Logicblocks by Techligence Robotics Task Sheet
Integrating Blocks

January 1, 2024

Important All the **underlined** text are **links**, which might not be visible based on your application, so do not forget to click any underlined text throughout the document and make sure to follow the coding standards given in the file.

1 Redux Summary

Redux is a predictable state container for JavaScript applications, commonly used with libraries like React. It helps manage and centralize the application state. Here's a quick summary of the core concepts:

Action, Reducers and Store

Actions are plain JavaScript objects that describe "what happened" in the application. They have a payload property or other fields to carry data necessary for the action. For example

```
export const motionSlice = createSlice({
name: "Motion",
initialState,
reducers: {
    moveSteps: {
        reducer: (state, action) => {
           const { rightSteps, upSteps } = action.payload;
            const angleInRadians = (state.angle * Math.PI) / 180;
            const newX = state.position.x + rightSteps * Math.cos(angleInRadians) -
            upSteps * Math.sin(angleInRadians);
            const newY = state.position.y + rightSteps * Math.sin(angleInRadians) +
            upSteps * Math.cos(angleInRadians);
            state.position.x = newX;
            state.position.y = newY;
        },
        prepare: (rightSteps, upSteps) => ({ payload: { rightSteps, upSteps } })
    },
    ..........
```

Reducers are pure functions that take the current state and an action as arguments and return a new state. They describe "how the state changes" in response to an action. The key point is that reducers must be pure i.e. they should not mutate the current state but return a new object if the state changes.

Store is an object that brings actions and reducers together. The store has several responsibilities:

1. Holds the application state;
2. Allows access to state via `getState()`;
3. Allows state to be updated via `dispatch(action)`;
4. Registers listeners via `subscribe(listener)`;
5. Handles unregistering of listeners via the function returned by `subscribe(listener)`.

Workflow

Here's how they work together:

1. **Store Creation:** You create a Redux store using `createStore()` and passing in your root reducer.
2. **Dispatching Actions:** Your application dispatches an action to express an intention to change the state.
3. **Handling Actions with Reducers:** The store runs the reducer function with the current state and the given action. The reducer returns a new state.
4. **Updating the Store:** The Redux store saves the new state returned by the reducer.
5. **Reacting to State Changes:** The application's UI is updated to reflect the new state. If you're using React, this is typically done via a library like `react-redux` that subscribes to the store and causes your components to re-render with the new state.

2 Redux Logic for the Application

In our application, user interactions are facilitated through a visual programming interface where blocks are connected to define the behavior of sprites on the canvas. Given the dynamic nature of these interactions, maintaining an accurate and up-to-date representation of each sprite's state is essential for the correct operation of the application. Redux serves as the state management framework, ensuring a consistent and predictable state across all components.

State Representation

Each sprite's state is represented as an object within the Redux store, encapsulating properties such as position, visibility, and other relevant attributes that can be altered by block-based actions.

Actions and Reducers

Actions in our Redux implementation are dispatched when a block is activated by the user, representing a clear intention to transform the state of a sprite. For instance, moving a sprite or changing its costume would dispatch corresponding actions like `MOVE_SPRITE` or `CHANGE_COSTUME`, respectively.

Reducers respond to these action dispatches, taking the current state and the action performed to produce a new state. It's critical that reducers remain pure functions—any side effects or asynchronous operations must be handled outside the reducers, typically in middleware like Redux Thunk.

Store

The store is the central hub where the state lives. It is created using the `createStore()` function provided by Redux, which is enhanced with middleware **https://redux.js.org/usage/writing-logic-thunks** to handle more complex scenarios such as asynchronous actions or logging.

Tracking Changes

Changes to the state are tracked through listeners that are registered via `subscribe()`. These listeners allow the application to react in real-time as the state changes, ensuring that the UI is always synchronized with the current state of the sprites.

Unidirectional Data Flow

Our application adheres to a unidirectional data flow pattern. This means that all state changes are funneled through a strict series of steps—actions are dispatched, reducers process these actions, and the store updates the state. This pattern facilitates easier debugging and testing, as well as more predictable data management.

3 Does Your Category Need Redux?

Deciding whether to use Redux for state management in a particular category of LogicBlocks depends on various factors. Redux excels in scenarios where the state is complex, where operations on the state are non-trivial, and where different parts of the application need to stay synchronized with the state.

3.1 Motion

The "Motion" category typically involves changing the position, rotation, and other physical attributes of sprites and as a result changes the state for most of the motion blocks, so it is necessary to keep track of the state and any changes that happens to the state. So applying redux logic would be relevant here.

3.2 Control

The "Control" category contains a variety of blocks from loops to clones.

Loops

For loop constructs, such as "repeat" or "forever", incorporating Redux for state management may not be inherently necessary if they only affect local component state. Loops typically execute a set of actions repeatedly and do not, by themselves, result in state changes that need to be reflected globally across the application. However, if the loop's execution or termination has implications on the application level — for instance, if other components need to react to the start or end of a loop — then dispatching Redux actions for these events could be advantageous.

Clones

On the other hand, blocks like "create clone of myself" have a broader impact on the application state. Cloning operations introduce new entities that the application must track. In such cases, using Redux is beneficial to manage the state of each entity. Redux allows for a centralized state management approach, where each clone's state is maintained within the global store. This is crucial for ensuring that actions affecting clones are reflected throughout the application, enabling consistent state across components.

For example, when a clone is created, an action is dispatched to update the Redux store with the new entity's state. Similarly, when a clone is deleted, an action is dispatched to remove the entity's state from the store. By doing so, we maintain a single source of truth for the state of all entities, which simplifies the state management and provides clear paths for debugging and state updates.

So to conclude, for the loops implementing the Redux logic would not make any sense as there is no corresponding state change. However, for the clone blocks, since we are creating separate entities, it is therefore necessary to keep track of their corresponding states.

3.3 Looks

Say and Think Blocks

Redux is typically not necessary unless the application needs to keep a history of what sprites have said or thought, or if these messages trigger changes in other parts of the application.

Costume and Backdrop Blocks

If other components need to react to changes in a sprite's costume or the scene's backdrop, or if these changes affect game logic (such as different interactions depending on the costume), then Redux can be used to manage these state changes globally.

Size and Color Effect Blocks

Redux becomes important if the size or visual effects of a sprite have implications beyond the local component—such as affecting collisions in a game or enabling and disabling certain interactions based on visual states.

Show and Hide Blocks

Visibility changes might require Redux when the visibility status of sprites needs to be referenced or controlled by other parts of the application, ensuring that all components have a consistent view of each sprite's visibility.

Layer Blocks

Layering is often critical in applications with complex UIs where the depth ordering of sprites affects gameplay or interactions. Redux is useful for managing layer order when it's important to maintain consistency across the application, or when layer changes should trigger other state updates.

3.4 Sound

Managing sound in an application can often benefit from centralized state management, especially in complex scenarios where different parts of the application need to respond to or control the audio state.

Playing and Stopping Sounds:

While playing a sound might not require Redux, stopping all sounds or coordinating sound effects across multiple components could benefit from Redux to ensure consistent behavior across the application.

Pitch Effects:

If the pitch effect needs to persist or be altered from various places within the application, using Redux to manage this state globally allows for a synchronized effect.

Volume Control:

Adjusting the volume might be local to a component. However, if there's a need for a global mute feature or consistent volume levels across different scenes or components, Redux can be instrumental.

Sound Effects:

Clearing sound effects, if it affects multiple sounds or needs to be triggered from different application areas, might necessitate Redux to coordinate these actions centrally.

Redux facilitates the management of sound in an application by providing a predictable state container that can be accessed and modified. This is important as sound plays a crucial role in the user experience.

Small task for sounds category

Include the following sound as the default sound and should be integrated in the application. The sound file is made available in the main branch **at the following location**

3.5 Events

Event handling is central to interactive applications. It dictates how the application responds to user inputs and system-generated events. While individual event handlers might not require global state management, there are scenarios where Redux can play a crucial role.

User Interactions:

For events like "when clicked" or "when key pressed," if the response involves updating multiple parts of the application or if the events trigger state changes that are not local to the component, Redux can be used to manage these interactions.

Sprite and Backdrop Changes:

Events such as "when sprite clicked" or "when backdrop switches" could necessitate Redux when these actions need to trigger state changes across the application.

Sound and Loudness Detection:

For "when loudness" events, Redux can help in managing the state in scenarios where the application needs to react globally to the ambient sound level, such as triggering accessibility features or adjusting game mechanics.

Messaging System:

The "when I receive" and "broadcast" blocks are especially suited for Redux, as they often imply a need for different parts of the application to respond to a common signal. Redux provides a structured way to handle these broadcast messages and ensure that all parts of the application react consistently.

Redux enhances the event-driven nature of an application by ensuring that all responses to events are coordinated through a central state management system. This is particularly advantageous here as events have wide-ranging impacts on the application's state and behavior.

3.6 Operators

Operator blocks, encompassing arithmetic, random number generation, comparison, logical, string manipulation, and other mathematical operations, are fundamental for scripting logical flow within an application. These blocks typically execute operations that are local to the context in which they are used and, as such, do not generally necessitate global state management via Redux.

3.7 Variables

Creating and Setting Variables:

Variables are often used to keep track of data that may change over time. While many variable operations are local to specific functions or components, Redux can be employed to manage state when variables are shared across multiple parts of the application or when their values need to persist beyond the lifecycle of a single component or page.

Changing Variables:

If the modification of a variable triggers other state changes within the application, or if the new value needs to be reflected across different components that are not directly related, Redux provides a predictable state management system to handle these changes.

Showing and Hiding Variables:

The visibility of variables may be a simple UI concern that does not require global state management. However, if the application needs to conditionally display variables based on global state or actions taken elsewhere in the application, Redux can be used to control their visibility.

4 Generated Code

As many of you have already learned the process of translating your blocks into code, there remains a slight oversight. The issue lies in the fact that the generated code is composed solely of function calls, not containing the essential function definitions. Without these definitions, the code cannot be compiled.
In the absence of these definitions, the function calls remains unanswered, and results in an error.
Therefore, your task is twofold:

1. **Ensure Function Definitions:** Begin by constructing the function definitions that correspond to each call. These definitions should encapsulate the logic that the blocks represent, translating the visual instructions into syntactic commands that the computer can comprehend.
2. **Integrate Definitions with Calls:** Once defined, these functions must be integrated into your codebase. Place the definitions so that they are accessible to the function calls—typically, this would mean defining functions at a scope that is at or above the level where the calls are made.
3. **Display on Screen:** Finally, ensure that the full code—including both definitions and calls—is rendered on the screen. This not only serves to verify the completeness of your code but also provides a tool for understanding the relationship between the blocks and the underlying code they generate.

For example
Instead of just the following

```
sayHello();
```

The output should look something like:

```
// Function definition
function sayHello() {
  console.log('Hello, World!');
}

// Function call
sayHello();
```

5 Block Execution

The remaining task is to initiate the execution of blocks and observe the resultant changes on the sprite. There are primarily two methods to execute the blocks: 1. Event-driven blocks, and 2. Direct interaction through clicking. Assuming event-driven execution has already been established, our focus will now shift to the latter method, which involves direct user interaction.

To facilitate execution via clicking, the blocks should be responsive to mouse clicks. This interactivity implies that upon the click event, the associated reducer function is invoked, thereby acting upon the current state and yielding an updated state reflective of the change commanded by the block's functionality.

Implementation for Single Blocks

For individual blocks, the implementation is straightforward. Each block is equipped with an event listener that detects clicks. Upon activation, the listener dispatches an action, which the reducer processes to update the state. This updated state then triggers a re-render of the sprite, manifesting the changes specified by the block.

```
// Example in JavaScript/React
<BlockComponent onClick={() => dispatch(actionForBlock())} />
```

Sequential Execution for Connected Blocks

When dealing with a sequence of connected blocks, the execution logic needs to account for the chain reaction wherein one block's execution potentially triggers the next. To achieve this, a clicking mechanism should be established not just for the initial block but for the entire chain. The system must identify the connection points between blocks and ensure the sequential execution follows the established order.

State Management Considerations

It is crucial for the state management logic to accommodate these interactions seamlessly. The state must be structured in such a way that it facilitates not only the current state's transformation but also maintains the integrity and continuity of the execution flow. This includes handling any asynchronous actions or side effects that may arise during the blocks' execution.

6 Important

This task should be handed in by **05.01.2024's team meeting, without fail.**