The goal of this exercise is to evaluate how you approach rendering, performance optimization, and architecture decisions in PixiJS. We are less interested in polish and more interested in clear thinking, practical tradeoffs, and senior-level implementation choices.

Please review the task brief below.

We do not expect a production-complete streaming engine.

Instead, we want to see a focused, working solution to the core rendering problem, plus a short explanation of how you would extend it further.

Please include:

- source code
- a short README with setup instructions
- a brief explanation of your architectural decisions and tradeoffs
- notes on what you would improve if given more time if anything is ambiguous, feel free to make reasonable assumptions and document them.

2D World Viewer

Build a small PixiJS application that demonstrates how you would render and navigate a 2D world efficiently. This task is intentionally focused on the core problem. We are not asking for a full production-grade asset streaming engine.

Imagine a large RTS-style world containing a very high number of world objects.

Your task is to implement a viewer where:

1. the world is much larger than the viewport
2. the camera can pan smoothly in any direction
3. the app simulates a large global dataset
4. only visible objects are projected into the active Pixi scene

Core Requirements Implement a PixiJS demo that includes:

1. Large world simulation

- Simulate a world containing at least 100,000 objects in total
- These objects may be tiles, markers, units, decorations, or similar lightweight world items

2. Camera and navigation

- Implement a 2D camera that supports smooth panning
- Infinite panning behavior is welcome, but not required
- The viewer should feel responsive while moving through the world

3. Manual visibility control

- Only objects intersecting the viewport, plus a small configurable margin, should be added to the active Pixi render tree
- Off-screen objects should be excluded from rendering and ideally not exist as active display objects
- The solution should not rely on simply creating all sprites and toggling visible

4. Spatial query strategy

- Use a reasonable spatial lookup strategy to determine what is visible
- This can be a spatial hash grid, chunked grid, quadtree, or another justified approach
- The choice should match the data model and be explained briefly

5. Render pooling

- Reuse Pixi display objects where practical instead of constantly creating and destroying them during panning
- Show that you are thinking about object lifecycle and GC pressure

Deliverables:

A working demo

- runnable locally
- clear instructions in README

Short technical notes

- your approach to visibility/culling
- why you chose your spatial structure
- how you handled render object reuse
- what the main bottlenecks are
- what you would build next if this needed to support streamed map tiles and strict GPU-memory limits

Please submit your solution in a GitHub repository and send us the repository URL, along with any access instructions if the repository is private, so our team can clone it and review the implementation locally.