React TS Keyboard Control

how to control keyboard with react ts example small game with full example

2/2

Creating a simple game in React TypeScript using keyboard controls is a fun way to get familiar with handling keyboard events. Below is a small example of a basic game where you control a square that moves around on the screen using the arrow keys.

This example includes:

1. Setting up keyboard event listeners for arrow key movements.
2. Updating the position of the player.
3. Rendering the player on the screen.

Let's get started!

1. Setup

Install dependencies (if not already set up):

```bash
npx create-react-app keyboard-game --template typescript
cd keyboard-game
```

2. Code Example

`App.tsx`

Here's the main file with the full code:

```typescript
import React, { useState, useEffect } from 'react';

type Position = {
  x: number;
  y: number;
};
```

```
const App: React.FC = () => {
  // Set initial position
  const [position, setPosition] = useState<Position>({ x: 50, y: 50 });

  // Key press handler
  const handleKeyPress = (event: KeyboardEvent) => {
    setPosition((prevPosition) => {
      switch (event.key) {
        case 'ArrowUp':
          return { ...prevPosition, y: Math.max(prevPosition.y - 10, 0) };
        case 'ArrowDown':
          return { ...prevPosition, y: Math.min(prevPosition.y + 10,
window.innerHeight - 50) };
        case 'ArrowLeft':
          return { ...prevPosition, x: Math.max(prevPosition.x - 10, 0) };
        case 'ArrowRight':
          return { ...prevPosition, x: Math.min(prevPosition.x + 10,
window.innerWidth - 50) };
        default:
          return prevPosition;
      }
    });
  };

  // Attach event listener
  useEffect(() => {
    window.addEventListener('keydown', handleKeyPress);
    return () => {
      window.removeEventListener('keydown', handleKeyPress);
    };
  }, []);

  return (
    <div
      style={{
        width: '100vw',
        height: '100vh',
        display: 'flex',
        alignItems: 'center',
        justifyContent: 'center',
        overflow: 'hidden',
      }}
```

```
    >
      <div
        style={{
          position: 'absolute',
          top: position.y,
          left: position.x,
          width: '50px',
          height: '50px',
          backgroundColor: 'dodgerblue',
        }}
      />
    </div>
  );
};

export default App;
```

Explanation

1. **State**: We maintain a `position` state to track the square's `x` and `y` coordinates.
2. **Key Handling**: The `handleKeyPress` function handles arrow key events. Based on the key pressed, we update the `x` or `y` coordinate.
3. **Event Listener**: We add a `keydown` event listener on `window` inside `useEffect` to detect arrow key presses and remove it on component unmount.
4. **Rendering**: The square is positioned using absolute positioning, and the position updates based on arrow key presses.

3. Run the Game

Start the app:

```bash
npm start
```

Now, you should see a blue square that you can move around the screen using the arrow keys!

how to control keyboard with react vite ts example small game with full example

Let's build a small game in React with TypeScript, using Vite as our build tool. In this example, we'll create a simple game where you can control a square on the screen using the arrow keys.

1. Set Up Vite with React and TypeScript

First, create a new Vite project and install dependencies:

```bash
npm create vite@latest my-game --template react-ts
cd my-game
npm install
```

2. Code the Game Logic

Open `src/App.tsx` and replace its content with the following code:

`App.tsx`

```typescript
import React, { useState, useEffect } from 'react';

type Position = {
  x: number;
  y: number;
};

const App: React.FC = () => {
  // Initial position state for the player (a square)
  const [position, setPosition] = useState<Position>({ x: 50, y: 50 });

  // Function to handle key press events
  const handleKeyPress = (event: KeyboardEvent) => {
    setPosition((prevPosition) => {
      switch (event.key) {
        case 'ArrowUp':
          return { ...prevPosition, y: Math.max(prevPosition.y - 10, 0) };
        case 'ArrowDown':
          return { ...prevPosition, y: Math.min(prevPosition.y + 10,
window.innerHeight - 50) };
```

```
        case 'ArrowLeft':
          return { ...prevPosition, x: Math.max(prevPosition.x - 10, 0) };
        case 'ArrowRight':
          return { ...prevPosition, x: Math.min(prevPosition.x + 10,
window.innerWidth - 50) };
        default:
          return prevPosition;
      }
    });
  };

  // Set up the event listener for keyboard input
  useEffect(() => {
    window.addEventListener('keydown', handleKeyPress);
    return () => {
      window.removeEventListener('keydown', handleKeyPress);
    };
  }, []);

  return (
    <div
      style={{
        width: '100vw',
        height: '100vh',
        display: 'flex',
        alignItems: 'center',
        justifyContent: 'center',
        overflow: 'hidden',
        backgroundColor: '#f0f0f0',
      }}
    >
      <div
        style={{
          position: 'absolute',
          top: position.y,
          left: position.x,
          width: '50px',
          height: '50px',
          backgroundColor: 'dodgerblue',
        }}
      />
    </div>
```

```
  );
};

export default App;
```

Explanation

1. **State**: We use a `position` state to track the x and y coordinates of the player (square).
2. **Keyboard Handler**: The `handleKeyPress` function listens for arrow key presses and updates the `position` accordingly:
   - `ArrowUp` and `ArrowDown` adjust the y-coordinate.
   - `ArrowLeft` and `ArrowRight` adjust the x-coordinate.
   - The position is constrained to keep the square within the window boundaries.
3. **Event Listener**: We use `useEffect` to add and remove the `keydown` event listener on component mount and unmount, respectively.
4. **Rendering**: The square is rendered at the current `position` using inline CSS for absolute positioning.

3. Start the Game

Now you can run the game:

```bash
npm run dev
```

When you open the app in your browser, you should see a blue square. Use the arrow keys to move it around the screen!

how to control keyboard with react vite ts example small game with full example with object and not switch

Using an object instead of a switch statement can make the code more concise. Here's how to structure the example to handle key press events with an object-based approach.

Steps

We'll:

1. Define an object to map key names to position changes.
2. Use this object in our keyboard event handler instead of a switch statement.

Full Code Example

`App.tsx`

Replace the existing code in `App.tsx` with the following:

```typescript
import React, { useState, useEffect } from 'react';

type Position = {
  x: number;
  y: number;
};

// Define the movement mapping outside of the component
const movementMap: { [key: string]: Position } = {
  ArrowUp: { x: 0, y: -10 },
  ArrowDown: { x: 0, y: 10 },
  ArrowLeft: { x: -10, y: 0 },
  ArrowRight: { x: 10, y: 0 },
};

const App: React.FC = () => {
  // Initial position state for the player (a square)
  const [position, setPosition] = useState<Position>({ x: 50, y: 50 });

  // Function to handle key press events using the movementMap
  const handleKeyPress = (event: KeyboardEvent) => {
    const movement = movementMap[event.key];
    if (movement) {
      setPosition((prevPosition) => ({
        x: Math.min(
          Math.max(prevPosition.x + movement.x, 0),
          window.innerWidth - 50
        ),
        y: Math.min(
          Math.max(prevPosition.y + movement.y, 0),
          window.innerHeight - 50
        ),
```

```
      }));
    }
  };

  // Set up the event listener for keyboard input
  useEffect(() => {
    window.addEventListener('keydown', handleKeyPress);
    return () => {
      window.removeEventListener('keydown', handleKeyPress);
    };
  }, []);

  return (
    <div
      style={{
        width: '100vw',
        height: '100vh',
        display: 'flex',
        alignItems: 'center',
        justifyContent: 'center',
        overflow: 'hidden',
        backgroundColor: '#f0f0f0',
      }}
    >
      <div
        style={{
          position: 'absolute',
          top: position.y,
          left: position.x,
          width: '50px',
          height: '50px',
          backgroundColor: 'dodgerblue',
        }}
      />
    </div>
  );
};

export default App;
```

Explanation

1. **Movement Map**: We define `movementMap`, an object that maps each arrow key to an x and y position change. This allows us to avoid using a `switch` statement by directly referencing `movementMap[event.key]`.
2. **Position Update**: Inside `handleKeyPress`, we check if the `event.key` exists in `movementMap`. If it does, we retrieve the corresponding position change values and use them to update the player's position, keeping the square within the window boundaries.
3. **Event Listener**: We set up and clean up the `keydown` listener with `useEffect`.

Run the Game

Start the development server:

```bash
npm run dev
```

You can now control the square with the arrow keys, using an object-based approach for the key mappings!