Deployment Approach (today-friendly)

1. Recommended Deployment Approach (today-friendly)

✅ Stack choice

Use:

- **Frontend:** TypeScript + either
  - **Next.js (App Router) +** `<canvas>` (my top pick), or
  - **Next.js + Phaser.js** for a more “game-enginey” feel.
- **Hosting: Vercel** (static + serverless).
- **Backend (later):** Vercel serverless functions or a lightweight external service (Supabase, etc.) for multiplayer or persistent logs.

For a **single-player Rampart** and your first iterations, you can:

- Run the **entire game loop in the browser**.
- Deploy a **pure frontend** to Vercel.
- Optionally post logs to a **single API route** (`/api/log`) that just appends them to a file / KV / DB later.

This gives you:

- One GitHub repo.
- One Vercel project.
- Automatic deployments on push.
- Sandbox URLs for each branch/PR to playtest.

2. Concrete “Get It Live Today” Plan

Step 1 – Create a minimal Next.js game shell

Goal: You can hit a URL and see *something moving* that’s tied to your game loop.

1. Create a repo `rampart-remake` on GitHub.
2. Initialize Next.js (App Router, TS):
   - `npx create-next-app@latest rampart-remake`
3. In `app/page.tsx` (or `src/app/page.tsx`):
   - Render a `<canvas>` element.
   - Attach a simple `useEffect` that:
     - Grabs the canvas context.
     - Runs a `requestAnimationFrame` loop.
     - Clears the canvas and draws a simple grid or rectangle that moves.

Now you have:

- A **main game loop** (render loop) running in the browser.
- One place to hang your future systems.

Push to GitHub → connect repo to **Vercel** → deploy. Boom: “live today.”

Step 2 – File & Folder architecture for the game logic

Inside your Next.js app, create a **game folder structure** like:

```
/src/game/core          // game loop, phases
/src/game/grid          // tiles, walls, castles
/src/game/systems       // build, combat, territory, AI
/src/game/types         // shared enums/interfaces (Tile, Phase, etc.)
/src/game/logging       // logger
```

This lets you:

- Keep the **React UI thin** (only rendering + input).
- Keep the **simulation pure TS** (easier to test, maybe port later).

Your `page.tsx`:

- Instantiates a `Game` instance (from `/src/game/core/Game.ts`).
- Calls `game.update(deltaTime)` in your loop.
- Calls `game.render(ctx)` to draw to the canvas.

Step 3 – Logging from day one

Inside `/src/game/logging/Logger.ts`:

- Create a simple interface:

```
export type LogLevel = 'info' | 'warn' | 'error' | 'event';

export interface LogEventPayload {
  [key: string]: unknown;
}

export class Logger {
  constructor(private context: string) {}

  info(message: string, payload?: LogEventPayload) { /* ... */ }
  warn(message: string, payload?: LogEventPayload) { /* ... */ }
  error(message: string, payload?: LogEventPayload) { /* ... */ }
  event(name: string, payload?: LogEventPayload) { /* ... */ }
}
```

- For **Phase 1**, just write to:
  - `console.log` (with `[context]` prefix).

    - Optionally push into an in-memory array for “session logs” visible via a debug panel.

Later, you can:

- Add a **Next.js API route** `/api/log` that:
    - Receives JSON log events.
    - For now: just `console.log` them server-side, or write to a KV / DB.
- Add an environment flag: client-only logs in dev, server logging in prod.

---

3. What to Build in What Order (Deployment-aware)

Here’s how to combine the **build order** with “works on Vercel right now”.

Phase A – Single-page, single-player, no backend

All in the browser:

1. **Minimal Game Loop (canvas working)**
    - Already described: draw something, animate it.
2. **GridSystem + simple map**
    - Hardcode a map in TS.
    - Render tiles as colored squares on the canvas.
    - Log `MapLoaded`.
3. **Phase State Machine**
    - Implement `GamePhase` and cycle between phases automatically.
    - Draw different background color per phase so you can see it working.
4. **Build Phase – Place blocks**
    - Add fake “piece” (1×3 rectangle).
    - Move it with arrow keys or WASD.
    - Place it with spacebar.
    - Update tiles, re-render.

5. **Territory detection (single castle)**
    - Implement enclosed / not-enclosed logic.
    - If not enclosed when Build timer ends → game over.
6. **Cannon placement & shooting**
    - Allow 1 cannon inside enclosed area.
    - On Combat phase, press arrow keys to aim, space to fire.
    - Draw simple projectiles.

All of this runs **entirely client-side**, so Vercel is just serving static assets.

Phase B – Light server-side logging (optional but nice)

When you're happy with core loop:

1. Add `/app/api/log/route.ts` (Next.js API Route).
2. POST JSON events from the Logger when:
    - Phase changes.
    - Player places a piece.
    - Player loses.
3. For now, on the server just:
    - `console.log("LOG_EVENT", body);`
    - Optionally write to a simple DB (Vercel Postgres or KV) later.

You've now got:

- **Telemetry for sessions**, even in production.
- A path to build **replay tools** or analytics later.

Phase C – Preparing for multiplayer (future-friendly, still on Vercel)

When (not if 😄) you want multiplayer:

1. Keep the **simulation deterministic**:

- Use a seeded RNG.
- Step the game in **fixed ticks** (e.g., 60 Hz) inside your Game class.

2. Use:
    - Vercel serverless functions for **matchmaking**.
    - WebSockets (via a 3rd-party service like Ably/Supabase Realtime/Pusher) for realtime state sync, OR
    - Turn-based/hybrid: each combat/build phase syncs at phase boundaries instead of every frame.

But importantly: **you don’t have to change where it’s hosted** – just extend what’s already there.