Collision Detection in dnd-kit: Complete Guide

Table of Contents

1. What is Collision Detection?
2. Built-in Collision Detection Algorithms
3. Why Custom Collision Detection?
4. Understanding the Dual-ID Pattern
5. Custom Collision Detection Breakdown
6. Priority System
7. Visual Examples

What is Collision Detection? {#what-is-collision-detection}

Collision detection is the algorithm that determines which droppable area(s) a draggable item is currently hovering over. It's the core mechanism that makes drag-and-drop work.

The Process

```
User drags item → Collision Detection runs → Returns list of overlapping
droppables → dnd-kit sets "over"
```

Key Concepts

- **Active**: The item currently being dragged
- **Over**: The droppable zone the active item is currently over
- **Collision**: When a draggable item's bounds intersect with a droppable zone's bounds

---

Built-in Collision Detection Algorithms {#built-in-algorithms}

dnd-kit provides several built-in collision detection algorithms:

1. `rectIntersection` (Default)

```
import { rectIntersection } from "@dnd-kit/core";
```

How it works: Uses rectangular bounding box intersection. Returns all droppables whose bounding box intersects with the draggable item's bounding box.

Pros:

- Fast and efficient
- Works well for grid layouts

- Detects overlaps even with small intersections

Cons:

- Can detect multiple collisions at once
- Less intuitive for nested layouts

Best for: Grid layouts, card lists, simple sortable lists

---

2. `pointerWithin`

```
import { pointerWithin } from "@dnd-kit/core";
```

How it works: Checks if the pointer (mouse/touch) is within the bounds of a droppable area.

Pros:

- Very precise
- Only one collision at a time
- Intuitive for users (follows the cursor)

Cons:

- Can miss large droppable areas if pointer is outside
- Requires pointer to be directly over target

Best for: Large drop zones, precise targeting

---

3. `closestCenter`

```
import { closestCenter } from "@dnd-kit/core";
```

How it works: Returns the droppable whose center point is closest to the draggable item's center point.

Pros:

- Only one collision at a time
- Works well for sorting
- Predictable behavior

Cons:

- Can feel less intuitive for large items
- Doesn't consider actual overlap

Best for: Sortable lists, reordering

---

4. `closestCorners`

```
import { closestCorners } from "@dnd-kit/core";
```

How it works: Calculates distance between corners of the draggable and droppable items.

Pros:

- More accurate for irregularly shaped items
- Works well for multi-row/column layouts

Cons:

- More computationally expensive
- Can be less predictable

Best for: Complex grid layouts, Kanban boards

Why Custom Collision Detection? {#why-custom}

In this project, we need custom collision detection because we have **multiple competing collision types**:

The Challenge

```
When dragging a script, we need to:
1. ✅ Allow reordering with other scripts (script-to-script collision)
2. ✅ Allow dropping into folders (script-to-folder collision)
3. ✅ Allow dropping into root area (script-to-root-droppable collision)
4. ✅ Prioritize script reordering over folder dropping when both are
possible
5. ✅ Highlight folders when hovering over them
6. ✅ Highlight root area when hovering over it
```

Built-in algorithms can't handle this complexity because they don't understand our business logic and priorities.

Understanding the Dual-ID Pattern {#dual-id-pattern}

Why Do Folders Have Two IDs?

Each folder has **two roles**:

1. **Sortable** (for reordering folders with each other)
    - ID: `folder.id` (e.g., `20`, `25`)
    - Hook: `useSortable()`
    - Purpose: Allow folders to be reordered

2. **Droppable** (for receiving scripts)
   - ID: `folder-droppable-${folder.id}` (e.g., `folder-droppable-20`)
   - Hook: `useDroppable()`
   - Purpose: Allow scripts to be dropped into the folder

Example in Code

```
// Sortable: for reordering folders
const { setNodeRef: setSortableNodeRef } = useSortable({
    id: folder.id, // e.g., 20
    data: { type: "folder", folderId: folder.id },
});

// Droppable: for receiving scripts
const { setNodeRef: setDroppableNodeRef } = useDroppable({
    id: `folder-droppable-${folder.id}`, // e.g., "folder-droppable-20"
    data: { type: "folder", folderId: folder.id },
});

// Combine both refs on the same element
const combinedRef = (node: HTMLElement | null) => {
    setSortableNodeRef(node);
    setDroppableNodeRef(node);
};
```

Why Not Just One ID?

If we only used `folder.id`:

- ❌ Collision detection would confuse folder reordering with script dropping
- ❌ Hard to distinguish "drag script over folder" from "drag folder over folder"
- ❌ Difficult to prioritize script sorting over folder dropping

With dual IDs:

- ✅ Clear separation of concerns
- ✅ Can detect and prioritize different collision types
- ✅ Can highlight folders independently of sorting

Custom Collision Detection Breakdown {#custom-breakdown}

Let's break down the custom collision detection function step by step:

Step 1: Identify What's Being Dragged

```
const customCollisionDetection: CollisionDetection = (args) => {
    const { active } = args;
    const isDraggingScript = active?.data.current?.type === "script";
```

Purpose: Check if we're dragging a script (vs. dragging a folder)

Why: Different dragging types need different collision logic:

- **Script**: Need to check for scripts, folders, and root droppable
- **Folder**: Only need to check for other folders (default behavior)

Step 2: Get Collision Candidates

```
if (isDraggingScript) {
    const pointerCollisions = pointerWithin(args);
    const rectCollisions = rectIntersection(args);
```

Purpose: Get two lists of potential collisions:

1. `pointerCollisions`: Items the pointer is directly over (precise)
2. `rectCollisions`: Items that overlap with dragged item's bounds (broad)

Why both?

- `pointerWithin`: For precise targeting (e.g., which script to swap with)
- `rectIntersection`: For broader detection (e.g., detect folder even if pointer is on script inside it)

Step 3: Filter Script Collisions

```
const scriptCollisions = pointerCollisions.filter(({ id, data }) => {
    const isDroppable = String(id).includes("droppable");
    const isFolder = data?.current?.type === "folder";
    return !isDroppable && !isFolder;
});
```

Purpose: Find all script items the pointer is over

Filters out:

- Droppable zones (IDs containing "droppable")
- Folders (type === "folder")

Result: Only script items remain

Step 4: Find Folder Droppable Collision

```
const folderDroppableCollision =
    pointerCollisions.find(({ id }) => String(id).startsWith("folder-
droppable-")) ||
    rectCollisions.find(({ id }) => String(id).startsWith("folder-
droppable-"));
```

Purpose: Check if we're over any folder's droppable zone

Why both pointer and rect?

- `pointerCollisions`: Detects when pointer is directly over folder row
- `rectCollisions`: Detects when dragged script overlaps folder (even if pointer is slightly off)

Result: The folder droppable we're currently over (if any)

Step 5: Prioritize Script Collisions

```
if (scriptCollisions.length > 0) {
    const droppablesToInclude = [];

    // Check for root droppable
    const rootDroppableCollision = /* ... */;
    if (rootDroppableCollision) {
        droppablesToInclude.push(rootDroppableCollision);
    }

    // Check for folder droppable
    if (folderDroppableCollision) {
        droppablesToInclude.push(folderDroppableCollision);
    }

    // Script collisions first, then droppables
    return [...scriptCollisions, ...droppablesToInclude];
}
```

Purpose: When over scripts, prioritize script reordering but also notify droppables

Why this order?

1. **Script collisions first**: dnd-kit will use the first collision for `over`, enabling smooth reordering
2. **Droppables second**: Still notified of the collision, so they can show highlights

Result: Scripts can reorder smoothly while folders/root still highlight

---

Step 6: Fallback to Folder Droppable

```
if (folderDroppableCollision) {
    return [folderDroppableCollision];
}
```

Purpose: If not over any scripts, check if over a folder

When this happens: Dragging into empty folder area or between script items

Result: Folder becomes the `over` target, can be dropped there

Step 7: Fallback to Root Droppable

```
const rootDroppableCollision = /* ... */;
if (rootDroppableCollision) {
    return [rootDroppableCollision];
}
```

Purpose: If not over scripts or folders, check if over root folder area

When this happens: Dragging into empty root scripts area

Result: Root area becomes the `over` target, can drop there

Step 8: Default Behavior

```
// For folders or when no droppable collision, use rect intersection
return rectIntersection(args);
```

Purpose: Fallback for all other cases (e.g., dragging folders)

Result: Standard dnd-kit collision detection

Priority System {#priority-system}

The collision detection implements a clear priority system:

When Dragging a Script:

```
Priority 1: Script items (for reordering)
    ↓
Priority 2: Folder droppables (for moving to folder)
    ↓
Priority 3: Root droppable (for moving to root)
```

```
     ↓
Priority 4: Default (rect intersection)
```

Visual Decision Tree

```
Is dragging a script?
├─ YES
│  ├─ Over any script items?
│  │  ├─ YES → Return scripts + droppables (for sorting + highlighting)
│  │  └─ NO → Continue...
│  │
│  ├─ Over any folder droppable?
│  │  ├─ YES → Return folder droppable (for dropping)
│  │  └─ NO → Continue...
│  │
│  ├─ Over root droppable?
│  │  ├─ YES → Return root droppable (for dropping)
│  │  └─ NO → Return default
│  │
└─ NO (dragging folder) → Return default (rect intersection)
```

Visual Examples {#visual-examples}

Example 1: Dragging Script Over Another Script

```
┌─────────────────────────────────────┐
│ Root Folder Scripts                 │
│ ┌───────────────────────────────┐   │
│ │ 📄 Script A (pointer here) ⬅️ │   │ ← scriptCollisions[0]
│ └───────────────────────────────┘   │
│                                     │
│ ┌───────────────────────────────┐   │
│ │ 📄 Script B                   │   │
│ └───────────────────────────────┘   │
└─────────────────────────────────────┘
    ↑
    rootDroppableCollision (also detected)
```

Collision Detection Returns:

```
[
    { id: scriptA.id, type: "script" }, // First = used for "over"
    { id: "root-scripts-droppable-1" }, // For highlighting
];
```

Result:

- `over` = Script A (scripts swap)
- Root area highlighted ✅

Example 2: Dragging Script Over Folder

```
┌──────────────────────────────┐
│  📁 Projects (pointer here) ⬅️  │ ← folderDroppableCollision
│     ├─ Script X              │
│     └─ Script Y              │
│                              │
└──────────────────────────────┘
```

Collision Detection Returns:

```
[{ id: "folder-droppable-20", type: "folder" }];
```

Result:

- `over` = Folder droppable (can drop)
- Folder highlighted ✅

Example 3: Dragging Script Over Script Inside Folder

```
┌──────────────────────────────┐
│  📁 Projects ▼                 │
│     ┌──────────────────────┐ │
│     │ 📄 Script X (pointer) ⬅️ │ │ ← scriptCollisions[0]
│     └──────────────────────┘ │
│     ┌──────────────────────┐ │
│     │ 📄 Script Y            │ │
│     └──────────────────────┘ │
└──────────────────────────────┘
   ↑
  folderDroppableCollision (also detected)
```

Collision Detection Returns:

```
[
    { id: scriptX.id, type: "script" }, // First = used for "over"
    { id: "folder-droppable-20" }, // For highlighting
];
```

Result:

- `over` = Script X (scripts swap within folder)
- Folder also highlighted ✅

Example 4: Dragging Script Over Empty Root Area

```
┌─────────────────────────────────────┐
│  Root Folder Scripts                │
│                                     │
│     (empty area, pointer here) ⬅️   │
│                                     │
└─────────────────────────────────────┘
   ↑
   rootDroppableCollision
```

Collision Detection Returns:

```
[{ id: "root-scripts-droppable-1" }];
```

Result:

- `over` = Root droppable (can drop)
- Root area highlighted ✅

Key Takeaways

1. **Collision detection determines what `over` is set to** during drag operations

2. **Custom collision detection allows prioritization** of different collision types

3. **The dual-ID pattern** (sortable + droppable) enables:

    - Clear separation of folder reordering vs. script dropping
    - Better collision detection accuracy
    - Independent highlighting logic

4. **Returning multiple collisions** allows:

    - First item = what dnd-kit uses for `over`
    - Remaining items = notified of collision (for highlighting)

5. **Combining `pointerWithin` and `rectIntersection`** provides:

    - Precise pointer-based detection
    - Broad overlap-based detection
    - Best of both worlds

6. **Priority order matters**:
    - Scripts first (smooth reordering)
    - Folders second (can drop when not over scripts)
    - Root third (fallback for empty areas)

Debugging Tips

Log Collisions

```
const customCollisionDetection: CollisionDetection = (args) => {
    const result = /* your logic */;
    console.log("Collision result:", result);
    return result;
};
```

Check Over State

```
const { over } = useDndContext();
console.log("Current over:", over?.id, over?.data.current);
```

Visualize Droppable IDs

Add temporary labels to see which ID is which:

```
<div className="droppable-id-debug">
    ID: {droppableId}
</div>
```

Related Files

- `ScriptsColumn.tsx`: Contains the custom collision detection function
- `SortatbleCollapsableFolder.tsx`: Implements dual-ID pattern for folders
- `SortableScriptsContext.tsx`: Root droppable area implementation
- `SortableScriptItem.tsx`: Individual script sortable items

Further Reading

- dnd-kit Collision Detection Docs
- Custom Collision Detection Example
- Understanding Droppable