Joint Control System

Event-Driven Architecture Documentation

Real-time Robot Joint Control

High-Performance Event-Based System

June 22, 2025

Contents

1 Introduction

The Joint Control System is a high-performance, event-driven architecture designed for real-time robot joint manipulation. It provides smooth, responsive control of robot joints through an efficient event-based communication system that separates high-frequency updates from UI rendering.

1.1 System Overview

The Joint Control System enables:

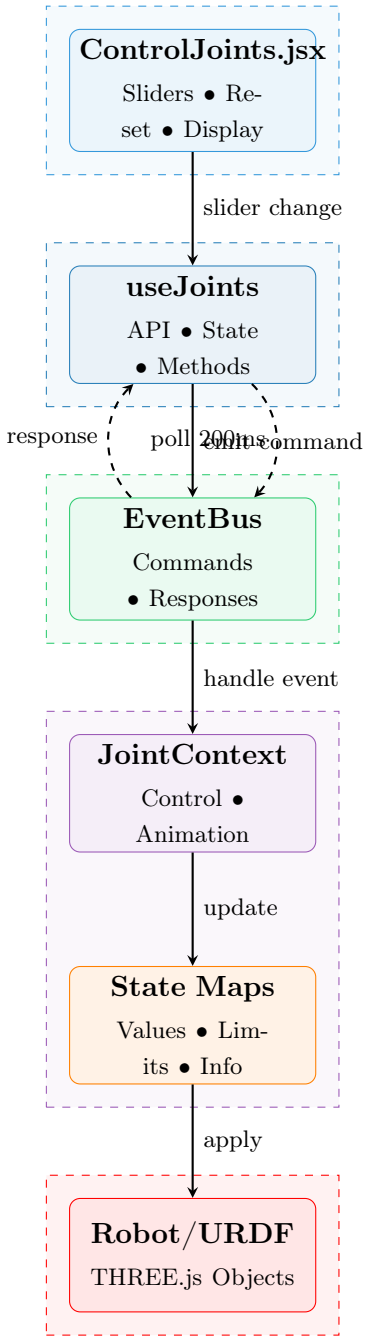
- Real-time joint angle control at 60Hz
- Smooth joint animation with multiple easing functions
- Joint limit enforcement and validation
- Multi-robot joint management
- Event-based command/response pattern
- Decoupled UI updates from robot control

1.2 Key Design Principles

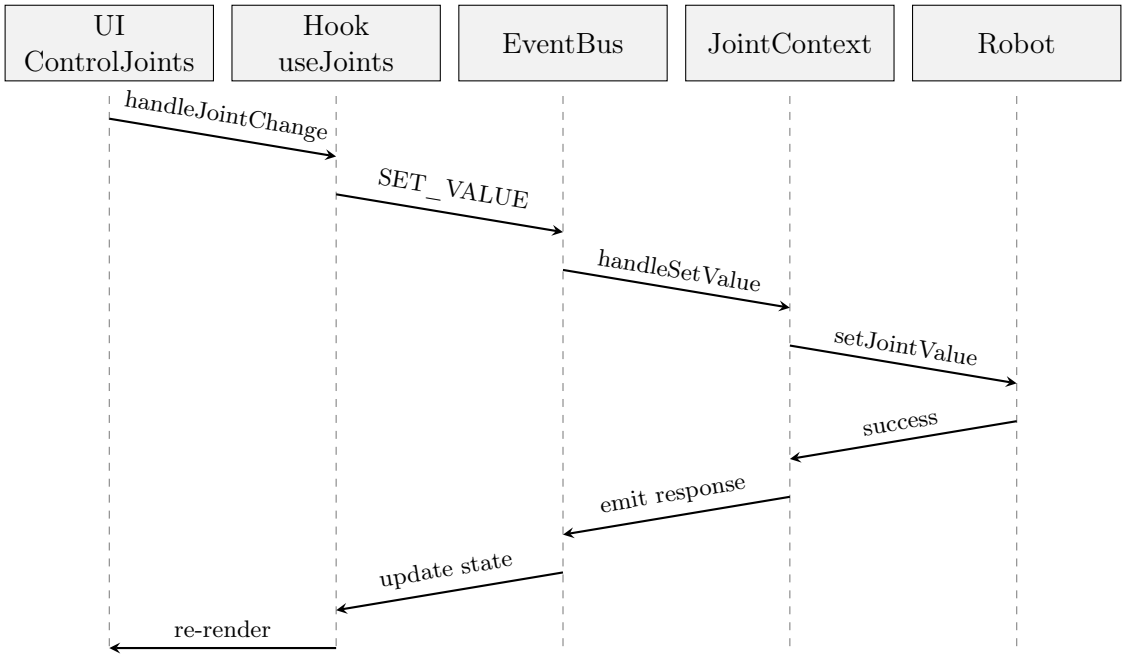
1. **Event-Driven**: All joint commands go through EventBus
2. **Performance First**: UI shows commanded values, not actual
3. **Separation of Concerns**: UI → Hook → Events → Context
4. **Real-time Updates**: 200ms polling for joint values
5. **Type Safety**: Command/Response pattern with request IDs

2 Architecture Overview

2.1 System Flow Diagram



2.2 Event Flow Sequence



3 Component Architecture

3.1 UI Component: ControlJoints

The UI component provides the interface for joint control:

```
import React, { useCallback } from 'react';
import useJoints from '../../../contexts/hooks/useJoints';

const ControlJoints = () => {
  // Get all joint functionality from single hook
  const joints = useJoints();

  const {
    robotId,
    setJointValue,
    resetJoints,
    getJointLimits,
    getJointValue,
    hasJoints,
    hasMovableJoints,
    getMovableJoints,
    debugJoint
  } = joints;

  // Handle joint change
  const handleJointChange = useCallback((jointName, value) => {
    const numValue = parseFloat(value);
    const success = setJointValue(jointName, numValue);

    if (!success) {
```

```
      debugJoint(`Failed to update joint ${jointName}`);
    }
  }, [setJointValue, debugJoint]);

  // Get movable joints for display
  const movableJoints = getMovableJoints();

  return (
    <div className="controls-section">
      <h3>Joint Control - {robotId}</h3>
      {movableJoints.map((joint) => (
        <JointSlider
          key={joint.name}
          joint={joint}
          value={getJointValue(joint.name)}
          limits={getJointLimits(joint.name)}
          onChange={(value) => handleJointChange(joint.name, value)}
        />
      ))}
      <button onClick={resetJoints}>Reset All Joints</button>
    </div>
  );
};
```

Listing 1: ControlJoints Component Structure

3.2 Hook Layer: useJoints

The hook provides a clean API and manages polling:

```
export const useJoints = (robotIdOverride = null) => {
  const jointContext = useJointContext();
  const { activeId: contextRobotId } = useRobotSelection();
  const robotIdToUse = robotIdOverride || contextRobotId;

  // State to track commanded joint values
  const [commandedValues, setCommandedValues] = useState({});
  const [polledJointValues, setPolledJointValues] = useState({});

  // Poll for joint values via GET_VALUES event every 200ms
  useEffect(() => {
    if (!robotIdToUse) return;
    let isMounted = true;
    let interval;
    let requestId = 'getvals_' + Date.now();

    const handleResponse = (data) => {
      if (isMounted && data.robotId === robotIdToUse &&
          data.requestId === requestId) {
        setPolledJointValues(data.values);
        // Update commanded values if not set
        setCommandedValues(prev => {
          if (Object.keys(prev).length === 0) {
            return data.values;
          }
          return prev;
```

```
        });
      }
    };

    const unsub = EventBus.on(JointEvents.Responses.GET_VALUES,
                              handleResponse);

    // Poll every 200ms
    interval = setInterval(() => {
      requestId = 'getvals_' + Date.now();
      EventBus.emit(JointEvents.Commands.GET_VALUES,
                    { robotId: robotIdToUse, requestId });
    }, 200);

    // Initial fetch
    EventBus.emit(JointEvents.Commands.GET_VALUES,
                  { robotId: robotIdToUse, requestId });

    return () => {
      isMounted = false;
      clearInterval(interval);
      unsub();
    };
  }, [robotIdToUse]);

  // Event-driven joint operations
  const setJointValue = useCallback((jointName, value) => {
    if (!robotIdToUse) return false;
    EventBus.emit(JointEvents.Commands.SET_VALUE,
                  { robotId: robotIdToUse, jointName, value });
    return true;
  }, [robotIdToUse]);

  return {
    robotId: robotIdToUse,
    jointValues: commandedValues,
    setJointValue,
    resetJoints,
    getJointValue,
    // ... other methods
  };
};
```

Listing 2: useJoints Hook with Polling

4 Event System

4.1 Joint Events Definition

```
export const JointEvents = {
  // ========== Commands ==========
  Commands: {
    /**
```

```
     * Set single joint value
     * PAYLOAD: {
     *   robotId: string,
     *   jointName: string,
     *   value: number,
     *   requestId?: string
     * }
     */
    SET_VALUE: 'joint:command:set-value',

    /**
     * Set multiple joint values
     * PAYLOAD: {
     *   robotId: string,
     *   values: Object,      // { jointName: value, ... }
     *   requestId?: string
     * }
     */
    SET_VALUES: 'joint:command:set-values',

    /**
     * Get current joint values
     * PAYLOAD: {
     *   robotId: string,
     *   requestId: string    // Required
     * }
     */
    GET_VALUES: 'joint:command:get-values',

    /**
     * Reset all joints to zero
     * PAYLOAD: {
     *   robotId: string,
     *   requestId?: string
     * }
     */
    RESET: 'joint:command:reset'
  },

  // ========== Responses ==========
  Responses: {
    SET_VALUE: 'joint:response:set-value',
    SET_VALUES: 'joint:response:set-values',
    GET_VALUES: 'joint:response:get-values',
    RESET: 'joint:response:reset'
  }
};
```

Listing 3: Joint Events in dataTransfer.js

4.2 Event Handling in JointContext

```
// Handle command events
useEffect(() => {
  const handleSetValue = (data) => {
```

```
    const { robotId, jointName, value, requestId } = data;

    // Find robot
    const robot = findRobotWithFallbacks(robotId);
    if (!robot) {
      EventBus.emit(JointEvents.Responses.SET_VALUE, {
        robotId, jointName, value,
        success: false, requestId
      });
      return;
    }

    // Apply joint value
    const success = applyJointValue(robot, jointName, value);

    // Update state
    if (success) {
      setRobotJointValues(prev => {
        const newMap = new Map(prev);
        const current = newMap.get(robotId) || {};
        newMap.set(robotId, { ...current, [jointName]: value });
        return newMap;
      });
    }

    // Send response
    EventBus.emit(JointEvents.Responses.SET_VALUE, {
      robotId, jointName, value,
      success, requestId
    });
  };

  const handleGetValues = (data) => {
    const { robotId, requestId } = data;
    const values = robotJointValues.get(robotId) || {};

    EventBus.emit(JointEvents.Responses.GET_VALUES, {
      robotId, values, requestId
    });
  };

  const unsubSet = EventBus.on(JointEvents.Commands.SET_VALUE,
                               handleSetValue);
  const unsubGet = EventBus.on(JointEvents.Commands.GET_VALUES,
                               handleGetValues);

  return () => {
    unsubSet();
    unsubGet();
  };
}, [robotJointValues]);
```

Listing 4: JointContext Event Handlers

5 Performance Optimization

5.1 Commanded vs Actual Values

The system displays commanded values for immediate UI feedback:

Value Type	Description	Update Method	Frequency
Commanded	Target joint position	Direct on change	Immediate
Actual	Current robot position	Polling via events	200ms
Display	Shown in UI	Commanded preferred	Real-time

Table 1: Joint Value Types

5.2 Animation System

Joint animations use requestAnimationFrame for smooth motion:

```
const animateToJointValues = useCallback(async (robotId, targetValues
    , options = {}) => {
  const {
    duration = 1000,
    easing = 'exponential',
    onProgress,
    onComplete
  } = options;

  const robot = findRobotWithFallbacks(robotId);
  if (!robot) return { success: false };

  return new Promise((resolve) => {
    const currentValues = getRobotJointValues(robotId);
    const startTime = Date.now();
    const animationId = `anim_${robotId}_${startTime}`;

    // Store animation state
    animationStatesRef.current.set(robotId, {
      id: animationId,
      active: true,
      startTime,
      targetValues
    });

    const animate = () => {
      const animState = animationStatesRef.current.get(robotId);
      if (!animState || animState.id !== animationId) {
        return resolve({ success: false, reason: 'cancelled' });
      }

      const elapsed = Date.now() - startTime;
      const progress = Math.min(elapsed / duration, 1);

      // Apply easing
      const easedProgress = easingFunctions[easing](progress);
```

```
36
37        // Interpolate joint values
38        const interpolatedValues = {};
39        Object.keys(targetValues).forEach(jointName => {
40          const start = currentValues[jointName] || 0;
41          const end = targetValues[jointName];
42          interpolatedValues[jointName] =
43            start + (end - start) * easedProgress;
44        });
45
46        // Apply to robot
47        setRobotJointValues_Internal(robotId, interpolatedValues);
48
49        // Emit progress event
50        if (onProgress) {
51          onProgress(progress, interpolatedValues);
52        }
53
54        // Check completion
55        if (progress >= 1) {
56          animationStatesRef.current.delete(robotId);
57          if (onComplete) onComplete();
58          resolve({ success: true });
59        } else {
60          requestAnimationFrame(animate);
61        }
62      };
63
64      requestAnimationFrame(animate);
65    });
66  }, []);
```

Listing 5: Joint Animation Implementation

6 State Management

6.1 JointContext State Structure

```
1  const JointProvider = ({ children }) => {
2    // Robot joint information (structure, limits, types)
3    const [robotJoints, setRobotJoints] = useState(new Map());
4
5    // Current joint values for each robot
6    const [robotJointValues, setRobotJointValues] = useState(new Map())
       ;
7
8    // Animation states
9    const [isAnimating, setIsAnimating] = useState(new Map());
10
11   // Local robot registry for fast lookups
12   const robotRegistryRef = useRef(new Map());
13   const robotManagerRef = useRef(null);
14   const animationStatesRef = useRef(new Map());
```

```
15
16    // Example state structure:
17    // robotJoints.get('ur5_001') = [
18    //   { name: 'shoulder_pan_joint', type: 'revolute',
19    //     limits: { lower: -6.28, upper: 6.28 } },
20    //   { name: 'shoulder_lift_joint', type: 'revolute',
21    //     limits: { lower: -3.14, upper: 3.14 } },
22    //   ...
23    // ]
24
25    // robotJointValues.get('ur5_001') = {
26    //   'shoulder_pan_joint': 0.523,
27    //   'shoulder_lift_joint': -1.047,
28    //   ...
29    // }
30 };
```

Listing 6: Joint State Management

7 Best Practices

7.1 Event-Driven Guidelines

1. **Always Use Events**: Never directly manipulate joints from UI
2. **Request IDs**: Include for request/response correlation
3. **Error Handling**: Always send response, even on failure
4. **Cleanup**: Unsubscribe from events on unmount
5. **Debouncing**: Consider for high-frequency slider updates

7.2 Performance Guidelines

1. **Polling Interval**: 200ms balances responsiveness and performance
2. **Commanded Values**: Show immediately for responsive UI
3. **Batch Updates**: Use SET_VALUES for multiple joints
4. **Animation Frame**: Use for smooth visual updates
5. **Memoization**: Cache computed values in hooks

8 Integration Examples

8.1 Complete Joint Control Flow

```
// 1. User Component
const RobotJointPanel = () => {
  const {
    movableJoints,
    setJointValue,
    getJointValue
  } = useJoints();

  return (
    <div>
      {movableJoints.map(joint => (
        <JointControl
          key={joint.name}
          name={joint.name}
          value={getJointValue(joint.name)}
          onChange={(value) => setJointValue(joint.name, value)}
        />
      ))}
    </div>
  );
};

// 2. Joint Control Component
const JointControl = ({ name, value, onChange }) => {
  return (
    <div className="joint-control">
      <label>{name}</label>
      <input
        type="range"
        value={value}
        onChange={(e) => onChange(parseFloat(e.target.value))}
        min={-Math.PI}
        max={Math.PI}
        step={0.01}
      />
      <span>{(value * 180 / Math.PI).toFixed(1)}  </span>
    </div>
  );
};

// 3. Usage in App
const App = () => {
  return (
    <ViewerProvider>
      <RobotProvider>
        <JointProvider>
          <div className="app">
            <Canvas3D />
            <RobotJointPanel />
          </div>
        </JointProvider>
      </RobotProvider>
    </ViewerProvider>
  );
};
```

Listing 7: Full Integration Example

9 Troubleshooting

9.1 Common Issues

Issue	Solution
Joints not updating	Check robot ID matches and robot is loaded
Delayed response	Verify EventBus subscriptions are active
Jumpy animations	Ensure only one animation per robot at a time
Missing joints	Confirm robot URDF has proper joint definitions

Table 2: Common Joint Control Issues