockfish (GPL)** – A variant of Stockfish that supports many chess variants via text configuration [1]. It can handle diverse piece moves (Xiangqi, Shogi, crazyhouse, etc.) but is limited to rectangular boards and has no built-in "trigger/effect" system. As a UCI engine in C++, it is not directly embeddable in a web app (would require a server or compiling to WebAssembly).

- **Jocly (AGPL)** – A JavaScript library for abstract board games with built-in support for many variants. Its demos include **circular** and **hexagonal** chess [2], and it provides a ready web UI (2D/3D) and AI. Rules are defined via game modules; custom piece moves are possible, but advanced triggers (like on-move transforms) must be coded in its game script. Embedding in a TypeScript/Next.js app is feasible by including the Jocly library (via npm or `<script>` [3]).

- **Boardzilla (AGPL)** – A new TypeScript framework for web board games [4]. It explicitly supports *arbitrary grid topologies*: e.g. `HexGrid` with rhombus or hex shapes [5], and a flexible `PieceGrid` where pieces can have *irregular shapes* via `setShape(...)` [6]. Boardzilla manages game state, players, turns and animations, and its action API allows chaining custom behaviors. For example, one can attach a `.do()` callback to an action to implement piece transforms or cooldowns [7]. Boardzilla also includes messaging and can hide information per player. As a pure TypeScript library (installable via npm), it runs in browser or Node.js.

- **Boardgame.io (MIT)** – A general JS/TS framework for turn-based games [8]. It provides state management, multiplayer sync, AI, etc., and the developer writes simple move functions that update a central state (`G`). It is *view-layer agnostic*, so you must code or integrate your own board UI and movement logic. In principle, you can represent any board shape (square grid, hex grid, graph of spaces) in `G`, and implement movement accordingly. Triggers/effects would be coded in the game reducer or React layer (e.g. detecting threats after each move). Boardgame.io does not natively provide piece-shape or grid abstractions, but it is flexible and web-ready (MIT license, React-friendly).

- **ChessCreator (MIT)** – A lightweight JS engine for chess variants on a standard grid [9]. You supply piece move-offsets (singular or repeating) and it generates legal moves. It supports standard and Xiangqi boards out of the box. It does *not* handle irregular boards or square states, and there is no trigger system beyond basic captures and promotions. It is easy to embed (just a JS file) but very specialized to "chess-like" grids.

- **Ludii (CC-BY-NC-ND)** – A powerful general game system (Java) supporting *hundreds* of games via a descriptive language. Ludii can handle arbitrary board geometries (hex, stars, custom graphs) and has a rich set of atomic actions. For instance, it includes actions to hide/show a cell or change its state (e.g. `ActionSetHiddenState`, `ActionSetState`) and to move or transform

pieces [10] [11]. However, Ludii's license is restrictive (non-commercial) and it's a Java-based desktop engine, so not easily embeddable in a TS/Next.js app.

- **Zillions of Games (commercial)** – Not open-source, but historically the closest to this vision. It allowed arbitrary boards and scripted rules (ZRF language) and came with a generic AI. It ran only on Windows and is proprietary, so it cannot be directly used or extended in a web project, but it shows that such flexibility is conceptually possible.

Table: Comparative Summary

Engine/ Framework	License	Board Shapes	Square States	Piece Triggers/ Effects	Embeddable (Web/TS)	Notes
Fairy-Stockfish	GPLv3	Rectangular (8×8 etc.)	✗ (no)	✗ (standard moves only)	No (C++ CLI or UCI)	Very strong, supports many variants [1] but static boards.
Jocly	AGPL	2D/3D, includes hex/circular [2]	Partial (hidden?)	Limited (can script moves)	Yes (npm/ browser)	Includes many demos (hex, circular chess). No built-in trigger system.
Boardzilla	AGPL	Fully arbitrary: hex, custom (PieceGrid) [5] [6]	Yes (via game state)	Yes (action chaining, `.do()` callbacks) [7]	Yes (npm/ browser)	Designed for web. Explicit API for custom boards and actions.
Boardgame.io	MIT	Any (developer-defined graph)	Yes (state data)	Yes (by coding move functions)	Yes (npm/ browser, React)	Generic engine. Developer must implement board geometry and triggers manually.
ChessCreator	MIT	Rectangular (chess grids)	✗ (no)	No (only static move generation)	Yes (standalone JS)	Simple chess variants engine [9], no advanced effects.

Engine/ Framework	License	Board Shapes	Square States	Piece Triggers/ Effects	Embeddable (Web/TS)	Notes
Ludii	CC BY-NC-ND	Arbitrary (many shapes)	Yes (see actions)	Yes (wide action set [10])	No (Java desktop)	Very flexible, but license not open-source and not web-friendly.
Zillions of Games	Proprietary	Arbitrary (any graph)	Yes	Yes (scriptable)	No	Commercial PC software; scripting language (ZRF), now discontinued.

Assessment: Of the above, **Boardzilla** is the most comprehensive and modern solution for a web/TypeScript project. It natively supports diverse board geometries (hex grids, irregular shapes) and has an actions API for implementing custom piece behaviors (with examples of `.do()` callbacks and messaging [7]). **Jocly** is a more mature library with many built-in games, but its development has waned and it has a steeper learning curve. **Boardgame.io** is highly flexible and easy to integrate with React/Next.js, but requires you to hand-code movement rules and trigger logic in JS. **Fairy-Stockfish** is great for strong AI play but not for custom board logic or web embedding. **ChessCreator** is limited to chess variants on rectangular boards. **Ludii** has nearly all needed features (as shown by its rich action list [10] [11]) but cannot be directly used in a web app due to its licensing and Java requirement.

Building a Custom Engine in TypeScript

If none of the above perfectly fits your needs (or if you want full control), a custom TS engine can be designed with these components:

- **Board Representation:** Model the board as a *graph* of spaces (cells). Each space has a coordinate or ID and a list of adjacent spaces (for square, hex, or any tiling). For example, use arrays/objects or a library (e.g. Honeycomb.js for hex grids). To support *custom shapes*, you can assign a "shape mask" to multi-cell pieces or boards. (Boardzilla's `setShape` feature is one approach [6].)

- **Piece and Movement Logic:** Define a class for each piece type with its movement rules (offsets, range, jump capability). For arbitrary boards, generate moves by following graph connections. For example, a queen-like piece would do a BFS in all directions until blocked. Movement code should account for blocked/disabled spaces (you can mark some spaces as inactive or check occupancy). A generic move-generator can be driven by per-piece parameters (vectors and "sliding" vs "leaping" behavior, as in ChessCreator [12]).

- **Triggers/Effects System:** Implement an event-driven or rule-trigger mechanism. For instance, after each move, iterate relevant pieces or use observers to check conditions (e.g. `onMove`, `onThreatened`). Triggers can be functions stored in piece definitions. For example:

```
interface Trigger { event: string, action: (game, piece) => void }
class PieceType { triggers: Trigger[]; /* ... */ }
```

Then after handling a move, call triggers whose `event` matches. For "onThreatened", after each move you could check for any opponent pieces that could capture a given piece and invoke that trigger. Effects might include transforming a piece (`piece.type = newType`), removing a piece, adding a piece, or setting a cooldown counter on a piece (decremented each turn). This is analogous to Ludii's actions like `ActionSetValue` or `ActionMove` [13], but implemented in JavaScript functions.

- **State and Serialization:** Keep all game state in JSON-friendly objects: e.g. `{spaces: [...], pieces: [...], currentPlayer: ...}`. This makes it easy to save/load or send over network. You can define a JSON "ruleset" format describing the board (size/shape or adjacency list), initial piece setup, and piece types with their movement offsets and trigger definitions. For example:

```
{
  "board": { "type": "hex", "size": 7 },
  "pieces": [
    { "name": "Soldier", "moves": [[1,0],[0,1]], "triggers": [
        {"event":"move","effect":"promote","target":"General"},
        {"event":"threatened","effect":"remove"}
     ] }
  ],
  "setup": [ {"piece":"Soldier","space":"A1"} ]
}
```

Your TS engine would parse this config to create `Board`, `PieceType`, and `Piece` instances, then handle turn-by-turn play using these rules. This approach (data-driven rules) is similar in spirit to Fairy-Stockfish's variant config and Ludii's game descriptions.

- **Modularity:** Structure the engine into modules (e.g. `Board`, `Piece`, `MoveGenerator`, `GameController`). Define clear interfaces so new piece types or board shapes can be added without rewriting core logic. For example, a `Grid` interface can have multiple implementations (SquareGrid, HexGrid, CustomGraph) abstracting adjacency queries.

In summary, none of the existing engines is a perfect fit, but **Boardzilla** and **Boardgame.io** come closest for a TypeScript web app. If you opt to build your own, follow a **model-view-controller** style: a clear data model for the board/pieces, a controller for the rules/move generation (with an event/trigger system), and your Next.js frontend for the UI. Saving and loading variants can be done by serializing the model to JSON (just ensure your classes can be reconstructed from the data).

Recommendation: For most projects, we recommend using **Boardzilla** (AGPL) if you want a rich, out-of-the-box framework supporting arbitrary shapes and triggers [5] [7]. It is specifically designed for extensible board games on the web. If Boardzilla's license or learning curve is a concern, consider **Boardgame.io** (MIT) combined with a custom board layer (e.g. a hex/grid library) to handle movement and effects in code [8] [7]. Use Fairy-Stockfish or Jocly only if you mainly need a strong AI for standard

variant rules; otherwise, a bespoke TypeScript solution (inspired by the patterns above) will give you the full flexibility you need.

Sources: We drew on documentation for Jocly [2], Boardzilla [5] [7], Boardgame.io [8], Fairy-Stockfish [1], and Ludii [10] [11] to compare capabilities. These illustrate how each tool handles (or omits) custom boards, states, and triggers.

---

[1] Fairy-Stockfish | Open Source Chess Variant Engine
https://fairy-stockfish.github.io/

[2] [3] GitHub - aclap-dev/jocly: Javascript library and tools to provide user interface (2D, 3D, VR) and engine for playing board games
https://github.com/aclap-dev/jocly

[4] GitHub - boardzilla/boardzilla-core: Boardzilla core library
https://github.com/boardzilla/boardzilla-core

[5] [6] Adjacency and Grids | Boardzilla docs
https://docs.boardzilla.io/game/adjacency

[7] Actions | Boardzilla docs
https://docs.boardzilla.io/game/actions

[8] GitHub - boardgameio/boardgame.io: State Management and Multiplayer Networking for Turn-Based Games
https://github.com/boardgameio/boardgame.io

[9] [12] GitHub - Austinae/chesscreator: A javascript library to easily integrate a chess board with custom chess piece rules into your website.
https://github.com/Austinae/chesscreator

[10] [11] [13] ludii.games
https://ludii.games/downloads/LudiiGameLogicGuide.pdf