Robot Pose System

Reposition Architecture Documentation

Event-Based Robot Positioning

World Space Control System

June 22, 2025

Contents

1 Introduction

The Robot Pose System (Reposition) provides event-driven control over robot position and orientation in world space. It implements a clean separation between UI controls and the underlying THREE.js scene manipulation through a well-defined event architecture.

1.1 System Overview

The Robot Pose System enables:

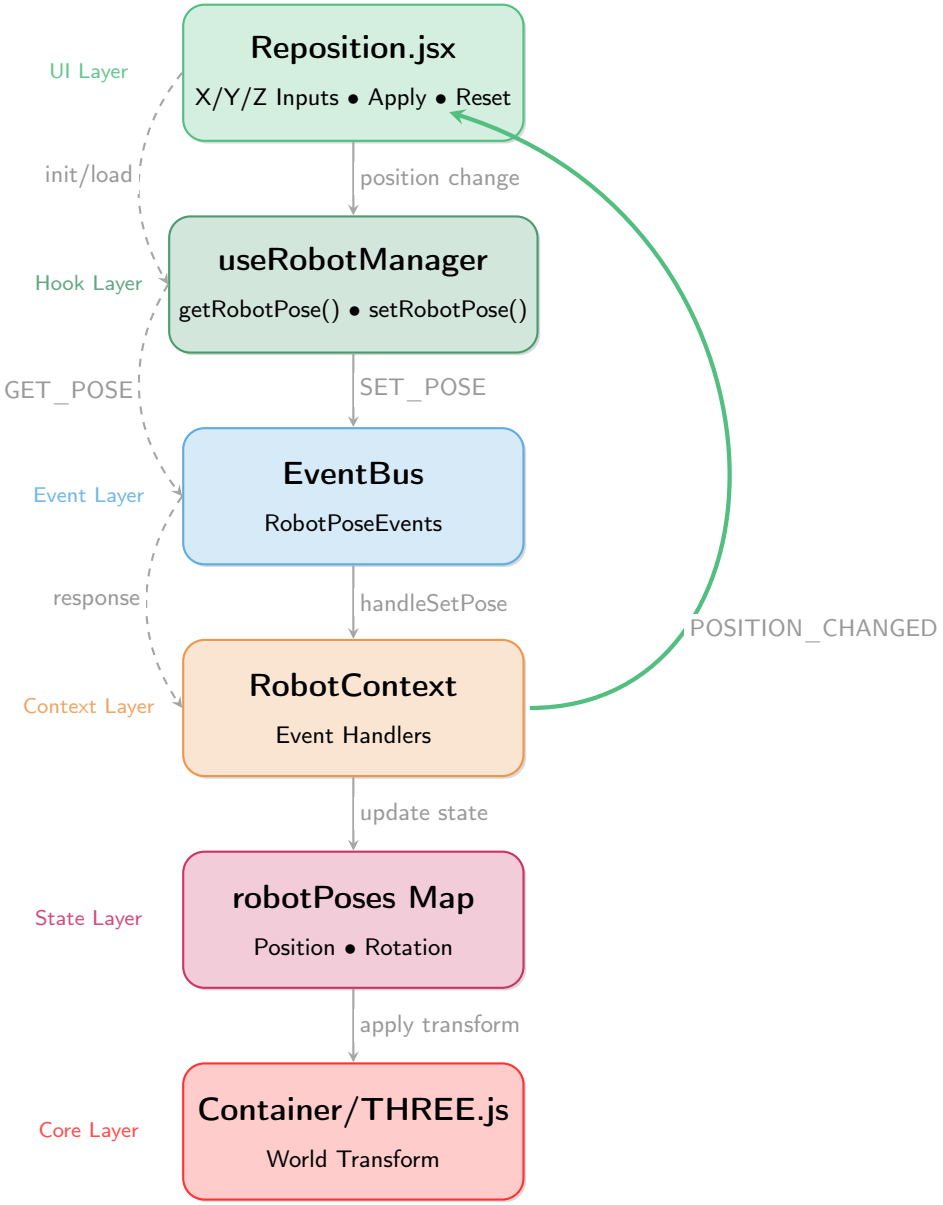
- Real-time robot position control in 3D space
- Event-based position updates
- Asynchronous pose queries with Promises
- Position persistence and state management
- Multi-robot position management
- Integration with camera focus system

1.2 Key Design Principles

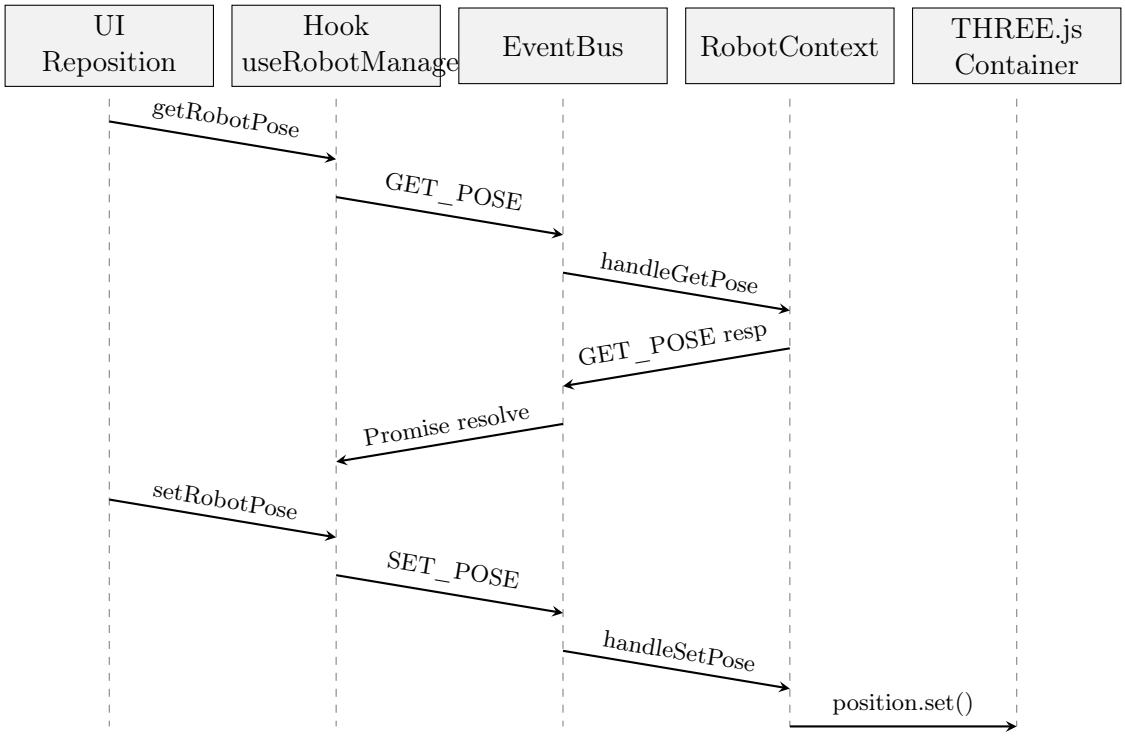
1. **Event-Driven**: All pose changes through EventBus
2. **Async-First**: Pose queries return Promises
3. **Decoupled**: UI never directly accesses robot containers
4. **State Tracking**: Positions tracked in context state
5. **Container-Based**: Robots wrapped in position containers

2 Architecture Overview

2.1 System Flow Diagram



2.2 Event Flow Sequence



3 Component Architecture

3.1 UI Component: Reposition

The Reposition component provides position control interface:

```
import React, { useState, useEffect, useCallback } from 'react';
import { useRobotManager, useRobotSelection } from '../../../contexts
    /hooks/useRobotManager';

const Reposition = ({ viewerRef }) => {
  const { activeId: activeRobotId } = useRobotSelection();
  const { getRobotPose, setRobotPose } = useRobotManager();
  const [position, setPosition] = useState({ x: 0, y: 0, z: 0 });
  const [isLoading, setIsLoading] = useState(false);

  // Initialize position when robot changes
  useEffect(() => {
    if (!activeRobotId) return;

    setIsLoading(true);

    // Get robot pose using event system (returns Promise)
    getRobotPose(activeRobotId).then(pose => {
      setPosition(pose.position);
      setIsLoading(false);
    });
  }, [activeRobotId, getRobotPose]);

```

```
  const handlePositionChange = (axis, value) => {
    const numValue = parseFloat(value);
    if (isNaN(numValue)) return;

    setPosition(prev => ({
      ...prev,
      [axis]: numValue
    }));
  };

  const applyPosition = useCallback(() => {
    if (!activeRobotId) return;

    // Use event-based system to set robot pose
    setRobotPose(activeRobotId, { position });

    if (viewerRef?.current?.focusOnRobot) {
      viewerRef.current.focusOnRobot(activeRobotId);
    }
  }, [activeRobotId, position, setRobotPose, viewerRef]);

  return (
    <div className="controls-section">
      <h3>Robot Position - {activeRobotId}</h3>
      <AxisControl
        axis="x"
        value={position.x}
        onChange={(value) => handlePositionChange('x', value)}
      />
      <AxisControl
        axis="y"
        value={position.y}
        onChange={(value) => handlePositionChange('y', value)}
      />
      <AxisControl
        axis="z"
        value={position.z}
        onChange={(value) => handlePositionChange('z', value)}
      />
      <button onClick={applyPosition}>Apply Position</button>
    </div>
  );
};
```

Listing 1: Reposition Component Structure

3.2 Hook Layer: useRobotManager

The hook provides async pose operations:

```
export const useRobotManager = () => {
  const context = useRobotContext();

  // Set robot pose using events
  const setRobotPose = useCallback((robotId, pose) => {
    if (!robotId) {
```

```
      console.warn('[useRobotManager] No robot ID for pose control');
      return false;
    }
    EventBus.emit(RobotPoseEvents.Commands.SET_POSE, {
      robotId,
      ...pose
    });
    return true;
  }, []);

  // Get robot pose (returns Promise)
  const getRobotPose = useCallback((robotId) => {
    if (!robotId) {
      return Promise.resolve({
        position: { x: 0, y: 0, z: 0 },
        rotation: { x: 0, y: 0, z: 0 }
      });
    }

    return new Promise((resolve) => {
      const requestId = `getpose_${Date.now()}`;

      const handleResponse = (data) => {
        if (data.requestId === requestId &&
            data.robotId === robotId) {
          EventBus.off(RobotPoseEvents.Responses.GET_POSE,
                       handleResponse);
          resolve({
            position: data.position,
            rotation: data.rotation
          });
        }
      };

      EventBus.on(RobotPoseEvents.Responses.GET_POSE,
                  handleResponse);
      EventBus.emit(RobotPoseEvents.Commands.GET_POSE, {
        robotId,
        requestId
      });

      // Timeout after 1 second
      setTimeout(() => {
        EventBus.off(RobotPoseEvents.Responses.GET_POSE,
                     handleResponse);
        resolve({
          position: { x: 0, y: 0, z: 0 },
          rotation: { x: 0, y: 0, z: 0 }
        });
      }, 1000);
    });
  }, []);

  return {
    // ... other methods
    setRobotPose,
```

```
    getRobotPose,
    robotPoses: context.robotPoses || new Map()
  };
};
```

Listing 2: useRobotManager Pose Methods

4 Event System

4.1 Robot Pose Events Definition

```
export const RobotPoseEvents = {
  // ========== Commands ==========
  Commands: {
    /**
     * Set robot pose (position and rotation)
     * EMITTED BY: Any component
     * LISTENED BY: RobotContext
     * PAYLOAD: {
     *   robotId: string,
     *   position?: {x,y,z},
     *   rotation?: {x,y,z},    // Euler angles
     *   requestId?: string
     * }
     * RESPONSE: RobotPoseEvents.Responses.SET_POSE
     */
    SET_POSE: 'robotpose:command:set-pose',

    /**
     * Get current robot pose
     * EMITTED BY: Any component
     * LISTENED BY: RobotContext
     * PAYLOAD: {
     *   robotId: string,
     *   requestId: string    // Required
     * }
     * RESPONSE: RobotPoseEvents.Responses.GET_POSE
     */
    GET_POSE: 'robotpose:command:get-pose'
  },

  // ========== Responses ==========
  Responses: {
    /**
     * Response to set robot pose
     * PAYLOAD: {
     *   robotId: string,
     *   position: {x,y,z},
     *   rotation: {x,y,z},
     *   success: boolean,
     *   requestId?: string
     * }
     */
```

```
    SET_POSE: 'robotpose:response:set-pose',

    /**
     * Response to get robot pose
     * PAYLOAD: {
     *   robotId: string,
     *   position: {x,y,z},
     *   rotation: {x,y,z},
     *   requestId: string
     * }
     */
    GET_POSE: 'robotpose:response:get-pose'
  }
};
```

Listing 3: RobotPoseEvents in dataTransfer.js

4.2 Event Handling in RobotContext

```
// Event handlers for robot pose events
useEffect(() => {
  // SET_POSE: update robot's position/rotation in 3D scene
  const handleSetPose = (data) => {
    const { robotId, position, rotation } = data;
    const robotData = loadedRobots.get(robotId);

    if (!robotData?.container) {
      log(`[RobotContext] Cannot set pose - robot ${robotId} not
          loaded`);
      return;
    }

    // Apply position if provided
    if (position) {
      robotData.container.position.set(
        position.x ?? robotData.container.position.x,
        position.y ?? robotData.container.position.y,
        position.z ?? robotData.container.position.z
      );
    }

    // Apply rotation if provided
    if (rotation) {
      robotData.container.rotation.set(
        rotation.x ?? robotData.container.rotation.x,
        rotation.y ?? robotData.container.rotation.y,
        rotation.z ?? robotData.container.rotation.z
      );
    }

    // Update THREE.js matrices
    robotData.container.updateMatrix();
    robotData.container.updateMatrixWorld(true);

    // Emit position changed event for other systems
```

```
    EventBus.emit(RobotEvents.POSITION_CHANGED, {
      robotId,
      position: robotData.container.position,
      rotation: robotData.container.rotation
    });
  };

  // GET_POSE: read robot's position/rotation and emit response
  const handleGetPose = (data) => {
    const { robotId, requestId } = data;
    const robotData = loadedRobots.get(robotId);

    let position = { x: 0, y: 0, z: 0 };
    let rotation = { x: 0, y: 0, z: 0 };

    if (robotData?.container) {
      position = {
        x: robotData.container.position.x,
        y: robotData.container.position.y,
        z: robotData.container.position.z
      };
      rotation = {
        x: robotData.container.rotation.x,
        y: robotData.container.rotation.y,
        z: robotData.container.rotation.z
      };
    }

    EventBus.emit(RobotPoseEvents.Responses.GET_POSE, {
      robotId,
      position,
      rotation,
      requestId
    });
  };

  const unsubSet = EventBus.on(RobotPoseEvents.Commands.SET_POSE,
                               handleSetPose);
  const unsubGet = EventBus.on(RobotPoseEvents.Commands.GET_POSE,
                               handleGetPose);

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

Listing 4: RobotContext Pose Event Handlers

5 Container Architecture

5.1 Robot Container Structure

Each robot is wrapped in a container for position management:

```
// During robot loading in RobotContext
const _loadRobotInternal = useCallback(async (robotId, urdfPath,
    options = {}) => {
  const { position = { x: 0, y: 0, z: 0 } } = options;

  // Load URDF robot
  const robot = await loadURDF(urdfPath);

  // Create container for position/rotation
  const robotContainer = new THREE.Object3D();
  robotContainer.name = `${robotId}_container`;
  robotContainer.add(robot);
  robotContainer.position.set(position.x, position.y, position.z);

  // Add to scene
  sceneSetupRef.current.robotRoot.add(robotContainer);

  // Store robot data with container reference
  const robotData = {
    robot: robot,
    container: robotContainer,
    id: robotId,
    // ... other data
  };

  setLoadedRobots(prev => new Map(prev).set(robotId, robotData));

  return robot;
}, []);
```

Listing 5: Container-Based Position Management

5.2 Container Benefits

Benefit	Description
Separation	Robot's internal transforms remain independent
Clean API	Position/rotation always at container level
Multi-robot	Each robot has its own world transform
Animation	Container can be animated without affecting joints
Hierarchy	Tools and attachments maintain proper relationships

Table 1: Container Architecture Benefits

6 State Management

6.1 Pose State Tracking

The system maintains robot poses in state:

```
export const RobotProvider = ({ children }) => {
  // Robot pose state - tracks position and rotation
```

```
  const [robotPoses, setRobotPoses] = useState(new Map());

  // Initialize pose when robot is loaded
  useEffect(() => {
    loadedRobots.forEach((robotData, robotId) => {
      if (!robotPoses.has(robotId) && robotData.container) {
        setRobotPoses(prev => {
          const newMap = new Map(prev);
          newMap.set(robotId, {
            position: {
              x: robotData.container.position.x,
              y: robotData.container.position.y,
              z: robotData.container.position.z
            },
            rotation: {
              x: robotData.container.rotation.x,
              y: robotData.container.rotation.y,
              z: robotData.container.rotation.z
            }
          });
          return newMap;
        });
      }
    });
  }, [loadedRobots, robotPoses]);

  // Example state structure:
  // robotPoses.get('ur5_001') = {
  //   position: { x: 1.5, y: 0, z: 0.5 },
  //   rotation: { x: 0, y: 0, z: 0 }
  // }
};
```

Listing 6: Robot Pose State Management

7 Integration Patterns

7.1 Complete Position Control Flow

```
// 1. Main Application Component
const RobotWorkspace = () => {
  const viewerRef = useRef();

  return (
    <div className="workspace">
      <ThreeCanvas ref={viewerRef} />
      <ControlPanel>
        <RobotSelector />
        <Reposition viewerRef={viewerRef} />
        <JointControl />
      </ControlPanel>
    </div>
  );
```

```
};

// 2. Position Input Component
const PositionInput = ({ label, value, onChange, onMove }) => {
  return (
    <div className="position-input">
      <label>{label}</label>
      <input
        type="number"
        value={value}
        onChange={(e) => onChange(parseFloat(e.target.value))}
        step="0.1"
      />
      <button onClick={() => onMove(-0.1)}>-</button>
      <button onClick={() => onMove(0.1)}>+</button>
    </div>
  );
};

// 3. Enhanced Reposition with Grid Snap
const RepositionAdvanced = () => {
  const { activeId } = useRobotSelection();
  const { getRobotPose, setRobotPose } = useRobotManager();
  const [gridSnap, setGridSnap] = useState(0.1);

  const snapToGrid = (value) => {
    return Math.round(value / gridSnap) * gridSnap;
  };

  const applyWithSnap = () => {
    const snappedPosition = {
      x: snapToGrid(position.x),
      y: snapToGrid(position.y),
      z: snapToGrid(position.z)
    };
    setRobotPose(activeId, { position: snappedPosition });
  };

  return (
    <div>
      {/* Position controls */}
      <label>Grid Snap: {gridSnap}m</label>
      <input
        type="range"
        min="0.01"
        max="1"
        step="0.01"
        value={gridSnap}
        onChange={(e) => setGridSnap(parseFloat(e.target.value))}
      />
    </div>
  );
};
```

Listing 7: Full Integration Example

8 Performance Considerations

8.1 Async Pattern Benefits

Aspect	Benefit
Non-blocking	UI remains responsive during pose queries
Error Handling	Timeouts prevent hanging requests
Flexibility	Easy to add loading states
Consistency	Same pattern for all async operations
Testing	Promises are easier to test than callbacks

Table 2: Async Pattern Advantages

8.2 Event Optimization

```
// Debounced position updates for drag operations
const usePositionDrag = (robotId) => {
  const { setRobotPose } = useRobotManager();
  const dragStateRef = useRef({ isDragging: false });

  // Debounce position updates
  const updatePosition = useMemo(() =>
    debounce((position) => {
      setRobotPose(robotId, { position });
    }, 16), // ~60fps
    [robotId, setRobotPose]
  );

  const handleDrag = useCallback((event) => {
    if (!dragStateRef.current.isDragging) return;

    const position = {
      x: event.world.x,
      y: event.world.y,
      z: event.world.z
    };

    updatePosition(position);
  }, [updatePosition]);

  return { handleDrag };
};
```

Listing 8: Optimized Event Handling

9 Best Practices

9.1 Event-Driven Guidelines

1. **Always Use Events**: Never directly manipulate containers

2. **Request IDs**: Essential for async operations
3. **Timeouts**: Prevent hanging Promise chains
4. **State Sync**: Keep robotPoses Map updated
5. **Matrix Updates**: Always call updateMatrixWorld()

9.2 UI/UX Guidelines

1. **Loading States**: Show during async operations
2. **Input Validation**: Validate numeric inputs
3. **Visual Feedback**: Update position immediately in UI
4. **Grid Snap**: Offer position snapping options
5. **Undo/Redo**: Consider position history

10 Advanced Features

10.1 Position Constraints

```
const usePositionConstraints = () => {
  const [constraints, setConstraints] = useState({
    minX: -10, maxX: 10,
    minY: -10, maxY: 10,
    minZ: 0, maxZ: 10
  });

  const constrainPosition = (position) => {
    return {
      x: Math.max(constraints.minX,
          Math.min(constraints.maxX, position.x)),
      y: Math.max(constraints.minY,
          Math.min(constraints.maxY, position.y)),
      z: Math.max(constraints.minZ,
          Math.min(constraints.maxZ, position.z))
    };
  };

  const setConstrainedRobotPose = (robotId, pose) => {
    const constrained = {
      ...pose,
      position: pose.position ?
        constrainPosition(pose.position) : undefined
    };
    setRobotPose(robotId, constrained);
  };

  return { constraints, setConstraints, setConstrainedRobotPose };
};
```

Listing 9: Position Constraints System

10.2 Multi-Robot Coordination

```
const useMultiRobotPosition = () => {
  const { loadedRobots, getRobotPose, setRobotPose } =
     useRobotManager();

  // Arrange robots in a line
  const arrangeInLine = async (spacing = 2) => {
    const robotIds = Array.from(loadedRobots.keys());

    for (let i = 0; i < robotIds.length; i++) {
      await setRobotPose(robotIds[i], {
        position: { x: i * spacing, y: 0, z: 0 }
      });
    }
  };

  // Arrange robots in a circle
  const arrangeInCircle = async (radius = 3) => {
    const robotIds = Array.from(loadedRobots.keys());
    const angleStep = (2 * Math.PI) / robotIds.length;

    for (let i = 0; i < robotIds.length; i++) {
      const angle = i * angleStep;
      await setRobotPose(robotIds[i], {
        position: {
          x: radius * Math.cos(angle),
          y: 0,
          z: radius * Math.sin(angle)
        },
        rotation: {
          x: 0,
          y: angle + Math.PI / 2,
          z: 0
        }
      });
    }
  };

  return { arrangeInLine, arrangeInCircle };
};
```

Listing 10: Multi-Robot Position Management

11 Troubleshooting

11.1 Common Issues

Issue	Solution
Position not updating	Check robot is loaded and has container
Promise timeout	Verify EventBus subscriptions are active
Wrong position	Ensure using container, not robot position
Jump on load	Initialize pose state after robot loads
Camera not following	Check viewerRef is properly passed

Table 3: Common Position Control Issues

12 Future Enhancements

1. **Path Planning**: Move along defined paths
2. **Collision Detection**: Prevent invalid positions
3. **Position History**: Undo/redo position changes
4. **Keyboard Control**: Arrow keys for fine positioning
5. **AR Markers**: Position based on real-world markers
6. **Save/Load**: Persist robot arrangements

13 Conclusion

The Robot Pose System demonstrates a clean event-driven architecture for managing robot positions in 3D space. By using a container-based approach with asynchronous event communication, the system maintains separation between UI controls and the underlying THREE.js scene while providing responsive, real-time position updates.

The Promise-based API for pose queries ensures non-blocking operations, while the event system enables loose coupling between components. This architecture scales well for multi-robot scenarios and provides a solid foundation for advanced positioning features.