Session 3 — Managing Game Flow

You're about to unlock one of React's most powerful features — shared state that controls your entire app! This guide walks you through implementing screen navigation, understanding the difference between local and shared state, and using React's Context API to manage game flow. Ready to make your buttons actually navigate? Let's go!

Table of Contents

☁️ Accessing Your Codespace

Visit github.com/codespaces to relaunch your Codespace from Session 2.

🧠 State vs Props

Before we dive into code, let's understand the key difference between **state** and **props** — two fundamental concepts that control how data flows in React apps.

Props vs State: The Key Differences

Props	State
Data flows **down** from parent to child	Data lives **inside** a component

Read-only — child can't change them	**Changeable** — component can update it
Like function parameters	Like component memory
External data	Internal data

Props are like ingredients you receive to make a recipe — you can't change them, but you use them to create something. **State** is like your kitchen's current condition — you can rearrange, add, or remove things as needed.

> 💡 State vs Props
>
> Understanding this difference is crucial because it determines how data flows through your app and which component is responsible for managing what information. **Props** flow down from parent to child (one-way data flow), while **state** lives inside a component and can be updated by that component.

🏆 Adding Local State for Credits

🎯 **Goal:** Implement local state for the credits modal to see how components can manage their own data.

File: `src/components/SplashScreen.jsx`

Step 1: Add imports to SplashScreen

To use React's state management and the credits modal, we need to import `useState` and `CreditsModal`.

```
// Add these two imports
import { useState } from "react";
import CreditsModal from "./CreditsModal";
// Existing imports below
import GameButton from "./GameButton";
import GameLogo from "./GameLogo";
```

Step 2: Add local state

Create a state variable to track whether the modal is visible, starting with `false` since the modal should be hidden initially. The `useState` hook returns both the current value (`showCredits`) and a setter function (`setShowCredits`) that you'll use to update the state later.

```
export default function SplashScreen() {
  // Add this state to track modal visibility
  const [showCredits, setShowCredits] = useState(false);

  // ... rest of function ...
}
```

Step 3: Update the Credits button onClick

Connect the Credits button to your state by updating its onClick handler to show the modal when clicked.

```
<div className="splash-buttons">
  <GameButton
    text="Start Adventure"
    onClick={() => alert('Start Game!')}
    variant="primary"
  />
  <GameButton
    text="Credits"
    onClick={() => setShowCredits(true)}
    variant="secondary"
  />
  {/* ↑ Update onClick to set state */}
</div>
```

Step 4: Add the modal to JSX

Conditionally render the modal in your JSX so it only appears when `showCredits` is true, and pass a function to close it.

```
<div className="splash-screen">
  <GameLogo />
  <div className="splash-buttons">
    {/* ... existing buttons ... */}
  </div>

  {showCredits && <CreditsModal onClose={() => setShowCredits(false)} />}
  {/* ↑ Add this line: show modal when showCredits is true */}
</div>
```

Step 5: Test the modal

Test the complete flow by clicking the Credits button to open the modal, then closing it.

✓ **You should see:** The credits modal appears! Click outside or the close button to dismiss it.

> 💡 **Giving Components Their Own Memory**
>
> The `useState` hook gives a component its own memory that persists between renders. Since the credits modal only affects `SplashScreen`, we use component-level state rather than shared state. This pattern keeps data isolated where it belongs — only the component that needs it manages it.

> 🏆 **Bonus Challenge**
>
> Use React DevTools to inspect the SplashScreen component and watch the `showCredits` state change as you interact with the Credits button.

📋 Screen Constants

Before implementing navigation, let's understand how **constants** organize your game's screens and prevent errors.

File: `src/constants/screens.js`

Here's the SCREENS object that defines your game's navigation states:

```
export const SCREENS = {
  SPLASH: "splash",              // ← Initial screen users see
  PLAYING: "playing",            // ← Active gameplay screen
  GAME_OVER: "gameover",         // ← End screen after completing all zones
};
```

How Constants Work in Your Game

Constants are named values that stay the same throughout your app. Instead of typing the string `"splash"` every time you need to reference the splash screen, you use `SCREENS.SPLASH`.

When you write:

```
if (screen === SCREENS.SPLASH) {
  return <SplashScreen />;
}
```

JavaScript sees:

```
if (screen === "splash") {
  return <SplashScreen />;
}
```

The constant `SCREENS.SPLASH` resolves to the string `"splash"`. This means you get autocomplete in your editor, protection from typos, and the ability to change the value in one place if needed.

In your game, you'll use these constants to:

- Check which screen is currently active: `screen === SCREENS.PLAYING`
- Change to a different screen: `setScreen(SCREENS.GAME_OVER)`
- Render the right component: `{screen === SCREENS.SPLASH && <SplashScreen />}`

> 💡 **Constants Create a Single Source of Truth**
>
> Constants create a "single source of truth" — one place where values are defined and referenced everywhere else. This same pattern works for any fixed values in your app: API endpoints, error messages, color themes, or game settings. When you see `SCREENS.SPLASH` in your code, you immediately know it's a screen constant. Your code becomes self-documenting, making it easier to understand what's happening at a glance.

🔗 Context and Prop Drilling

Now that you've seen the SCREENS constants, let's understand why we use **Context** to share the current screen value across components.

The Prop Drilling Problem

Prop drilling is when you have to pass data through multiple component levels, even when the middle components don't need that data. It's like having to ask your friend to ask their friend to ask their friend for something — inefficient and annoying.

Example of prop drilling:

```
App (has screen state)
 ↓ passes screen as prop
SplashScreen (doesn't need screen, just passes it along)
 ↓ passes screen as prop
GameButton (finally uses screen)
```

Every component in the chain needs to accept and pass along the `screen` prop, even if it doesn't use it. This creates brittle code that's hard to maintain.

The Context Solution

Context lets any component access shared data directly without passing it through every level:

```
GameProvider (provides screen state)
 ↓ any component can access directly
GameButton (uses useGame hook to get screen)
```

With Context, components that need the screen value can grab it directly using the `useGame` hook. Components that don't need it simply ignore it.

> 💡 **Context Eliminates Unnecessary Passing**
>
> **Context** is React's solution to prop drilling. It lets any component access shared data directly without passing it through every level. This keeps your code clean and makes it easy to add new components that need access to shared state. You'll use the `useGame` hook to access the current screen value from anywhere in your app — no prop drilling required.

🗺️ Adding Screen Navigation

🎯 **Goal:** Implement the core navigation system that will control which screen users see using shared state.

File: `src/App.jsx`

Step 1: Add imports to App

To access shared state and use screen constants, we need to import the `useGame` hook, `SCREENS` constants, and the `GameMap` component.

```
// Add these three imports
import { useGame } from './hooks/useGame';
import { SCREENS } from "./constants/screens";
import GameMap from "./components/GameMap";
// Existing import below
import SplashScreen from "./components/SplashScreen";
```

Step 2: Access shared state

Extract the `screen` value from Context using the `useGame` hook, which gives you access to the current screen state managed by `GameProvider`.

```
export default function App() {
  // Access shared screen state from Context
  const { screen } = useGame();

  // ... rest of function ...
}
```

Step 3: Add conditional rendering

Replace the hardcoded `<SplashScreen />` with conditional logic that renders different components based on the `screen` state value.

```
// Before:
<div className="app-container">
  <SplashScreen />
</div>

// After:
<div className="app-container">
  {screen === SCREENS.SPLASH && <SplashScreen />}
  {screen === SCREENS.PLAYING && <GameMap />}
</div>
```

Step 4: Test the setup

Run `npm run dev` if not already running.

✓ **You should see:** Your splash screen still appears normally. The navigation is ready, but we haven't wired up the button yet!

> 💡 **Shared State Controls Everything**
>
> The `&&` operator creates conditional rendering — when the left side is true, React renders the right side. By checking `screen` against different `SCREENS` constants, this single piece of shared state controls your entire app's display. Change the state in one place (like clicking a button), and the whole UI updates automatically. This is the power of centralized state management!

🔍 Using React DevTools for Exploring State

🎯 **Goal:** Use React DevTools to see how shared state works behind the scenes and experiment with changing it manually.

Step 1: Open DevTools and find GameProvider

1. **Press** `F12` or right-click → Inspect
2. **Find** the Components tab (next to Console, Network, etc.)
3. **Click** on GameProvider in the component tree

Step 2: Examine and modify state

1. **Look** for the hooks section showing the screen state value
2. **Enable** "Parse hook names" in the gear icon if hook names aren't clear
3. **Change** the screen value from "splash" to "playing" and watch the UI update!
4. **Change** it back to "splash" to see the SplashScreen return

✓ **You should see:** The screen instantly switches between SplashScreen and GameMap as you modify the state value!

> 💡 **Seeing and Changing State in Real-Time**
>
> React DevTools gives you X-ray vision into your app's **state**. You can see exactly what data each component has and even modify it in real-time. This is invaluable for debugging and understanding how **shared state** affects your entire app. Notice how changing one value in `GameProvider` instantly changes what component renders!

> 🏆 **Bonus Challenge**
>
> Try changing the screen state to different values and see what happens. What occurs when you set it to a value that doesn't match any of your conditions?

🎮 Implementing Start Game Function

🎯 **Goal:** Make your "Start Adventure" button actually start the game by updating the shared state.

File: `src/components/SplashScreen.jsx`

Step 1: Add imports to SplashScreen

To access screen constants and the shared state setter function, we need to import `SCREENS` and the `useGame` hook.

```
// Add these two imports
import { SCREENS } from "../constants/screens";
import { useGame } from "../hooks/useGame";
// Existing imports below
import { useState } from "react";
import CreditsModal from "./CreditsModal";
import GameButton from "./GameButton";
import GameLogo from "./GameLogo";
```

Step 2: Access the state setter

Extract the `setScreen` function from Context, which allows this component to update the shared screen state.

```
export default function SplashScreen() {
  const [showCredits, setShowCredits] = useState(false);
  // Add this line to access the shared state setter
  const { setScreen } = useGame();

  // ... rest of function ...
}
```

Step 3: Create the start game function

Define a function that changes the screen state to PLAYING when called, triggering the navigation to GameMap.

```
export default function SplashScreen() {
  const [showCredits, setShowCredits] = useState(false);
  const { setScreen } = useGame();

  // Add this function to handle game start
  const startGame = () => {
    setScreen(SCREENS.PLAYING);
  };

  // ... rest of function ...
}
```

Step 4: Update the Start Adventure button

Connect the button to your start game function by replacing the alert with the actual navigation handler.

```
// Before:
<GameButton
  text="Start Adventure"
  onClick={() => alert('Start Game!')}
  variant="primary"
/>

// After:
<GameButton
  text="Start Adventure"
  onClick={startGame}
  variant="primary"
/>
```

Step 5: Test the navigation

Click the “Start Adventure” button on your splash screen.

✓ **You should see:** The screen changes to GameMap! Your button now controls the entire app’s navigation through shared state.

> 💡 **One Change Updates the Whole App**
>
> When you call `setScreen(SCREENS.PLAYING)`, React updates the shared state in `GameProvider` and automatically re-renders all components that depend on that state. The `App` component sees the new screen value, evaluates its conditional rendering logic, and switches from `<SplashScreen />` to `<GameMap />`. This is the power of centralized state management — one function call orchestrates changes across your entire application.

📋 Essential Terms

Quick reference for all the state management concepts you just learned:

Term	Definition	Why it matters
🧠 state	Data that can change over time and causes components to re-render when it changes.	State lets components “remember” information and respond to user interactions dynamically.
🪝 hook	Functions starting with “use” that let you use React features like state and context.	Hooks like `useState` are your tools for managing data and behavior in components.
🔵 Context	React’s solution to prop drilling — lets components access shared data without passing props through multiple levels.	Context prevents “prop drilling” and provides shared state accessible from any component.

🔗 prop drilling	Passing data through multiple component levels, even when intermediate components don't need that data.	Context eliminates prop drilling by letting any component access shared data directly.
🔄 useState	A React hook that adds local state to functional components.	`useState` gives individual components their own memory for data that only they need to track.
📋 constants	Static values that don't change, used to prevent typos and make code more maintainable.	Constants like `SCREENS.SPLASH` prevent typos and make refactoring easier.
🔀 conditional rendering	Showing different components based on state or props using JavaScript expressions.	Conditional rendering with `&&` lets you control what users see based on app state.
🏪 Provider	A Context component that makes shared state available to all child components.	The Provider pattern wraps your app and gives all components access to shared data.

🤖 Ask the AI — Managing Game Flow

You just implemented both local and shared state, created screen navigation, and experienced the power of React's Context API — excellent work!

Now let's deepen your understanding of state management, hooks, and the React data flow. Here are the most impactful questions to ask your AI assistant about today's session:

- **What makes hooks special and why do they all start with "use"?**

- Explain `const [showCredits, setShowCredits] = useState(false);` in regular English.
- Explain state setter functions like `setScreen`, but in a non-tech example.
- What is “prop drilling” and how does the Context API prevent it? Give me non-tech examples.
- How does the GameProvider make state available to all components?
- Why use constants like `SCREENS.SPLASH` instead of just typing `"splash"` directly?