DM Toolkit- MAPS

Table of Contents

Overview

Section 1: Overview – Maps Portion of the DMToolkit

The Maps module of the **DMToolkit** provides a full-featured, interactive map editing experience for dungeon masters (DMs) to prepare and run tabletop sessions. The system is structured around:

- **Map configuration and metadata**
- **Grid-based map visualization**
- **Fog of war & vision blocking**
- **Token management**
- **Notes and annotations**
- **Layer and tool mode control**

At its core, the toolkit is divided into **UI Panels**, **Interactive Canvas**, and **Layered Konva Components**, all orchestrated through ToolkitMapEditor.jsx.

Main Entry Point: ToolkitMapEditor.jsx

This is the central component that initializes the **editable map environment**, storing map state (mapData) and rendering:

- Toolbar (tool and layer switching)
- Canvas (map visual rendering and interaction)
- Panels (token, size, note editing)

All user interactions (e.g., placing a token, editing fog) update the local map state (mapData), which drives the UI and canvas.

Canvas & Layers Breakdown

Section 2: Canvas & Layers Breakdown

At the heart of the map rendering system is the MapCanvas.jsx component. This uses **React Konva** to render layered visual elements on an HTML5 canvas via the <Stage> component.

2.1 Structure of MapCanvas.jsx

- **Stage Setup**: The <Stage> component creates the root canvas area. It's scaled based on the map's dimensions and rendered using individual Layer components for separation of visual concerns.
- **User Interaction**: MapCanvas handles mouse events for painting blockers, placing notes, and token selection/movement via mode-specific behavior.

2.2 Rendered Layers

The canvas composes several layers, each with a specific function:

a. StaticMapLayer.jsx

- Draws the static image of the map (mapImage).
- Renders optional grid lines if enabled.
- Displays note markers (triangles) and visual highlights for selected or active notes.

b. FogAndBlockerLayer.jsx

- Dynamically computes unrevealed cells and draws black fog rectangles.
- Displays red semi-transparent blocker rectangles for line-of-sight obstruction.
- Uses Konva animation (node.to) for smooth fog transitions.

c. MapTokenLayer.jsx

- Iterates over all token layers (player, dm, hidden).
- For each token, renders a TokenSprite instance.
- Only the tokens on the activeLayer can be interacted with (moved or selected).
- Tokens on the "dm" layer are rendered semi-transparently.

d. Reserved Tool Layer

- An empty <Layer> placeholder currently marked as TODO: drawing tools, reserved for future enhancements (e.g., drawing freehand lines or area templates).

2.3 Token Interactivity

The TokenSprite.jsx component:

- Handles token dragging with ghost previews.
- Animates token movement and highlights on selection.
- Displays a side health bar using relative proportions and HP-based coloring.
- Applies client-side movement animations when the immediatePositionOverride is set.

Editor Controls and Panels

Section 3: Editor Controls and Panels

The editor interface is organized around ToolkitMapEditor.jsx, which coordinates **toolbar-driven controls** and **contextual panels** for manipulating map features.

3.1 Toolbar: ToolkitMapEditorToolbar.jsx

This component is the primary UI control surface for:

- Navigating back to the maps list
- Toggling grid visibility
- Switching active token layer (player, dm, or hidden)
- Opening the map size or token library panels
- Enabling fog of war and toggling blocker painting mode
- Activating the note-taking tool mode

The toolbar updates the editor state via props like setGridVisible, setActiveLayer, and setToolMode. Each dropdown (e.g., for layers or fog) is controlled via local toggles in the toolbar component itself.

3.2 Token Management Panel: TokenPanel.jsx

This draggable panel allows DMs to search and select from a list of predefined tokens (mockTokens.json), rendered using TokenCard components.

Drag Behavior:

- Tokens are draggable using mouse events.
- On drag start: onStartDrag(token) sets the draggingToken state in ToolkitMapEditor.
- During drag: onDragMove updates the current ghost position via a ref.
- On drop: handleCanvasDrop in ToolkitMapEditor finalizes token placement into the active layer using createTokenOnDrop.

3.3 Map Size Panel: MapSizePanel.jsx

A small modal used to adjust the width and height (in tiles) of the current map. Upon clicking "Apply", it calls onUpdateSize, updating the mapData in the editor.

3.4 Notes Panel: NotesPanel.jsx

This panel allows the DM to:

- View a list of existing notes placed on the map.
- Create new notes, which get associated with a selected grid cell (activeNoteCell).
- Select a note from the list to highlight its location on the canvas.

Note placement is tied to clicking a cell on the map while in notes tool mode, which sets activeNoteCell. The panel uses this to show coordinate context during creation.

Map Metadata, Configuration, and Entry Points

Section 4: Map Metadata, Configuration, and Entry Points

This part of the system handles the management of maps outside the editing interface. It includes map listing, creation, and configuration details.

4.1 Map List Interface: Maps.jsx

This component presents the DM with a dashboard for all maps tied to the current campaign (useOutletContext() provides currentCampaign). Key features:

- **Search Bar**: Filters displayed maps by name.
- **New Map Button**: Toggles visibility of the MapForm for creating a new map.
- **Map Cards**: Renders multiple MapCard components from a hardcoded or fetched source (currently mapTemplate placeholder data).

4.2 Map Card: MapCard.jsx

Each card represents a single map preview. It includes:

- A thumbnail of the map image.
- Summary info: dimensions, grid size/type.
- Buttons:
    - **Edit**: Navigates to /dmtoolkit/maps/editor passing the full map object in navigation state.
    - **Delete**: Currently static, expected to integrate with future map management logic.

4.3 Map Detail Modal: MapDetail.jsx

A modal overlay that shows when a map is clicked in Maps.jsx. It provides:

- Full image preview
- Configuration summary (size, grid, fog of war, snap to grid, campaign)
- Optional notes metadata

Closes on click-outside or clicking the close button.

4.4 Map Creation Form: MapForm.jsx

This form is used for both creating new maps and editing existing ones (via defaultValues prop). The form includes:

- Text fields for name and image URL
- Numeric inputs for width, height, and grid size
- Dropdown for grid type (square or hex)
- Checkboxes for fog of war and snap-to-grid
- Submission triggers onSubmit, which is stubbed for future backend integration

The form consolidates user input into a formData object stored in local component state.

Fog of War and Blocker Logic

Section 5: Fog of War and Blocker Logic

The **fog of war** and **line-of-sight blockers** are two major systems used to control visibility on the map. They are tightly integrated with both the map data model and user interactions on the canvas.

5.1 Data Model

The fog and blocker data lives under map.fogOfWar, which includes:

- revealedCells: An array of { x, y } tiles that have been exposed to players.
- blockingCells: Tiles that block line of sight (used for pathing and lighting logic elsewhere).

This state is passed into the layers and updated via the canvas tool mode interaction handlers.

5.2 Rendering Logic: FogAndBlockerLayer.jsx

This combined Konva layer renders both:

- **Fog tiles**: Black rectangles with an easing fade-in, drawn only on cells **not** in revealedCells.
- **Blocker tiles**: Red semi-transparent rectangles rendered over blockingCells when blocker mode is active.

Fog visibility is toggled with showFog. Blocker visibility depends on the editor's toolMode.

The fog tiles are generated via useMemo() to avoid unnecessary re-renders, based on map dimensions and the revealed cell set.

5.3 Interaction: Painting Blockers

The MapCanvas.jsx handles painting/removing blockers:

- If toolMode === "paint-blockers", mouse clicks toggle blocker presence on the clicked cell.
- Internally, it checks if the cell exists in blockingCells. If yes, it's removed; if not, it's added.
- The update is applied to the mapData.fogOfWar.blockingCells array, triggering a re-render of the FogAndBlockerLayer.

5.4 Legacy / Alternative Fog Rendering: FogLayer.jsx and BlockerLayer.jsx

unused/alternative components:

- FogLayer.jsx: Does fog rendering without blocker support. Same animation logic, but likely obsolete.
- BlockerLayer.jsx: Renders only blockers without fog integration. Still useful if you choose to split fog/blocker logic visually or maintain modular control.

5.5 Fog and Token Interaction

Token movement via handleTokenMoveWithFog (called in MapCanvas.jsx) updates the revealedCells automatically. This is part of the token movement pipeline that handles revealing fog as tokens explore.

Token Rendering and Drag-and-Drop Architecture

Section 6: Token Rendering and Drag-and-Drop Architecture

Token management in the DMToolkit is built around the idea of interactive, grid-aligned assets that can be dynamically placed, selected, and moved by the user. This system is implemented with strong separation between rendering logic (TokenSprite) and editor control flow (TokenPanel, drag handlers, and canvas logic).

6.1 Token Rendering: TokenSprite.jsx

This is the core component responsible for drawing an individual token on the canvas. Key behaviors include:

- **Image Caching**: Uses getCachedImage() to avoid reloading the image on each render.
- **Visual Positioning**: Maintains visualPos state to animate smooth movement (e.g., when coordinates change externally).
- **Selection**: Applies visual effects (highlight stroke, shadow, yellow fill) if the token is selected.
- **HP Bar**: A vertical bar rendered as three Rects on the left edge, showing current health in color.
- **Rotation and Opacity**: Tokens support rotation and visibility toggling via their rotation and isVisible attributes.

Drag Handling

TokenSprite supports dragging through:

- useTokenDrag() — a custom hook managing ghost token behavior and drag lifecycle
- Ghost rendering: A translucent preview of the token appears during dragging
- Real movement updates are deferred until drag end, with visual feedback (e.g., shadow, gold outline) shown in the meantime

Animations for token repositioning are implemented using requestAnimationFrame to interpolate position deltas over time.

6.2 Token Layer: MapTokenLayer.jsx

This layer is responsible for rendering all tokens from all layers (player, dm, hidden) by:

- Iterating over map.layers entries
- Rendering a TokenSprite for each token
- Passing props like isSelected, onSelect, onTokenMove based on the current activeLayer

Only tokens on the currently active layer are interactive — others are inert and semi-transparent (if in the dm layer).

6.3 Token Placement: TokenPanel.jsx + ToolkitMapEditor.jsx

Token drag-and-drop behavior starts in the TokenPanel, where each token in the library can be:

- Click-dragged using onMouseDown
- Tracked across the screen using window-level mousemove and mouseup listeners

During the drag:

- draggingToken is stored in ToolkitMapEditor
- draggingPositionRef is updated with current pointer position
- MapTokenDragGhost renders a visual preview following the pointer

Upon mouse-up over the map:

- handleCanvasDrop() is triggered with the pointer position
- createTokenOnDrop() calculates grid-aligned position and constructs a new token object
- This token is inserted into the activeLayer.tokens array in mapData

Perfect — we'll keep this update scoped strictly to the map editor portion. Here's a refined section tailored to that:

6.4 Map Asset Layer and Interaction

The map editor now includes support for **Map Assets**, allowing DMs to place non-token visual elements (e.g., props, scenery, decorations) directly onto the map canvas.

Rendering: MapAssetLayer.jsx

A new Konva-based rendering layer has been added:

- **MapAssetLayer.jsx** renders all assets assigned to the current activeLayer.

- Each asset is rendered using a sprite component and supports selection and dragging.
- Assets are stored in map.layers[activeLayer].assets, separate from tokens.

Assets are drawn within the existing rendering order as follows:

1. StaticMapLayer
2. FogAndBlockerLayer
3. **MapAssetLayer**
4. MapTokenLayer
5. Reserved Tool Layer

Editor Behavior: MapCanvas.jsx

MapCanvas.jsx now includes full support for map assets:

- **Asset Placement:** Assets can be dropped on the canvas, and placed freely on the map.
- **Selection & Rotation:** Clicking an asset selects it. Pressing **Q** or **E** rotates the selected asset in 15° increments.
- **State Updates:** selectedAssetId is managed in component state. Updates to asset position or rotation are immediately applied to mapData.

Asset interactions are restricted to the current activeLayer, following the same convention as tokens.

Toolbar Integration: ToolkitMapEditorToolbar.jsx

The editor toolbar includes a new **"Assets"** button which, when clicked, opens the AssetPanel. This enables asset browsing and drag placement into the canvas.

Notes System

Section 7: Notes System

The notes system allows DMs to annotate specific map cells with custom labels and descriptions. These notes serve as embedded reminders or secret metadata tied to grid coordinates.

7.1 Data Structure

Notes are stored in mapData.notes, an array of objects with the following structure:

```
{
 id: string,
 name: string,
 body: string,
 cell: { x: number, y: number } | null
}
```

Each note may or may not be placed on the map. Notes with an associated cell appear as visual markers.

7.2 Note Creation and Management: NotesPanel.jsx

This panel provides two modes:

- **List View**: Displays existing notes, including title, body, and placement info.
- **Create View**: Allows input of a new note's title and body.

When creating a note:

- The active tool mode must be "notes".
- Clicking a grid cell sets activeNoteCell.
- The panel then uses that cell to place the new note upon save.
- Notes are added to mapData.notes via onUpdateNotes.

Selecting an existing note in the list sets selectedNoteCell, which triggers a highlight box on the canvas.

7.3 Placement Logic: MapCanvas.jsx

When the canvas is in "notes" tool mode:

- Clicking a cell triggers setActiveNoteCell(cell) to store the target.
- This cell is passed into NotesPanel and reflected in the UI.
- Note markers and highlights are rendered via the visual layers.

7.4 Visual Representation

There are two sources for note rendering:

a. StaticMapLayer.jsx

- Renders orange triangular markers in the bottom-right corner of the associated grid cell.
- Draws highlight outlines (yellow dashed for activeNoteCell, red solid for selectedNoteCell).
- These are updated reactively based on props.

b. NoteMarkersLayer.jsx

- A separate component capable of rendering just the triangle markers.
- Currently not used in the active canvas; included for optional or alternate rendering pipelines.

Grid and Base Map Rendering

Section 8: Grid and Base Map Rendering

This section outlines how the static background elements of the map are drawn—namely the base map image, the grid overlay, and coordinate alignment that supports all interactive features.

8.1 Map Image Rendering

a. MapImageLayer.jsx

- A simple Konva Layer that renders the background image using a <KonvaImage /> component.
- Accepts props: image, width, height, and imageReady.
- Skips rendering if the image is not loaded (imageReady === false).
- Intended to be lightweight, a minimal version or early-stage layer before StaticMapLayer expanded in function.

b. StaticMapLayer.jsx

- The main image rendering layer used in the live editor (MapCanvas.jsx).
- Extends the role of MapImageLayer by also:
    - Drawing optional grid lines
    - Rendering note markers
    - Drawing selection highlights for notes

This layer pulls data from:

- mapImage (preloaded with useImage)
- map object (for dimensions and grid size)
- notes, activeNoteCell, and selectedNoteCell

Image rendering uses perfectDrawEnabled={false} for performance, which disables anti-aliasing optimizations.

8.2 Grid Rendering

a. Inline Grid Rendering (StaticMapLayer)

- If gridVisible is true, vertical and horizontal lines are drawn directly in the same layer.
- Uses for loops to draw one line per tile column/row.
- Stroke color is hardcoded as #444.

b. GridLayer.jsx (Alternative Grid Renderer)

- An isolated layer that performs the same function as the inline version in StaticMapLayer.
- Uses useMemo() to optimize grid re-rendering based on width, height, gridSize, and color.
- Intended for cases where the grid might need to be toggled or re-rendered independently.

Currently, the grid is drawn inline in StaticMapLayer, and GridLayer is available as a potential drop-in replacement for greater separation of concerns.

8.3 Coordinate Mapping

All grid-based elements—including tokens, notes, fog, and blockers—are aligned using:

x * gridSize

y * gridSize

This ensures that every layer remains perfectly aligned on the grid, enabling intuitive and pixel-accurate user interactions.

Map Editor Data Flow

Section 9: Map Editor Data Flow

This section outlines how map state is managed, updated, and shared across the various components of the map editor.

9.1 Centralized State: ToolkitMapEditor.jsx

The source of truth for the editable map is the mapData object, initialized from navigation state (state.map) and extended with:

- Fog of war structure (if not already present)
- Hardcoded example notes

All edits to the map—size changes, token placements, fog updates, etc.—are executed by modifying mapData via setMapData.

Core Fields of mapData:

- width, height: Map grid dimensions
- gridSize, gridType: Visual size and structure of the grid
- layers: An object keyed by layer type (player, dm, hidden), each containing tokens
- fogOfWar: Contains revealedCells and blockingCells
- notes: Array of user-authored map annotations

9.2 Update Pathways

Changes to mapData can come from multiple UI entry points:

a. Canvas Interaction (MapCanvas)

- Clicking a cell in tool modes (notes, paint-blockers) updates fog or selected notes.
- Dragging tokens causes an update to token positions within mapData.layers[activeLayer].tokens.
- Movement may also trigger fog revealing, via handleTokenMoveWithFog().

b. Toolbar (ToolkitMapEditorToolbar)

- Changes gridVisible, fogVisible, toolMode, and activeLayer.
- Indirectly controls what components are shown and how the canvas behaves.

c. Panels

- **TokenPanel**: Adds new tokens to mapData when dropped.
- **MapSizePanel**: Modifies width and height.
- **NotesPanel**: Updates mapData.notes when notes are added, edited, or selected.

All panel state changes are synced to the canvas on the next render, ensuring consistency between UI state and rendered elements.

9.3 Drag-and-Drop Syncing

The drag ghost for tokens is rendered using MapTokenDragGhost, a shared component that pulls position data from a ref (draggingPositionRef). This separation keeps pointer tracking independent of the React re-render cycle.

The actual token is committed to the map only once dropped, keeping performance responsive even during large drags.

9.4 Live Canvas Feedback

Thanks to React Konva's efficient layering and use of useMemo, updates to tokens, fog, and notes propagate with minimal performance overhead. Animations and drag indicators are handled on the client side, ensuring smooth interactivity even during rapid changes.

Summary and Extension Points

Section 10: Summary and Extension Points

This final section provides a high-level summary of the system's current structure and outlines designed extension points for future development.

10.1 System Summary

The **Maps portion of the DMToolkit** provides a feature-rich, modular map editing environment tailored for tabletop RPGs. It is centered on:

- A **layered canvas architecture** using React Konva, enabling clean separation of visual elements (map image, tokens, fog, notes).
- A **reactive and centralized data flow**, where mapData in ToolkitMapEditor serves as the authoritative source for all map state.
- A **robust interaction model**, supporting drag-and-drop token placement, cell-based annotation, and dynamic fog/blocker editing.
- A **toolbar-driven editor interface** that allows real-time toggling of visibility layers, tool modes, and editor panels.
- A **component-per-concern** structure that allows for testability and separation of logic across UI, canvas, and data layers.

10.2 Extension Points

This system is structured to support continued development. Potential enhancements include:

- **Drawing Tools**: The reserved <Layer>{/* TODO: drawing tools */}</Layer> in MapCanvas can be filled out with tools like freehand lines, shapes, and area markers.
- **Token Metadata Panel**: A side panel for editing token attributes (name, HP, conditions) directly after selection.
- **Backend Integration**: Currently, map loading and saving are placeholder-based (e.g., mapTemplate). Hooking this into a backend would allow persistent storage.
- **Undo/Redo Support**: Changes to mapData could be wrapped in a state history system to allow undo/redo of edits.

- **Collaborative Editing**: With socket support, the architecture could support multi-user sessions with per-user layer visibility.
- **Grid Modes**: Support for non-square grids (e.g., hex) is stubbed but not yet implemented in the canvas rendering or token logic.

10.3 Code Readiness

- Most components are written in idiomatic, modern React.
- Re-render performance is managed using useMemo and careful separation of animated effects.
- Visual transitions (fog reveal, drag feedback) use native Konva animation APIs for fluid rendering.