Interactive X-Sheet\interactive xsheet working v1-06_01.html

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Interactive Animation X-Sheet with Audio Waveform</title>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/jspdf/2.5.1/jspdf.umd.min.js">
</script>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/html2canvas/1.4.1/html2canvas.min.js"></script>
    <style>
        /* General styling */
        body {
            font-family: Arial, sans-serif;
            font-size: 10pt;
            line-height: 1.2;
            margin: 0;
            padding: 10px;
        }

        .controls {
            background-color: #f5f5f5;
            padding: 10px;
            margin-bottom: 15px;
            border-radius: 5px;
            display: flex;
            flex-wrap: wrap;
            gap: 10px;
            align-items: center;
        }

        .controls select,
        .controls input,
        .controls button {
            padding: 6px 10px;
            border: 1px solid #ccc;
            border-radius: 4px;
        }

        .controls button {
            background-color: #4CAF50;
            color: white;
            border: none;
            cursor: pointer;
            transition: background-color 0.3s;
        }

```

```
        .controls button:hover {
            background-color: #45a049;
        }

        #pdf-button {
            background-color: #f44336;
        }

        #pdf-button:hover {
            background-color: #d32f2f;
        }

        #print-button {
            background-color: #2196F3;
        }

        #print-button:hover {
            background-color: #0b7dda;
        }

        #audio-button {
            background-color: #9c27b0;
        }

        #audio-button:hover {
            background-color: #7b1fa2;
        }

        .header {
            text-align: center;
            margin-bottom: 5px;
        }

        .title {
            font-size: 14pt;
            font-weight: bold;
            margin-bottom: 5px;
        }

        .metadata {
            display: grid;
            grid-template-columns: repeat(3, 1fr);
            gap: 5px;
            margin-bottom: 10px;
        }

        .metadata div {
            border: 1px solid #000;
            padding: 3px 5px;
        }
```

```
97
98          .metadata span {
99              font-weight: bold;
100             margin-right: 5px;
101         }
102
103         .metadata input {
104             border: none;
105             width: 70%;
106             font-family: Arial, sans-serif;
107             font-size: 9pt;
108         }
109
110         table {
111             width: 100%;
112             border-collapse: collapse;
113             table-layout: fixed;
114         }
115
116         th,
117         td {
118             border: 1px solid #000;
119             padding: 2px 4px;
120             vertical-align: top;
121             height: 20px;
122             overflow: hidden;
123         }
124
125         th {
126             background-color: #eee;
127             font-weight: bold;
128             text-align: center;
129             font-size: 9pt;
130         }
131
132         .action-col {
133             width: 16%;
134         }
135
136         .frame-col {
137             width: 4%;
138             text-align: center;
139         }
140
141         .waveform-col {
142             width: 10%;
143             padding: 0;
144             position: relative;
145         }
146
```

```
        .dialogue-col {
            width: 10%;
            text-align: center;
        }

        .sound-col {
            width: 9%;
            text-align: center;
        }

        .technical-col {
            width: 9%;
        }

        .extra1-col {
            width: 8%;
        }

        .extra2-col {
            width: 8%;
        }

        .camera-col {
            width: 12%;
        }

        .waveform-container {
            position: absolute;
            width: 100%;
            top: 0;
            left: 0;
            z-index: 10;
            pointer-events: none;
        }

        .waveform-overlay {
            position: absolute;
            width: 100%;
            height: 100%;
            top: 0;
            left: 0;
            z-index: 11;
            pointer-events: auto;
            cursor: crosshair;
        }

        .waveform-marker {
            position: absolute;
            width: 100%;
            height: 20px;
```

```
            background-color: rgba(255, 255, 0, 0.2);
            pointer-events: none;
            text-align: center;
            font-size: 7pt;
            line-height: 18px;
            color: #666;
            z-index: 20;
        }

        .waveform-canvas {
            position: absolute;
            top: 0;
            left: 0;
            width: 100%;
            z-index: 15;
            pointer-events: none;
        }

        .waveform-col-container {
            position: relative;
        }

        .phonetic-label {
            position: absolute;
            background-color: rgba(255, 255, 255, 0.8);
            border: 1px solid #ccc;
            border-radius: 2px;
            font-size: 7pt;
            padding: 1px 2px;
            z-index: 3;
            pointer-events: none;
        }

        .footer {
            font-size: 8pt;
            text-align: center;
            margin-top: 5px;
            color: #333;
            font-style: italic;
        }

        [contenteditable="true"] {
            min-height: 18px;
            cursor: text;
        }

        [contenteditable="true"]:focus {
            background-color: #f0f7ff;
            outline: none;
        }
```

```
247
248         [contenteditable="true"]:empty:before {
249             content: attr(data-placeholder);
250             color: #888;
251             font-style: italic;
252         }
253
254         .frame-number {
255             background-color: #eee;
256             font-weight: bold;
257             text-align: center;
258         }
259
260         .modified {
261             background-color: #fffacd;
262         }
263
264         .selected-cell {
265             background-color: rgba(0, 123, 255, 0.2) !important;
266             outline: 2px solid #0d6efd;
267         }
268
269         .status {
270             margin-top: 10px;
271             padding: 5px;
272             background-color: #f0f0f0;
273             border-radius: 4px;
274             font-style: italic;
275             color: #555;
276         }
277
278         #audio-controls {
279             display: flex;
280             flex-wrap: wrap;
281             gap: 5px;
282             align-items: center;
283             margin-top: 5px;
284             padding: 5px;
285             background-color: #f9f9f9;
286             border-radius: 4px;
287         }
288
289         #audio-controls button {
290             padding: 4px 8px;
291             background-color: #673ab7;
292             color: white;
293             border: none;
294             border-radius: 3px;
295             cursor: pointer;
296         }
```

```
297
298         #audio-controls button:hover {
299             background-color: #5e35b1;
300         }
301
302         #audio-info {
303             font-size: 8pt;
304             color: #333;
305         }
306
307         #phonetic-input {
308             position: absolute;
309             z-index: 100;
310             background: white;
311             border: 1px solid #ccc;
312             padding: 5px;
313             border-radius: 4px;
314             box-shadow: 0 2px 4px rgba(0, 0, 0, 0.2);
315             display: none;
316         }
317
318         #audio-upload {
319             display: none;
320         }
321
322         /* Drawing system specific styles */
323         .drawing-toolbar button {
324             transition: background-color 0.2s, color 0.2s;
325         }
326
327         .drawing-toolbar button:hover {
328             background-color: #e6e6e6 !important;
329         }
330
331         .drawing-toolbar button.active {
332             background-color: #4CAF50 !important;
333             color: white !important;
334         }
335
336         .drawing-layer-container {
337             pointer-events: none;
338         }
339
340         .drawing-layer-container canvas {
341             pointer-events: none;
342             touch-action: none;
343         }
344
345         /* Print specific styles */
346         @media print {
```

```
            .controls,
            button,
            #frame-count-container,
            .status,
            #audio-controls,
            #phonetic-input,
            .drawing-toolbar {
                display: none !important;
            }

            body {
                margin: 0;
                padding: 0;
            }

            @page {
                size: auto;
                margin: 0.5cm;
            }

            thead {
                display: table-header-group;
            }

            tfoot {
                display: table-footer-group;
            }

            tr {
                page-break-inside: avoid;
            }

            /* Better waveform printing */
            .waveform-col {
                position: relative !important;
                overflow: hidden !important;
            }

            .print-waveform-container {
                display: block !important;
                position: absolute !important;
                z-index: 1000 !important;
                pointer-events: none !important;
                overflow: hidden !important;
            }

            .waveform-col * {
                page-break-inside: avoid !important;
            }

```

```
            /* ensure the waveform clone prints correctly */
            .print-waveform-clone {
                display: block !important;
                position: absolute !important;
                z-index: 1000 !important;
                pointer-events: none !important;
            }

            .cell-waveform-window {
                position: relative !important;
                width: 100% !important;
                height: 100% !important;
                overflow: hidden !important;
            }

            /* Hide original waveform during print */
            body>.waveform-container {
                display: none !important;
            }

           .drawing-layer-container {
        display: block !important;
        position: absolute !important;
        z-index: 1000 !important;
        pointer-events: none !important;
        page-break-inside: avoid !important;
    }

        .drawing-layer-container.printing {
            transform: translate(0, 0) !important; /* Prevent any transforms during print */
        }

        .drawing-layer-container canvas {
        display: block !important;
        position: absolute !important;
    }
        }
    </style>
</head>

<body>

    <script>
        function updateLayoutSize() {
            // Preserve any existing functionality in this function

            // Update drawing layer position and size based on table position
            if (window.xsheetDrawing && window.xsheetDrawing.layerSystem) {
                const tableRect = document.getElementById('xsheet-
table').getBoundingClientRect();
```

```
                const drawingContainer = document.querySelector('.drawing-layer-container');

                if (drawingContainer) {
                    drawingContainer.style.left = tableRect.left + 'px';
                    drawingContainer.style.top = tableRect.top + 'px';
                    drawingContainer.style.width = tableRect.width + 'px';
                    drawingContainer.style.height = tableRect.height + 'px';

                    // Trigger redraw of the drawing layer
                    window.xsheetDrawing.layerSystem.resize(tableRect.width, tableRect.height);
                }
            }
        }

        // Add event listener for window resize if not already present
        window.addEventListener('resize', updateLayoutSize);
    </script>
    <div class="controls">
        <div>
            <label for="template-selector">Template:</label>
            <select id="template-selector">
                <option value="large">11"x17" (96 Frames)</option>
                <option value="small">8"x10" (48 Frames)</option>
            </select>
        </div>

        <div id="frame-count-container">
            <label for="frame-count">Frames:</label>
            <input type="number" id="frame-count" min="24" step="8" value="96">
        </div>

        <button id="audio-button">Import Audio</button>
        <input type="file" id="audio-upload" accept="audio/*">

        <button id="save-button">Save Project</button>
        <button id="load-button">Load Project</button>
        <button id="pdf-button">Export PDF</button>
        <button id="print-button">Print</button>
        <button id="add-rows-button">Add 8 Rows</button>
        <button id="clear-button">Clear All</button>
    </div>

    <script>
        // Initialize the drawing layer system
        const xsheetTable = document.getElementById('xsheet-table');
        window.xsheetDrawing = {
            layerSystem: new DrawingLayerSystem(xsheetTable)
        };

```

```
        // Initialize the drawing toolbar and connect it to the layer system
        const drawingToolbar = new DrawingToolSystem(window.xsheetDrawing.layerSystem);

        // Append the toolbar to the controls section
        document.querySelector('.controls').appendChild(drawingToolbar.element);

        // Call updateLayoutSize initially to set correct position
        updateLayoutSize();
    </script>

    <div id="audio-controls">
        <button id="play-audio">Play/Pause</button>
        <button id="stop-audio">Stop</button>
        <input type="range" id="audio-scrubber" min="0" max="100" value="0" style="width: 200px;">
        <span id="audio-info">No audio loaded</span>
        <button id="add-phonetic">Add Phonetic Marker</button>
        <div style="margin-left: 10px; color: #666; font-style: italic;">
            ✨ TIP: Drag down the waveform column while holding the left mouse button to scrub audio frame-by-frame<br>
            💡 TIP: Click and drag to select multiple cells (use Ctrl+C to copy, Delete to clear)
        </div>
    </div>

    <div id="phonetic-input">
        <input type="text" id="phonetic-text" placeholder="Enter phonetic sound">
        <button id="save-phonetic">Save</button>
        <button id="cancel-phonetic">Cancel</button>
    </div>

    <div id="printable-area">
        <div class="header">
            <div class="title">3D ANIMATION X-SHEET</div>
        </div>

        <div class="metadata">
            <div><span>Project #:</span><input type="text" id="project-number"></div>
            <div><span>DATE:</span><input type="date" id="project-date"></div>
            <div><span>PAGE #:</span><input type="text" id="page-number"></div>
            <div><span>ANIMATOR:</span><input type="text" id="animator-name"></div>
            <div><span>VERSION:</span><input type="text" id="version-number"></div>
            <div><span>Shot #:</span><input type="text" id="shot-number"></div>
        </div>

        <table id="xsheet-table">
            <thead>
                <tr>
                    <th class="action-col">Action/Description</th>
                    <th class="frame-col">Fr</th>
                    <th class="waveform-col">Audio Waveform</th>
```

```
                    <th class="dialogue-col">Dialogue</th>
                    <th class="sound-col">Sound FX</th>
                    <th class="technical-col">Tech. Notes</th>
                    <th class="extra1-col">Extra 1</th>
                    <th class="extra2-col">Extra 2</th>
                    <th class="frame-col">Fr</th>
                    <th class="camera-col">Camera Moves</th>
                </tr>
            </thead>
            <tbody id="xsheet-body">
                <!-- Rows will be generated via JavaScript -->
            </tbody>
        </table>

        <div class="footer">
            Bold lines mark 8-frame intervals. Double lines mark 24-frame intervals (24fps).
            Left columns track character actions, middle columns for technical notes, right for camera moves.
        </div>
    </div>

    <div class="status" id="status-message"></div>

    <script>
        /**
         * INTERACTIVE X-SHEET DRAWING TOOLS IMPLEMENTATION
         *
         * This code adds comprehensive drawing capabilities to the animation X-Sheet.
         * Features include:
         * - Multiple drawing layers
         * - Various drawing tools (pen, line, arrows, shapes, text, images, animation symbols)
         * - Grid-aware annotation that can span multiple frames
         * - Object selection and manipulation
         * - Integration with saving, loading, and printing
         */

        /**
         * DRAWING LAYER SYSTEM
         * Manages the canvas layers that contain drawing objects
         */
        class DrawingLayerSystem {
            constructor(xsheetTable) {
                this.xsheetTable = xsheetTable;
                this.layers = [];
                this.activeLayerIndex = 0;
                this.container = null;

                this.init();
            }
```

```
591
592             init() {
593                 // Create container aligned with table
594                 const tableRect = this.xsheetTable.getBoundingClientRect();
595                 this.container = document.createElement('div');
596                 this.container.className = 'drawing-layer-container';
597                 this.container.style.position = 'absolute';
598                 this.container.style.left = `${tableRect.left}px`;
599                 this.container.style.top = `${tableRect.top}px`;
600                 this.container.style.width = `${tableRect.width}px`;
601                 this.container.style.height = `${tableRect.height}px`;
602                 this.container.style.pointerEvents = 'none'; // Initially pass events
    through
603                 this.container.style.zIndex = '5';
604                 this.container.style.overflow = 'hidden'; // Prevent drawings from
    overflowing
605                 document.body.appendChild(this.container);
606
607                 // Create default background and foreground layers
608                 this.addLayer('background');
609                 this.addLayer('foreground');
610                 this.setActiveLayer(1); // Set foreground as active by default
611
612                 // Handle window resize and table changes
613                 this.setupResizeHandling();
614             }
615
616             addLayer(name) {
617                 const canvas = document.createElement('canvas');
618                 canvas.className = `drawing-layer-${name}`;
619                 canvas.width = this.container.clientWidth;
620                 canvas.height = this.container.clientHeight;
621                 canvas.style.position = 'absolute';
622                 canvas.style.left = '0';
623                 canvas.style.top = '0';
624                 canvas.style.pointerEvents = 'none';
625
626                 this.container.appendChild(canvas);
627
628                 const layer = {
629                     name: name,
630                     canvas: canvas,
631                     context: canvas.getContext('2d'),
632                     objects: [],
633                     visible: true
634                 };
635
636                 this.layers.push(layer);
637                 return this.layers.length - 1; // Return index of new layer
638             }
```

```
639
640             setActiveLayer(index) {
641                 if (index >= 0 && index < this.layers.length) {
642                     this.activeLayerIndex = index;
643                     return true;
644                 }
645                 return false;
646             }
647
648             getActiveLayer() {
649                 return this.layers[this.activeLayerIndex];
650             }
651            updateLayoutSize() {
652                 const tableRect = this.xsheetTable.getBoundingClientRect();
653
654                 // Store current drawings from each layer
655                 const tempCanvases = this.layers.map(layer => {
656                     const tempCanvas = document.createElement('canvas');
657                     tempCanvas.width = layer.canvas.width;
658                     tempCanvas.height = layer.canvas.height;
659                     const tempCtx = tempCanvas.getContext('2d');
660                     tempCtx.drawImage(layer.canvas, 0, 0);
661                     return tempCanvas;
662                 });
663
664                 // Update container position and dimensions
665                 this.container.style.position = 'absolute';
666                 this.container.style.left = `${tableRect.left}px`;
667                 this.container.style.top = `${tableRect.top}px`;
668                 this.container.style.width = `${tableRect.width}px`;
669                 this.container.style.height = `${tableRect.height}px`;
670
671                 // Update each layer canvas
672                 this.layers.forEach((layer, i) => {
673                     const scaleX = tableRect.width / layer.canvas.width;
674                     const scaleY = tableRect.height / layer.canvas.height;
675
676                     layer.canvas.width = tableRect.width;
677                     layer.canvas.height = tableRect.height;
678
679                     // Redraw with scaling
680                     layer.context.save();
681                     layer.context.scale(scaleX, scaleY);
682                     layer.context.drawImage(tempCanvases[i], 0, 0);
683                     layer.context.restore();
684                 });
685
686                 // Force redraw of all objects
687                 this.redrawAll();
688
```

```
689                 // Dispatch a custom event that the layout was updated
690                 document.dispatchEvent(new Event('drawing-layout-updated'));
691             }
692 
693             enableDrawing() {
694                 this.layers.forEach(layer => {
695                     layer.canvas.style.pointerEvents = 'auto';
696                 });
697             }
698 
699             disableDrawing() {
700                 this.layers.forEach(layer => {
701                     layer.canvas.style.pointerEvents = 'none';
702                 });
703             }
704 
705             clearLayer(layerIndex) {
706                 if (layerIndex >= 0 && layerIndex < this.layers.length) {
707                     const layer = this.layers[layerIndex];
708                     layer.context.clearRect(0, 0, layer.canvas.width, layer.canvas.height);
709                     layer.objects = [];
710                 }
711             }
712 
713             clearAllLayers() {
714                 this.layers.forEach((layer, index) => {
715                     this.clearLayer(index);
716                 });
717             }
718 
719             // Convert screen coordinates to canvas coordinates
720             screenToCanvas(screenX, screenY) {
721                 const containerRect = this.container.getBoundingClientRect();
722                 return {
723                     x: screenX - containerRect.left,
724                     y: screenY - containerRect.top
725                 };
726             }
727 
728             // Convert frame/column to canvas coordinates (for multi-frame spanning)
729             gridToCanvas(frame, column) {
730                 const cell = document.querySelector(`tr.frame-${frame} td:nth-
    child(${column})`);
731                 if (!cell) return null;
732 
733                 const cellRect = cell.getBoundingClientRect();
734                 const containerRect = this.container.getBoundingClientRect();
735 
736                 return {
737                     x: cellRect.left - containerRect.left + cellRect.width / 2,
```

```
                    y: cellRect.top - containerRect.top + cellRect.height / 2
                };
            }

            // Add a drawing object to the active layer
            addObject(object) {
                const layer = this.getActiveLayer();
                layer.objects.push(object);
                this.redrawLayer(this.activeLayerIndex);
                return object;
            }

            // Redraw a specific layer
            redrawLayer(layerIndex) {
                if (layerIndex >= 0 && layerIndex < this.layers.length) {
                    const layer = this.layers[layerIndex];
                    layer.context.clearRect(0, 0, layer.canvas.width, layer.canvas.height);

                    // Draw all objects in this layer
                    layer.objects.forEach(obj => {
                        if (obj.visible) {
                            obj.draw(layer.context);
                        }
                    });
                }
            }

            // Redraw all layers
            redrawAll() {
                this.layers.forEach((_, index) => {
                    this.redrawLayer(index);
                });
            }

            // Find object under point
            findObjectAt(x, y) {
                // Check active layer first, then others in reverse order (top to bottom)
                const activeLayer = this.getActiveLayer();

                // Check active layer
                for (let i = activeLayer.objects.length - 1; i >= 0; i--) {
                    const obj = activeLayer.objects[i];
                    if (obj.containsPoint(x, y)) {
                        return { object: obj, layerIndex: this.activeLayerIndex };
                    }
                }

                // Check other layers from top to bottom
                for (let l = this.layers.length - 1; l >= 0; l--) {
```

```
                    if (l === this.activeLayerIndex) continue; // Skip active layer (already checked)

                    const layer = this.layers[l];
                    if (!layer.visible) continue;

                    for (let i = layer.objects.length - 1; i >= 0; i--) {
                        const obj = layer.objects[i];
                        if (obj.containsPoint(x, y)) {
                            return { object: obj, layerIndex: l };
                        }
                    }
                }

                return null;
            }

            // Remove object
            removeObject(object, layerIndex) {
                const layer = layerIndex !== undefined ? this.layers[layerIndex] : this.getActiveLayer();
                const index = layer.objects.indexOf(object);
                if (index !== -1) {
                    layer.objects.splice(index, 1);
                    this.redrawLayer(layerIndex !== undefined ? layerIndex : this.activeLayerIndex);
                    return true;
                }
                return false;
            }
        }

        /**
         * DRAWING OBJECT MODEL
         * Defines the object classes for different types of drawings
         */

        // Base class for all drawing objects
        class DrawingObject {
            constructor(props = {}) {
                this.x = props.x || 0;
                this.y = props.y || 0;
                this.color = props.color || '#000000';
                this.lineWidth = props.lineWidth || 2;
                this.visible = props.visible !== undefined ? props.visible : true;
                this.selected = false;
                this.type = 'drawingObject'; // Base type
            }

            draw(context) {
                // Base drawing functionality
```

```
            if (this.selected) {
                this.drawSelectionMarkers(context);
            }
        }

        drawSelectionMarkers(context) {
            // Draw selection handles (default implementation)
            context.save();
            context.strokeStyle = '#0099ff';
            context.lineWidth = 1;
            context.setLineDash([5, 3]);

            // Default is to draw a box around the object
            // This should be overridden by subclasses with specific bounds
            const bounds = this.getBounds();
            context.strokeRect(
                bounds.x - 2,
                bounds.y - 2,
                bounds.width + 4,
                bounds.height + 4
            );

            context.restore();
        }

        getBounds() {
            // Default implementation - should be overridden
            return { x: this.x, y: this.y, width: 0, height: 0 };
        }

        containsPoint(x, y) {
            // Default implementation - should be overridden
            return false;
        }

        move(dx, dy) {
            this.x += dx;
            this.y += dy;
        }

        toJSON() {
            return {
                type: this.type,
                x: this.x,
                y: this.y,
                color: this.color,
                lineWidth: this.lineWidth,
                visible: this.visible
            };
        }
```

```
885
886             static fromJSON(data) {
887                 // Factory method to create objects from JSON
888                 // This will be overridden by subclasses
889                 return new DrawingObject(data);
890             }
891         }
892
893         // Line object
894         class LineObject extends DrawingObject {
895             constructor(props = {}) {
896                 super(props);
897                 this.x2 = props.x2 || 0;
898                 this.y2 = props.y2 || 0;
899                 this.type = 'line';
900                 this.dashPattern = props.dashPattern || [];
901             }
902
903             draw(context) {
904                 context.save();
905                 context.beginPath();
906                 context.strokeStyle = this.color;
907                 context.lineWidth = this.lineWidth;
908
909                 if (this.dashPattern.length > 0) {
910                     context.setLineDash(this.dashPattern);
911                 }
912
913                 context.moveTo(this.x, this.y);
914                 context.lineTo(this.x2, this.y2);
915                 context.stroke();
916                 context.restore();
917
918                 super.draw(context);
919             }
920
921             getBounds() {
922                 const minX = Math.min(this.x, this.x2);
923                 const minY = Math.min(this.y, this.y2);
924                 const width = Math.abs(this.x2 - this.x);
925                 const height = Math.abs(this.y2 - this.y);
926
927                 return { x: minX, y: minY, width, height };
928             }
929
930             containsPoint(x, y) {
931                 // Check if point is near the line
932                 const lineLength = Math.sqrt(
933                     Math.pow(this.x2 - this.x, 2) + Math.pow(this.y2 - this.y, 2)
934                 );
```

```
935
936                 // If line is too short, use a minimum distance
937                 if (lineLength < 1) {
938                     const dx = x - this.x;
939                     const dy = y - this.y;
940                     return Math.sqrt(dx * dx + dy * dy) <= 5;
941                 }
942
943                 // Calculate distance from point to line segment
944                 const t = ((x - this.x) * (this.x2 - this.x) + (y - this.y) * (this.y2 -
    this.y)) / (lineLength * lineLength);
945
946                 if (t < 0) {
947                     // Point is beyond start point
948                     const dx = x - this.x;
949                     const dy = y - this.y;
950                     return Math.sqrt(dx * dx + dy * dy) <= 5;
951                 }
952
953                 if (t > 1) {
954                     // Point is beyond end point
955                     const dx = x - this.x2;
956                     const dy = y - this.y2;
957                     return Math.sqrt(dx * dx + dy * dy) <= 5;
958                 }
959
960                 // Calculate perpendicular distance
961                 const px = this.x + t * (this.x2 - this.x);
962                 const py = this.y + t * (this.y2 - this.y);
963                 const dx = x - px;
964                 const dy = y - py;
965                 return Math.sqrt(dx * dx + dy * dy) <= 5;
966             }
967
968             move(dx, dy) {
969                 super.move(dx, dy);
970                 this.x2 += dx;
971                 this.y2 += dy;
972             }
973
974             toJSON() {
975                 const json = super.toJSON();
976                 return {
977                     ...json,
978                     x2: this.x2,
979                     y2: this.y2,
980                     dashPattern: this.dashPattern
981                 };
982             }
983
```

```
            static fromJSON(data) {
                return new LineObject(data);
            }
        }

        // Arrow object (extends Line)
        class ArrowObject extends LineObject {
            constructor(props = {}) {
                super(props);
                this.arrowSize = props.arrowSize || 10;
                this.type = 'arrow';
            }

            draw(context) {
                // Draw the line part
                super.draw(context);

                // Draw the arrowhead
                const angle = Math.atan2(this.y2 - this.y, this.x2 - this.x);
                context.save();
                context.fillStyle = this.color;
                context.beginPath();
                context.moveTo(this.x2, this.y2);
                context.lineTo(
                    this.x2 - this.arrowSize * Math.cos(angle - Math.PI/6),
                    this.y2 - this.arrowSize * Math.sin(angle - Math.PI/6)
                );
                context.lineTo(
                    this.x2 - this.arrowSize * Math.cos(angle + Math.PI/6),
                    this.y2 - this.arrowSize * Math.sin(angle + Math.PI/6)
                );
                context.closePath();
                context.fill();
                context.restore();
            }

            toJSON() {
                const json = super.toJSON();
                return {
                    ...json,
                    arrowSize: this.arrowSize
                };
            }

            static fromJSON(data) {
                return new ArrowObject(data);
            }
        }

        // Rectangle object
```

```
1034        class RectangleObject extends DrawingObject {
1035            constructor(props = {}) {
1036                super(props);
1037                this.width = props.width || 0;
1038                this.height = props.height || 0;
1039                this.fill = props.fill || false;
1040                this.fillColor = props.fillColor || this.color;
1041                this.type = 'rectangle';
1042            }
1043
1044            draw(context) {
1045                context.save();
1046                context.strokeStyle = this.color;
1047                context.lineWidth = this.lineWidth;
1048
1049                // Draw rectangle
1050                if (this.fill) {
1051                    context.fillStyle = this.fillColor;
1052                    context.fillRect(this.x, this.y, this.width, this.height);
1053                }
1054                context.strokeRect(this.x, this.y, this.width, this.height);
1055                context.restore();
1056
1057                super.draw(context);
1058            }
1059
1060            getBounds() {
1061                return {
1062                    x: this.x,
1063                    y: this.y,
1064                    width: this.width,
1065                    height: this.height
1066                };
1067            }
1068
1069            containsPoint(x, y) {
1070                // Check if point is inside or near the edge of the rectangle
1071                if (this.fill) {
1072                    // For filled rectangles, check if point is inside
1073                    return (
1074                        x >= this.x && x <= this.x + this.width &&
1075                        y >= this.y && y <= this.y + this.height
1076                    );
1077                } else {
1078                    // For unfilled rectangles, check if point is near the edges
1079                    const nearLeft = Math.abs(x - this.x) <= 5;
1080                    const nearRight = Math.abs(x - (this.x + this.width)) <= 5;
1081                    const nearTop = Math.abs(y - this.y) <= 5;
1082                    const nearBottom = Math.abs(y - (this.y + this.height)) <= 5;
1083
```

```
                    return (
                        (nearLeft || nearRight) && (y >= this.y && y <= this.y + this.height) ||
                        (nearTop || nearBottom) && (x >= this.x && x <= this.x + this.width)
                    );
                }
            }

            toJSON() {
                const json = super.toJSON();
                return {
                    ...json,
                    width: this.width,
                    height: this.height,
                    fill: this.fill,
                    fillColor: this.fillColor
                };
            }

            static fromJSON(data) {
                return new RectangleObject(data);
            }
        }

        // Circle/Ellipse object
        class EllipseObject extends DrawingObject {
            constructor(props = {}) {
                super(props);
                this.radiusX = props.radiusX || 0;
                this.radiusY = props.radiusY || 0;
                this.fill = props.fill || false;
                this.fillColor = props.fillColor || this.color;
                this.type = 'ellipse';
            }

            draw(context) {
                context.save();
                context.beginPath();
                context.strokeStyle = this.color;
                context.lineWidth = this.lineWidth;

                // Draw ellipse
                context.ellipse(
                    this.x,
                    this.y,
                    this.radiusX,
                    this.radiusY,
                    0,
                    0,
                    2 * Math.PI
```

```
1133                );
1134
1135                if (this.fill) {
1136                    context.fillStyle = this.fillColor;
1137                    context.fill();
1138                }
1139                context.stroke();
1140                context.restore();
1141
1142                super.draw(context);
1143            }
1144
1145            getBounds() {
1146                return {
1147                    x: this.x - this.radiusX,
1148                    y: this.y - this.radiusY,
1149                    width: this.radiusX * 2,
1150                    height: this.radiusY * 2
1151                };
1152            }
1153
1154            containsPoint(x, y) {
1155                // Check if point is inside or near the edge of the ellipse
1156                const normalizedX = (x - this.x) / this.radiusX;
1157                const normalizedY = (y - this.y) / this.radiusY;
1158                const distance = Math.sqrt(normalizedX * normalizedX + normalizedY *
    normalizedY);
1159
1160                if (this.fill) {
1161                    // For filled ellipses, check if point is inside
1162                    return distance <= 1.0;
1163                } else {
1164                    // For unfilled ellipses, check if point is near the edge
1165                    return Math.abs(distance - 1.0) <= 5 / this.radiusX;
1166                }
1167            }
1168
1169            toJSON() {
1170                const json = super.toJSON();
1171                return {
1172                    ...json,
1173                    radiusX: this.radiusX,
1174                    radiusY: this.radiusY,
1175                    fill: this.fill,
1176                    fillColor: this.fillColor
1177                };
1178            }
1179
1180            static fromJSON(data) {
1181                return new EllipseObject(data);
```

```
            }
        }

        // Text object
        class TextObject extends DrawingObject {
            constructor(props = {}) {
                super(props);
                this.text = props.text || '';
                this.fontSize = props.fontSize || 14;
                this.fontFamily = props.fontFamily || 'Arial, sans-serif';
                this.align = props.align || 'left';
                this.type = 'text';
            }

            draw(context) {
                context.save();
                context.fillStyle = this.color;
                context.font = `${this.fontSize}px ${this.fontFamily}`;
                context.textAlign = this.align;

                // Draw text
                context.fillText(this.text, this.x, this.y);
                context.restore();

                super.draw(context);
            }

            getBounds() {
                // Estimate text dimensions
                const dummyCanvas = document.createElement('canvas');
                const ctx = dummyCanvas.getContext('2d');
                ctx.font = `${this.fontSize}px ${this.fontFamily}`;
                const metrics = ctx.measureText(this.text);
                const height = this.fontSize; // Approximation

                return {
                    x: this.align === 'center' ? this.x - metrics.width / 2 :
                       this.align === 'right' ? this.x - metrics.width : this.x,
                    y: this.y - height,
                    width: metrics.width,
                    height: height
                };
            }

            containsPoint(x, y) {
                const bounds = this.getBounds();
                return (
                    x >= bounds.x && x <= bounds.x + bounds.width &&
                    y >= bounds.y && y <= bounds.y + bounds.height
                );
```

```
        }

        toJSON() {
            const json = super.toJSON();
            return {
                ...json,
                text: this.text,
                fontSize: this.fontSize,
                fontFamily: this.fontFamily,
                align: this.align
            };
        }

        static fromJSON(data) {
            return new TextObject(data);
        }
    }

    // Image object
    class ImageObject extends DrawingObject {
        constructor(props = {}) {
            super(props);
            this.width = props.width || 0;
            this.height = props.height || 0;
            this.imageUrl = props.imageUrl || '';
            this.image = null;
            this.loaded = false;
            this.type = 'image';

            // Load the image if provided
            if (this.imageUrl) {
                this.loadImage(this.imageUrl);
            }
        }

        loadImage(url) {
            this.image = new Image();
            this.image.onload = () => {
                this.loaded = true;

                // If dimensions not specified, use image dimensions
                if (this.width === 0 || this.height === 0) {
                    this.width = this.image.width;
                    this.height = this.image.height;
                }

                // Trigger redraw
                document.dispatchEvent(new Event('xsheet-redraw'));
            };
            this.image.src = url;
```

```
1282            }
1283
1284            draw(context) {
1285                if (this.loaded && this.image) {
1286                    context.save();
1287                    context.drawImage(this.image, this.x, this.y, this.width, this.height);
1288                    context.restore();
1289                } else if (!this.loaded) {
1290                    // Draw placeholder while loading
1291                    context.save();
1292                    context.strokeStyle = '#999999';
1293                    context.lineWidth = 1;
1294                    context.strokeRect(this.x, this.y, this.width, this.height);
1295                    context.font = '10px Arial';
1296                    context.fillStyle = '#999999';
1297                    context.fillText('Loading Image...', this.x + 5, this.y + 15);
1298                    context.restore();
1299                }
1300
1301                super.draw(context);
1302            }
1303
1304            getBounds() {
1305                return {
1306                    x: this.x,
1307                    y: this.y,
1308                    width: this.width,
1309                    height: this.height
1310                };
1311            }
1312
1313            containsPoint(x, y) {
1314                return (
1315                    x >= this.x && x <= this.x + this.width &&
1316                    y >= this.y && y <= this.y + this.height
1317                );
1318            }
1319
1320            toJSON() {
1321                const json = super.toJSON();
1322                return {
1323                    ...json,
1324                    width: this.width,
1325                    height: this.height,
1326                    imageUrl: this.imageUrl
1327                };
1328            }
1329
1330            static fromJSON(data) {
1331                return new ImageObject(data);
```

```
            }
        }

        // Symbol object (predefined animation symbols)
        class SymbolObject extends DrawingObject {
            constructor(props = {}) {
                super(props);
                this.symbolType = props.symbolType || 'default';
                this.scale = props.scale || 1.0;
                this.type = 'symbol';
            }

            draw(context) {
                context.save();
                context.strokeStyle = this.color;
                context.fillStyle = this.color;
                context.lineWidth = this.lineWidth;

                // Draw based on symbol type
                switch (this.symbolType) {
                    case 'anticipation':
                        this.drawAnticipation(context);
                        break;
                    case 'impact':
                        this.drawImpact(context);
                        break;
                    case 'keyframe':
                        this.drawKeyframe(context);
                        break;
                    case 'inbetween':
                        this.drawInbetween(context);
                        break;
                    case 'hold':
                        this.drawHold(context);
                        break;
                    default:
                        this.drawDefault(context);
                }

                context.restore();
                super.draw(context);
            }

            drawAnticipation(context) {
                context.save();
                context.translate(this.x, this.y);
                context.scale(this.scale, this.scale);

                // Draw curved arrow going back
                context.beginPath();
```

```
                context.moveTo(0, 0);
                context.bezierCurveTo(-20, -5, -25, 10, -10, 15);
                context.stroke();

                // Draw arrowhead
                context.beginPath();
                context.moveTo(-10, 15);
                context.lineTo(-5, 10);
                context.lineTo(-15, 5);
                context.closePath();
                context.fill();

                context.restore();
            }

            drawImpact(context) {
                context.save();
                context.translate(this.x, this.y);
                context.scale(this.scale, this.scale);

                // Draw impact star
                for (let i = 0; i < 8; i++) {
                    const angle = (i / 8) * Math.PI * 2;
                    const innerRadius = 5;
                    const outerRadius = 15;

                    context.beginPath();
                    context.moveTo(
                        innerRadius * Math.cos(angle),
                        innerRadius * Math.sin(angle)
                    );
                    context.lineTo(
                        outerRadius * Math.cos(angle),
                        outerRadius * Math.sin(angle)
                    );
                    context.stroke();
                }

                context.restore();
            }

            drawKeyframe(context) {
                context.save();
                context.translate(this.x, this.y);
                context.scale(this.scale, this.scale);

                // Draw diamond
                context.beginPath();
                context.moveTo(0, -10);
                context.lineTo(10, 0);
```

```
                context.lineTo(0, 10);
                context.lineTo(-10, 0);
                context.closePath();
                context.stroke();
                context.fill();

                context.restore();
            }

            drawInbetween(context) {
                context.save();
                context.translate(this.x, this.y);
                context.scale(this.scale, this.scale);

                // Draw circle
                context.beginPath();
                context.arc(0, 0, 7, 0, Math.PI * 2);
                context.stroke();

                context.restore();
            }

            drawHold(context) {
                context.save();
                context.translate(this.x, this.y);
                context.scale(this.scale, this.scale);

                // Draw horizontal bar
                context.beginPath();
                context.moveTo(-15, 0);
                context.lineTo(15, 0);
                context.lineWidth = this.lineWidth * 2;
                context.stroke();

                context.restore();
            }

            drawDefault(context) {
                context.save();
                context.translate(this.x, this.y);
                context.scale(this.scale, this.scale);

                // Draw square
                context.beginPath();
                context.rect(-7, -7, 14, 14);
                context.stroke();

                context.restore();
            }

```

```
            getBounds() {
                // Approximate bounds based on symbol type
                const size = 20 * this.scale;
                return {
                    x: this.x - size / 2,
                    y: this.y - size / 2,
                    width: size,
                    height: size
                };
            }

            containsPoint(x, y) {
                const bounds = this.getBounds();
                const dx = x - this.x;
                const dy = y - this.y;
                const distance = Math.sqrt(dx * dx + dy * dy);

                // Use a radius-based check as most symbols are roughly circular
                return distance <= bounds.width / 2;
            }

            toJSON() {
                const json = super.toJSON();
                return {
                    ...json,
                    symbolType: this.symbolType,
                    scale: this.scale
                };
            }

            static fromJSON(data) {
                return new SymbolObject(data);
            }
        }

        // Free-form path (for pen/brush tools)
        class PathObject extends DrawingObject {
            constructor(props = {}) {
                super(props);
                this.points = props.points || [];
                this.smoothing = props.smoothing !== undefined ? props.smoothing : true;
                this.closed = props.closed || false;
                this.fill = props.fill || false;
                this.fillColor = props.fillColor || this.color;
                this.type = 'path';
            }

            addPoint(x, y) {
                this.points.push({ x, y });
            }
```

```
1532
1533            draw(context) {
1534                if (this.points.length < 2) return;
1535
1536                context.save();
1537                context.beginPath();
1538                context.strokeStyle = this.color;
1539                context.lineWidth = this.lineWidth;
1540                context.lineJoin = 'round';
1541                context.lineCap = 'round';
1542
1543                // Start from first point
1544                context.moveTo(this.points[0].x, this.points[0].y);
1545
1546                if (this.smoothing && this.points.length > 2) {
1547                    // Draw using bezier curves for smoothing
1548                    for (let i = 1; i < this.points.length - 1; i++) {
1549                        const p1 = this.points[i];
1550                        const p2 = this.points[i + 1];
1551
1552                        const xc = (p1.x + p2.x) / 2;
1553                        const yc = (p1.y + p2.y) / 2;
1554
1555                        context.quadraticCurveTo(p1.x, p1.y, xc, yc);
1556                    }
1557
1558                    // Connect to the last point
1559                    const last = this.points[this.points.length - 1];
1560                    context.lineTo(last.x, last.y);
1561                } else {
1562                    // Simple line segments
1563                    for (let i = 1; i < this.points.length; i++) {
1564                        context.lineTo(this.points[i].x, this.points[i].y);
1565                    }
1566                }
1567
1568                if (this.closed) {
1569                    context.closePath();
1570                }
1571
1572                if (this.fill) {
1573                    context.fillStyle = this.fillColor;
1574                    context.fill();
1575                }
1576
1577                context.stroke();
1578                context.restore();
1579
1580                super.draw(context);
1581            }
```

```
1582
1583            getBounds() {
1584                if (this.points.length === 0) {
1585                    return { x: this.x, y: this.y, width: 0, height: 0 };
1586                }
1587
1588                let minX = this.points[0].x;
1589                let maxX = this.points[0].x;
1590                let minY = this.points[0].y;
1591                let maxY = this.points[0].y;
1592
1593                // Find min/max coordinates
1594                for (let i = 1; i < this.points.length; i++) {
1595                    const point = this.points[i];
1596                    minX = Math.min(minX, point.x);
1597                    maxX = Math.max(maxX, point.x);
1598                    minY = Math.min(minY, point.y);
1599                    maxY = Math.max(maxY, point.y);
1600                }
1601
1602                return {
1603                    x: minX,
1604                    y: minY,
1605                    width: maxX - minX,
1606                    height: maxY - minY
1607                };
1608            }
1609
1610            containsPoint(x, y) {
1611                if (this.points.length < 2) return false;
1612
1613                // Check if point is near any line segment
1614                for (let i = 0; i < this.points.length - 1; i++) {
1615                    const p1 = this.points[i];
1616                    const p2 = this.points[i + 1];
1617
1618                    const lineLength = Math.sqrt(
1619                        Math.pow(p2.x - p1.x, 2) + Math.pow(p2.y - p1.y, 2)
1620                    );
1621
1622                    // If line is too short, check distance to point
1623                    if (lineLength < 1) {
1624                        const dx = x - p1.x;
1625                        const dy = y - p1.y;
1626                        if (Math.sqrt(dx * dx + dy * dy) <= 5) {
1627                            return true;
1628                        }
1629                        continue;
1630                    }
1631
```

```
1632                    // Calculate distance from point to line segment
1633                    const t = ((x - p1.x) * (p2.x - p1.x) + (y - p1.y) * (p2.y - p1.y)) /
(lineLength * lineLength);
1634
1635                    if (t < 0) {
1636                        // Point is beyond start point
1637                        const dx = x - p1.x;
1638                        const dy = y - p1.y;
1639                        if (Math.sqrt(dx * dx + dy * dy) <= 5) {
1640                            return true;
1641                        }
1642                    } else if (t > 1) {
1643                        // Point is beyond end point
1644                        const dx = x - p2.x;
1645                        const dy = y - p2.y;
1646                        if (Math.sqrt(dx * dx + dy * dy) <= 5) {
1647                            return true;
1648                        }
1649                    } else {
1650                        // Calculate perpendicular distance
1651                        const px = p1.x + t * (p2.x - p1.x);
1652                        const py = p1.y + t * (p2.y - p1.y);
1653                        const dx = x - px;
1654                        const dy = y - py;
1655                        if (Math.sqrt(dx * dx + dy * dy) <= 5) {
1656                            return true;
1657                        }
1658                    }
1659                }
1660
1661                // If closed and filled, also check if point is inside
1662                if (this.closed && this.fill) {
1663                    // Use point-in-polygon algorithm
1664                    let inside = false;
1665                    for (let i = 0, j = this.points.length - 1; i < this.points.length; j =
i++) {
1666                        const xi = this.points[i].x;
1667                        const yi = this.points[i].y;
1668                        const xj = this.points[j].x;
1669                        const yj = this.points[j].y;
1670
1671                        const intersect = ((yi > y) !== (yj > y)) &&
1672                            (x < (xj - xi) * (y - yi) / (yj - yi) + xi);
1673
1674                        if (intersect) inside = !inside;
1675                    }
1676                    return inside;
1677                }
1678
1679                return false;
```

```
            }

            move(dx, dy) {
                super.move(dx, dy);

                // Move all points
                this.points.forEach(point => {
                    point.x += dx;
                    point.y += dy;
                });
            }

            toJSON() {
                const json = super.toJSON();
                return {
                    ...json,
                    points: this.points,
                    smoothing: this.smoothing,
                    closed: this.closed,
                    fill: this.fill,
                    fillColor: this.fillColor
                };
            }

            static fromJSON(data) {
                return new PathObject(data);
            }
        }

        // Frame-Spanning Line (connects specific cells in the grid)
        class FrameSpanningLineObject extends DrawingObject {
            constructor(props = {}) {
                super(props);
                this.startFrame = props.startFrame || 1;
                this.startColumn = props.startColumn || 1;
                this.endFrame = props.endFrame || 1;
                this.endColumn = props.endColumn || 1;
                this.type = 'frameSpanningLine';
                this.dashPattern = props.dashPattern || [];
                this.arrowStart = props.arrowStart || false;
                this.arrowEnd = props.arrowEnd || false;
                this.arrowSize = props.arrowSize || 10;
            }

            draw(context) {
                // Calculate actual coordinates from frame and column
                const layerSystem = window.xsheetDrawing.layerSystem;

                const startPos = layerSystem.gridToCanvas(this.startFrame,
this.startColumn);
```

```
1729                const endPos = layerSystem.gridToCanvas(this.endFrame, this.endColumn);
1730
1731                if (!startPos || !endPos) return; // Skip if cells not found
1732
1733                context.save();
1734                context.beginPath();
1735                context.strokeStyle = this.color;
1736                context.lineWidth = this.lineWidth;
1737
1738                if (this.dashPattern.length > 0) {
1739                    context.setLineDash(this.dashPattern);
1740                }
1741
1742                context.moveTo(startPos.x, startPos.y);
1743                context.lineTo(endPos.x, endPos.y);
1744                context.stroke();
1745
1746                // Draw arrows if needed
1747                if (this.arrowStart) {
1748                    this.drawArrow(context, endPos.x, endPos.y, startPos.x, startPos.y);
1749                }
1750
1751                if (this.arrowEnd) {
1752                    this.drawArrow(context, startPos.x, startPos.y, endPos.x, endPos.y);
1753                }
1754
1755                context.restore();
1756
1757                // Store computed coordinates for selection/hit testing
1758                this.computedStart = startPos;
1759                this.computedEnd = endPos;
1760
1761                super.draw(context);
1762            }
1763
1764            drawArrow(context, fromX, fromY, toX, toY) {
1765                const angle = Math.atan2(toY - fromY, toX - fromX);
1766
1767                context.save();
1768                context.fillStyle = this.color;
1769                context.beginPath();
1770                context.moveTo(toX, toY);
1771                context.lineTo(
1772                    toX - this.arrowSize * Math.cos(angle - Math.PI/6),
1773                    toY - this.arrowSize * Math.sin(angle - Math.PI/6)
1774                );
1775                context.lineTo(
1776                    toX - this.arrowSize * Math.cos(angle + Math.PI/6),
1777                    toY - this.arrowSize * Math.sin(angle + Math.PI/6)
1778                );
```

```
1779                context.closePath();
1780                context.fill();
1781                context.restore();
1782            }
1783
1784            getBounds() {
1785                if (!this.computedStart || !this.computedEnd) return { x: 0, y: 0, width: 0,
    height: 0 };
1786
1787                const minX = Math.min(this.computedStart.x, this.computedEnd.x);
1788                const minY = Math.min(this.computedStart.y, this.computedEnd.y);
1789                const width = Math.abs(this.computedEnd.x - this.computedStart.x);
1790                const height = Math.abs(this.computedEnd.y - this.computedStart.y);
1791
1792                return { x: minX, y: minY, width, height };
1793            }
1794
1795            containsPoint(x, y) {
1796                if (!this.computedStart || !this.computedEnd) return false;
1797
1798                // Same algorithm as LineObject
1799                const lineLength = Math.sqrt(
1800                    Math.pow(this.computedEnd.x - this.computedStart.x, 2) +
1801                    Math.pow(this.computedEnd.y - this.computedStart.y, 2)
1802                );
1803
1804                if (lineLength < 1) {
1805                    const dx = x - this.computedStart.x;
1806                    const dy = y - this.computedStart.y;
1807                    return Math.sqrt(dx * dx + dy * dy) <= 5;
1808                }
1809
1810                const t = ((x - this.computedStart.x) * (this.computedEnd.x -
    this.computedStart.x) +
1811                           (y - this.computedStart.y) * (this.computedEnd.y -
    this.computedStart.y)) /
1812                          (lineLength * lineLength);
1813
1814                if (t < 0) {
1815                    const dx = x - this.computedStart.x;
1816                    const dy = y - this.computedStart.y;
1817                    return Math.sqrt(dx * dx + dy * dy) <= 5;
1818                }
1819
1820                if (t > 1) {
1821                    const dx = x - this.computedEnd.x;
1822                    const dy = y - this.computedEnd.y;
1823                    return Math.sqrt(dx * dx + dy * dy) <= 5;
1824                }
1825
```

```
1826                const px = this.computedStart.x + t * (this.computedEnd.x -
this.computedStart.x);
1827                const py = this.computedStart.y + t * (this.computedEnd.y -
this.computedStart.y);
1828                const dx = x - px;
1829                const dy = y - py;
1830                return Math.sqrt(dx * dx + dy * dy) <= 5;
1831            }
1832
1833            // This object type doesn't use the regular move method
1834            // Instead, it updates the frame/column values
1835
1836            toJSON() {
1837                const json = super.toJSON();
1838                return {
1839                    ...json,
1840                    startFrame: this.startFrame,
1841                    startColumn: this.startColumn,
1842                    endFrame: this.endFrame,
1843                    endColumn: this.endColumn,
1844                    dashPattern: this.dashPattern,
1845                    arrowStart: this.arrowStart,
1846                    arrowEnd: this.arrowEnd,
1847                    arrowSize: this.arrowSize
1848                };
1849            }
1850
1851            static fromJSON(data) {
1852                return new FrameSpanningLineObject(data);
1853            }
1854        }
1855
1856        // Register all object types in a factory
1857        const DrawingObjectFactory = {
1858            types: {
1859                'drawingObject': DrawingObject,
1860                'line': LineObject,
1861                'arrow': ArrowObject,
1862                'rectangle': RectangleObject,
1863                'ellipse': EllipseObject,
1864                'text': TextObject,
1865                'image': ImageObject,
1866                'symbol': SymbolObject,
1867                'path': PathObject,
1868                'frameSpanningLine': FrameSpanningLineObject
1869            },
1870
1871            createFromJSON(data) {
1872                const Type = this.types[data.type];
1873                if (Type) {
```

```
                    return Type.fromJSON(data);
                }
                return null;
            }
        };

        /**
         * DRAWING TOOL SYSTEM
         * Manages the drawing tools and interfaces with the layer system
         */
        class DrawingToolSystem {
            constructor(layerSystem) {
                this.layerSystem = layerSystem;
                this.activeTool = null;
                this.toolSettings = {
                    color: '#ff0000',
                    lineWidth: 2,
                    fill: false,
                    fillColor: '#ff8080',
                    fontSize: 16,
                    fontFamily: 'Arial, sans-serif',
                    textAlign: 'left',
                    symbolType: 'default',
                    symbolScale: 1.0
                };

                this.availableTools = {
                    'select': new SelectTool(this),
                    'pen': new PenTool(this),
                    'line': new LineTool(this),
                    'arrow': new ArrowTool(this),
                    'rectangle': new RectangleTool(this),
                    'ellipse': new EllipseTool(this),
                    'text': new TextTool(this),
                    'image': new ImageTool(this),
                    'symbol': new SymbolTool(this),
                    'frameLine': new FrameSpanningLineTool(this),
                    'eraser': new EraserTool(this)
                };

                // Default to select tool
                this.setActiveTool('select');

                // Set up toolbar UI
                this.createToolbar();
            }

            setActiveTool(toolName) {
                if (this.activeTool) {
                    this.activeTool.deactivate();
```

```
1924                }
1925
1926                if (this.availableTools[toolName]) {
1927                    this.activeTool = this.availableTools[toolName];
1928                    this.activeTool.activate();
1929                    return true;
1930                }
1931
1932                return false;
1933            }
1934
1935            createToolbar() {
1936                // Create toolbar container
1937                const toolbar = document.createElement('div');
1938                toolbar.className = 'drawing-toolbar';
1939                toolbar.style.display = 'flex';
1940                toolbar.style.flexWrap = 'wrap';
1941                toolbar.style.gap = '5px';
1942                toolbar.style.padding = '10px';
1943                toolbar.style.backgroundColor = '#f5f5f5';
1944                toolbar.style.marginBottom = '10px';
1945                toolbar.style.borderRadius = '5px';
1946
1947                // Add tool buttons
1948                this.addToolButton(toolbar, 'select', 'Select', '👆');
1949                this.addToolButton(toolbar, 'pen', 'Freehand Drawing', '🖍️');
1950                this.addToolButton(toolbar, 'line', 'Line', '—');
1951                this.addToolButton(toolbar, 'arrow', 'Arrow', '→');
1952                this.addToolButton(toolbar, 'rectangle', 'Rectangle', '□');
1953                this.addToolButton(toolbar, 'ellipse', 'Circle/Ellipse', '○');
1954                this.addToolButton(toolbar, 'text', 'Text', 'T');
1955                this.addToolButton(toolbar, 'image', 'Insert Image', '🖼️');
1956                this.addToolButton(toolbar, 'symbol', 'Animation Symbol', '⭐');
1957                this.addToolButton(toolbar, 'frameLine', 'Multi-Frame Line', '↕️');
1958
1959                // Add separator
1960                toolbar.appendChild(document.createElement('div')).style.borderLeft = '1px
    solid #ccc';
1961                toolbar.lastChild.style.height = '30px';
1962
1963                // Add color picker
1964                const colorContainer = document.createElement('div');
1965                colorContainer.style.display = 'flex';
1966                colorContainer.style.alignItems = 'center';
1967                colorContainer.style.gap = '5px';
1968
1969                const colorLabel = document.createElement('label');
1970                colorLabel.textContent = 'Color:';
1971                colorLabel.htmlFor = 'drawing-color';
1972
```

```
                const colorPicker = document.createElement('input');
                colorPicker.type = 'color';
                colorPicker.id = 'drawing-color';
                colorPicker.value = this.toolSettings.color;
                colorPicker.addEventListener('input', (e) => {
                    this.toolSettings.color = e.target.value;
                });

                colorContainer.appendChild(colorLabel);
                colorContainer.appendChild(colorPicker);
                toolbar.appendChild(colorContainer);

                // Add line width selector
                const lineWidthContainer = document.createElement('div');
                lineWidthContainer.style.display = 'flex';
                lineWidthContainer.style.alignItems = 'center';
                lineWidthContainer.style.gap = '5px';

                const lineWidthLabel = document.createElement('label');
                lineWidthLabel.textContent = 'Width:';
                lineWidthLabel.htmlFor = 'drawing-line-width';

                const lineWidthSelect = document.createElement('select');
                lineWidthSelect.id = 'drawing-line-width';

                const widths = [1, 2, 3, 5, 8, 12];
                widths.forEach(width => {
                    const option = document.createElement('option');
                    option.value = width;
                    option.textContent = width + 'px';
                    if (width === this.toolSettings.lineWidth) {
                        option.selected = true;
                    }
                    lineWidthSelect.appendChild(option);
                });

                lineWidthSelect.addEventListener('change', (e) => {
                    this.toolSettings.lineWidth = parseInt(e.target.value);
                });

                lineWidthContainer.appendChild(lineWidthLabel);
                lineWidthContainer.appendChild(lineWidthSelect);
                toolbar.appendChild(lineWidthContainer);

                // Add fill option
                const fillContainer = document.createElement('div');
                fillContainer.style.display = 'flex';
                fillContainer.style.alignItems = 'center';
                fillContainer.style.gap = '5px';

```

```
                const fillCheck = document.createElement('input');
                fillCheck.type = 'checkbox';
                fillCheck.id = 'drawing-fill';
                fillCheck.checked = this.toolSettings.fill;

                const fillLabel = document.createElement('label');
                fillLabel.textContent = 'Fill';
                fillLabel.htmlFor = 'drawing-fill';

                fillCheck.addEventListener('change', (e) => {
                    this.toolSettings.fill = e.target.checked;
                    fillColorPicker.disabled = !e.target.checked;
                });

                const fillColorPicker = document.createElement('input');
                fillColorPicker.type = 'color';
                fillColorPicker.id = 'drawing-fill-color';
                fillColorPicker.value = this.toolSettings.fillColor;
                fillColorPicker.disabled = !this.toolSettings.fill;

                fillColorPicker.addEventListener('input', (e) => {
                    this.toolSettings.fillColor = e.target.value;
                });

                fillContainer.appendChild(fillCheck);
                fillContainer.appendChild(fillLabel);
                fillContainer.appendChild(fillColorPicker);
                toolbar.appendChild(fillContainer);

                // Add separator
                toolbar.appendChild(document.createElement('div')).style.borderLeft = '1px
solid #ccc';
                toolbar.lastChild.style.height = '30px';

                // Add layer selector
                const layerSelector = document.createElement('select');
                layerSelector.id = 'drawing-layer-selector';

                // Add options for each layer
                this.layerSystem.layers.forEach((layer, index) => {
                    const option = document.createElement('option');
                    option.value = index;
                    option.textContent = layer.name;
                    if (index === this.layerSystem.activeLayerIndex) {
                        option.selected = true;
                    }
                    layerSelector.appendChild(option);
                });

                layerSelector.addEventListener('change', (e) => {
```

```
                    this.layerSystem.setActiveLayer(parseInt(e.target.value));
                });

                const layerLabel = document.createElement('label');
                layerLabel.textContent = 'Layer:';
                layerLabel.htmlFor = 'drawing-layer-selector';
                layerLabel.style.marginRight = '5px';

                toolbar.appendChild(layerLabel);
                toolbar.appendChild(layerSelector);

                // Add separator
                toolbar.appendChild(document.createElement('div')).style.borderLeft = '1px solid #ccc';
                toolbar.lastChild.style.height = '30px';

                // Add eraser tool
                this.addToolButton(toolbar, 'eraser', 'Eraser', '🧹');

                // Add clear button
                const clearButton = document.createElement('button');
                clearButton.textContent = 'Clear All Drawings';
                clearButton.style.backgroundColor = '#ff5555';
                clearButton.style.color = 'white';
                clearButton.style.border = 'none';
                clearButton.style.borderRadius = '4px';
                clearButton.style.padding = '5px 10px';
                clearButton.style.cursor = 'pointer';

                clearButton.addEventListener('click', () => {
                    if (confirm('Are you sure you want to clear all drawings?')) {
                        this.layerSystem.clearAllLayers();
                    }
                });

                toolbar.appendChild(clearButton);

                // Find controls div and add toolbar before it
                const controls = document.querySelector('.controls');
                if (controls) {
                    controls.parentNode.insertBefore(toolbar, controls.nextSibling);
                } else {
                    document.body.insertBefore(toolbar, document.body.firstChild);
                }
            }

            addToolButton(toolbar, toolName, tooltip, icon) {
                const button = document.createElement('button');
                button.textContent = icon;
                button.title = tooltip;
```

```
                button.style.width = '36px';
                button.style.height = '36px';
                button.style.fontSize = '16px';
                button.style.margin = '0';
                button.style.padding = '5px';
                button.style.borderRadius = '4px';
                button.style.border = '1px solid #ccc';
                button.style.backgroundColor = 'white';
                button.style.cursor = 'pointer';

                button.addEventListener('click', () => {
                    this.setActiveTool(toolName);

                    // Update active button styling
                    document.querySelectorAll('.drawing-toolbar button').forEach(btn => {
                        btn.style.backgroundColor = 'white';
                        btn.style.color = 'black';
                    });

                    button.style.backgroundColor = '#4CAF50';
                    button.style.color = 'white';
                });

                // Set active state for default tool
                if (toolName === 'select') {
                    button.style.backgroundColor = '#4CAF50';
                    button.style.color = 'white';
                }

                toolbar.appendChild(button);
            }
        }

        /**
         * DRAWING TOOLS
         * Individual tool implementations
         */

        // Base tool class
        class DrawingTool {
            constructor(toolSystem) {
                this.toolSystem = toolSystem;
                this.layerSystem = toolSystem.layerSystem;
                this.active = false;
                this.settings = toolSystem.toolSettings;
            }

            activate() {
                this.active = true;
                this.layerSystem.enableDrawing();
```

```
            this.attachEvents();
        }

        deactivate() {
            this.active = false;
            // turn pointer-events back off so clicks go to the table cells
            this.toolSystem.layerSystem.disableDrawing();
            this.detachEvents();
        }

        attachEvents() {
            // Override in subclasses
        }

        detachEvents() {
            // Override in subclasses
        }
    }

    // Select tool for manipulating objects
    class SelectTool extends DrawingTool {

        constructor(toolSystem) {
            super(toolSystem);
            this.selectedObject = null;
            this.selectedLayer = null;
            this.dragging = false;
            this.dragStart = { x: 0, y: 0 };
            this.objectStart = { x: 0, y: 0 };
        }
        activate() {
            this.active = true;
            // let pointer-events go to the table cells instead of the canvases
            this.layerSystem.disableDrawing();
            this.attachEvents();
        }

        deactivate() {
            this.active = false;
            this.layerSystem.disableDrawing();
            this.detachEvents();
        }
        attachEvents() {
            const canvas = this.layerSystem.getActiveLayer().canvas;

            canvas.addEventListener('mousedown', this.handleMouseDown);
            canvas.addEventListener('pointerdown', this.handleMouseDown);
            document.addEventListener('mousemove', this.handleMouseMove);
            document.addEventListener('pointermove', this.handleMouseMove);
            document.addEventListener('mouseup', this.handleMouseUp);
```

```
2221                document.addEventListener('pointerup',   this.handleMouseUp);
2222                document.addEventListener('keydown', this.handleKeyDown);
2223            }
2224
2225            detachEvents() {
2226                const canvas = this.layerSystem.getActiveLayer().canvas;
2227
2228                canvas.removeEventListener('mousedown', this.handleMouseDown);
2229                canvas.removeEventListener('pointerdown', this.handleMouseDown);
2230                document.removeEventListener('mousemove', this.handleMouseMove);
2231                document.removeEventListener('pointermove', this.handleMouseMove);
2232                document.removeEventListener('mouseup', this.handleMouseUp);
2233                document.removeEventListener('pointerup',   this.handleMouseUp);
2234                document.removeEventListener('keydown', this.handleKeyDown);
2235
2236                // Clear selection
2237                this.clearSelection();
2238            }
2239
2240            handleMouseDown = (e) => {
2241                // Convert to canvas coordinates
2242                const coords = this.layerSystem.screenToCanvas(e.clientX, e.clientY);
2243
2244                // Check if clicked on an object
2245                const hit = this.layerSystem.findObjectAt(coords.x, coords.y);
2246
2247                if (hit) {
2248                    // Select the object
2249                    this.selectObject(hit.object, hit.layerIndex);
2250
2251                    // Start drag
2252                    this.dragging = true;
2253                    this.dragStart = { x: coords.x, y: coords.y };
2254                    this.objectStart = { x: hit.object.x, y: hit.object.y };
2255                } else {
2256                    // Clear selection if clicked empty space
2257                    this.clearSelection();
2258                }
2259            }
2260
2261            handleMouseMove = (e) => {
2262                if (!this.dragging || !this.selectedObject) return;
2263
2264                const coords = this.layerSystem.screenToCanvas(e.clientX, e.clientY);
2265
2266                // Calculate move distance
2267                const dx = coords.x - this.dragStart.x;
2268                const dy = coords.y - this.dragStart.y;
2269
2270                // Move the selected object
```

```
                this.selectedObject.x = this.objectStart.x + dx;
                this.selectedObject.y = this.objectStart.y + dy;

                // For objects with special move handling
                this.selectedObject.move(dx, dy);
                this.selectedObject.x = this.objectStart.x; // Reset x as move() already handled it
                this.selectedObject.y = this.objectStart.y; // Reset y as move() already handled it

                // Redraw
                this.layerSystem.redrawAll();
            }

            handleMouseUp = () => {
                this.dragging = false;
            }

            handleKeyDown = (e) => {
                if (!this.selectedObject) return;

                // Delete key
                if (e.key === 'Delete' || e.key === 'Backspace') {
                    this.layerSystem.removeObject(this.selectedObject, this.selectedLayer);
                    this.clearSelection();
                }

                // Arrow keys for fine movement
                const moveDistance = e.shiftKey ? 10 : 1;

                if (e.key === 'ArrowLeft') {
                    this.selectedObject.move(-moveDistance, 0);
                    this.layerSystem.redrawAll();
                } else if (e.key === 'ArrowRight') {
                    this.selectedObject.move(moveDistance, 0);
                    this.layerSystem.redrawAll();
                } else if (e.key === 'ArrowUp') {
                    this.selectedObject.move(0, -moveDistance);
                    this.layerSystem.redrawAll();
                } else if (e.key === 'ArrowDown') {
                    this.selectedObject.move(0, moveDistance);
                    this.layerSystem.redrawAll();
                }
            }

            selectObject(object, layerIndex) {
                // Clear previous selection
                this.clearSelection();

                // Set new selection
```

```
                this.selectedObject = object;
                this.selectedLayer = layerIndex;
                object.selected = true;

                // Redraw with selection visual
                this.layerSystem.redrawAll();
            }

            clearSelection() {
                if (this.selectedObject) {
                    this.selectedObject.selected = false;
                    this.selectedObject = null;
                    this.selectedLayer = null;
                    this.layerSystem.redrawAll();
                }
            }
        }

        // Pen tool for free drawing
        class PenTool extends DrawingTool {
            constructor(toolSystem) {
                super(toolSystem);
                this.currentPath = null;
                this.drawing = false;
            }

            attachEvents() {
                const canvas = this.layerSystem.getActiveLayer().canvas;

                canvas.addEventListener('mousedown', this.handleMouseDown);
                canvas.addEventListener('pointerdown', this.handleMouseDown);
                document.addEventListener('mousemove', this.handleMouseMove);
                document.addEventListener('pointermove', this.handleMouseMove);
                document.addEventListener('mouseup', this.handleMouseUp);
                document.addEventListener('pointerup',   this.handleMouseUp);
            }

            detachEvents() {
                const canvas = this.layerSystem.getActiveLayer().canvas;

                canvas.removeEventListener('mousedown', this.handleMouseDown);
                canvas.removeEventListener('pointerdown', this.handleMouseDown);
                document.removeEventListener('mousemove', this.handleMouseMove);
                document.removeEventListener('pointermove', this.handleMouseMove);
                document.removeEventListener('mouseup', this.handleMouseUp);
                document.removeEventListener('pointerup',   this.handleMouseUp);

                // Finish any in-progress drawing
                this.finishDrawing();
            }
```

```
2369
2370            handleMouseDown = (e) => {
2371                e.preventDefault();
2372                if (e.pointerId != null) {
2373                e.target.setPointerCapture(e.pointerId);
2374                }
2375
2376                const coords = this.layerSystem.screenToCanvas(e.clientX, e.clientY);
2377
2378                // Start a new path
2379                this.currentPath = new PathObject({
2380                    color: this.settings.color,
2381                    lineWidth: this.settings.lineWidth,
2382                    x: coords.x,
2383                    y: coords.y
2384                });
2385
2386                this.currentPath.addPoint(coords.x, coords.y);
2387                this.drawing = true;
2388
2389                // Add to layer
2390                this.layerSystem.addObject(this.currentPath);
2391            }
2392
2393            handleMouseMove = (e) => {
2394                if (!this.drawing || !this.currentPath) return;
2395
2396                const coords = this.layerSystem.screenToCanvas(e.clientX, e.clientY);
2397
2398                // Add point to path
2399                this.currentPath.addPoint(coords.x, coords.y);
2400
2401                // Redraw
2402                this.layerSystem.redrawLayer(this.layerSystem.activeLayerIndex);
2403            }
2404
2405            handleMouseUp = (e) => {
2406                if (e.pointerId != null) {
2407                  e.target.releasePointerCapture(e.pointerId);
2408                }
2409                this.finishDrawing();
2410            }
2411
2412            finishDrawing() {
2413                if (this.drawing && this.currentPath) {
2414                    // Finish the path
2415                    this.drawing = false;
2416                    this.currentPath = null;
2417                }
2418            }
```

```
        }

        // Line tool
        class LineTool extends DrawingTool {
            constructor(toolSystem) {
                super(toolSystem);
                this.startPoint = null;
                this.currentLine = null;
                this.drawing = false;
            }

            attachEvents() {
                const canvas = this.layerSystem.getActiveLayer().canvas;

                canvas.addEventListener('mousedown', this.handleMouseDown);
                document.addEventListener('mousemove', this.handleMouseMove);
                document.addEventListener('mouseup', this.handleMouseUp);
            }

            detachEvents() {
                const canvas = this.layerSystem.getActiveLayer().canvas;

                canvas.removeEventListener('mousedown', this.handleMouseDown);
                document.removeEventListener('mousemove', this.handleMouseMove);
                document.removeEventListener('mouseup', this.handleMouseUp);

                // Finish any in-progress drawing
                this.finishDrawing();
            }

            handleMouseDown = (e) => {
                const coords = this.layerSystem.screenToCanvas(e.clientX, e.clientY);

                this.startPoint = coords;

                // Create temporary line
                this.currentLine = new LineObject({
                    x: coords.x,
                    y: coords.y,
                    x2: coords.x,
                    y2: coords.y,
                    color: this.settings.color,
                    lineWidth: this.settings.lineWidth
                });

                this.drawing = true;

                // Add to layer
                this.layerSystem.addObject(this.currentLine);
            }
```

```
2469
2470            handleMouseMove = (e) => {
2471                if (!this.drawing || !this.currentLine) return;
2472
2473                const coords = this.layerSystem.screenToCanvas(e.clientX, e.clientY);
2474
2475                // Update end point
2476                this.currentLine.x2 = coords.x;
2477                this.currentLine.y2 = coords.y;
2478
2479                // Redraw
2480                this.layerSystem.redrawLayer(this.layerSystem.activeLayerIndex);
2481            }
2482
2483            handleMouseUp = () => {
2484                this.finishDrawing();
2485            }
2486
2487            finishDrawing() {
2488                if (this.drawing && this.currentLine) {
2489                    // Finish the line
2490                    this.drawing = false;
2491                    this.startPoint = null;
2492                    this.currentLine = null;
2493                }
2494            }
2495        }
2496
2497        // Arrow tool (extends Line tool)
2498        class ArrowTool extends LineTool {
2499            handleMouseDown = (e) => {
2500                const coords = this.layerSystem.screenToCanvas(e.clientX, e.clientY);
2501
2502                this.startPoint = coords;
2503
2504                // Create temporary arrow
2505                this.currentLine = new ArrowObject({
2506                    x: coords.x,
2507                    y: coords.y,
2508                    x2: coords.x,
2509                    y2: coords.y,
2510                    color: this.settings.color,
2511                    lineWidth: this.settings.lineWidth
2512                });
2513
2514                this.drawing = true;
2515
2516                // Add to layer
2517                this.layerSystem.addObject(this.currentLine);
2518            }
```

```
        }

        // Rectangle tool
        class RectangleTool extends DrawingTool {
            constructor(toolSystem) {
                super(toolSystem);
                this.startPoint = null;
                this.currentRect = null;
                this.drawing = false;
            }

            attachEvents() {
                const canvas = this.layerSystem.getActiveLayer().canvas;

                canvas.addEventListener('mousedown', this.handleMouseDown);
                document.addEventListener('mousemove', this.handleMouseMove);
                document.addEventListener('mouseup', this.handleMouseUp);
            }

            detachEvents() {
                const canvas = this.layerSystem.getActiveLayer().canvas;

                canvas.removeEventListener('mousedown', this.handleMouseDown);
                document.removeEventListener('mousemove', this.handleMouseMove);
                document.removeEventListener('mouseup', this.handleMouseUp);

                // Finish any in-progress drawing
                this.finishDrawing();
            }

            handleMouseDown = (e) => {
                const coords = this.layerSystem.screenToCanvas(e.clientX, e.clientY);

                this.startPoint = coords;

                // Create temporary rectangle
                this.currentRect = new RectangleObject({
                    x: coords.x,
                    y: coords.y,
                    width: 0,
                    height: 0,
                    color: this.settings.color,
                    lineWidth: this.settings.lineWidth,
                    fill: this.settings.fill,
                    fillColor: this.settings.fillColor
                });

                this.drawing = true;

                // Add to layer
```

```
                this.layerSystem.addObject(this.currentRect);
            }

            handleMouseMove = (e) => {
                if (!this.drawing || !this.currentRect) return;

                const coords = this.layerSystem.screenToCanvas(e.clientX, e.clientY);

                // Update dimensions
                const width = coords.x - this.startPoint.x;
                const height = coords.y - this.startPoint.y;

                if (width < 0) {
                    this.currentRect.x = coords.x;
                    this.currentRect.width = Math.abs(width);
                } else {
                    this.currentRect.x = this.startPoint.x;
                    this.currentRect.width = width;
                }

                if (height < 0) {
                    this.currentRect.y = coords.y;
                    this.currentRect.height = Math.abs(height);
                } else {
                    this.currentRect.y = this.startPoint.y;
                    this.currentRect.height = height;
                }

                // Redraw
                this.layerSystem.redrawLayer(this.layerSystem.activeLayerIndex);
            }

            handleMouseUp = () => {
                this.finishDrawing();
            }

            finishDrawing() {
                if (this.drawing && this.currentRect) {
                    // Finish the rectangle
                    this.drawing = false;
                    this.startPoint = null;
                    this.currentRect = null;
                }
            }
        }

        // Ellipse tool
        class EllipseTool extends DrawingTool {
            constructor(toolSystem) {
                super(toolSystem);
```

```
2619                this.center = null;
2620                this.currentEllipse = null;
2621                this.drawing = false;
2622            }
2623
2624            attachEvents() {
2625                const canvas = this.layerSystem.getActiveLayer().canvas;
2626
2627                canvas.addEventListener('mousedown', this.handleMouseDown);
2628                document.addEventListener('mousemove', this.handleMouseMove);
2629                document.addEventListener('mouseup', this.handleMouseUp);
2630            }
2631
2632            detachEvents() {
2633                const canvas = this.layerSystem.getActiveLayer().canvas;
2634
2635                canvas.removeEventListener('mousedown', this.handleMouseDown);
2636                document.removeEventListener('mousemove', this.handleMouseMove);
2637                document.removeEventListener('mouseup', this.handleMouseUp);
2638
2639                // Finish any in-progress drawing
2640                this.finishDrawing();
2641            }
2642
2643            handleMouseDown = (e) => {
2644                const coords = this.layerSystem.screenToCanvas(e.clientX, e.clientY);
2645
2646                this.center = coords;
2647
2648                // Create temporary ellipse
2649                this.currentEllipse = new EllipseObject({
2650                    x: coords.x,
2651                    y: coords.y,
2652                    radiusX: 0,
2653                    radiusY: 0,
2654                    color: this.settings.color,
2655                    lineWidth: this.settings.lineWidth,
2656                    fill: this.settings.fill,
2657                    fillColor: this.settings.fillColor
2658                });
2659
2660                this.drawing = true;
2661
2662                // Add to layer
2663                this.layerSystem.addObject(this.currentEllipse);
2664            }
2665
2666            handleMouseMove = (e) => {
2667                if (!this.drawing || !this.currentEllipse) return;
2668
```

```
2669                const coords = this.layerSystem.screenToCanvas(e.clientX, e.clientY);
2670
2671                // Update radii
2672                this.currentEllipse.radiusX = Math.abs(coords.x - this.center.x);
2673                this.currentEllipse.radiusY = Math.abs(coords.y - this.center.y);
2674
2675                // Redraw
2676                this.layerSystem.redrawLayer(this.layerSystem.activeLayerIndex);
2677            }
2678
2679            handleMouseUp = () => {
2680                this.finishDrawing();
2681            }
2682
2683            finishDrawing() {
2684                if (this.drawing && this.currentEllipse) {
2685                    // Finish the ellipse
2686                    this.drawing = false;
2687                    this.center = null;
2688                    this.currentEllipse = null;
2689                }
2690            }
2691        }
2692
2693        // Text tool
2694        class TextTool extends DrawingTool {
2695            constructor(toolSystem) {
2696                super(toolSystem);
2697                this.textInput = null;
2698            }
2699
2700            attachEvents() {
2701                const canvas = this.layerSystem.getActiveLayer().canvas;
2702
2703                canvas.addEventListener('click', this.handleClick);
2704            }
2705
2706            detachEvents() {
2707                const canvas = this.layerSystem.getActiveLayer().canvas;
2708
2709                canvas.removeEventListener('click', this.handleClick);
2710
2711                // Remove any active text input
2712                this.removeTextInput();
2713            }
2714
2715            handleClick = (e) => {
2716                const coords = this.layerSystem.screenToCanvas(e.clientX, e.clientY);
2717
2718                // Show text input at click position
```

```
2719                this.showTextInput(coords.x, coords.y);
2720            }
2721
2722            showTextInput(x, y) {
2723                // Remove any existing text input
2724                this.removeTextInput();
2725
2726                // Create text input element
2727                this.textInput = document.createElement('div');
2728                this.textInput.style.position = 'absolute';
2729                this.textInput.style.zIndex = '100';
2730
2731                // Position relative to canvas
2732                const canvasRect = this.layerSystem.container.getBoundingClientRect();
2733                this.textInput.style.left = (canvasRect.left + x) + 'px';
2734                this.textInput.style.top = (canvasRect.top + y) + 'px';
2735
2736                // Style the input
2737                this.textInput.style.backgroundColor = 'white';
2738                this.textInput.style.border = '1px solid #ccc';
2739                this.textInput.style.padding = '5px';
2740                this.textInput.style.borderRadius = '3px';
2741                this.textInput.style.boxShadow = '0 2px 5px rgba(0,0,0,0.1)';
2742
2743                // Create the actual input
2744                const input = document.createElement('input');
2745                input.type = 'text';
2746                input.placeholder = 'Enter text...';
2747                input.style.width = '200px';
2748                input.style.padding = '5px';
2749                input.style.border = '1px solid #ddd';
2750                input.style.borderRadius = '3px';
2751
2752                // Create font size selector
2753                const sizeSelect = document.createElement('select');
2754                [8, 10, 12, 14, 16, 18, 20, 24, 36].forEach(size => {
2755                    const option = document.createElement('option');
2756                    option.value = size;
2757                    option.textContent = size + 'px';
2758                    if (size === this.settings.fontSize) {
2759                        option.selected = true;
2760                    }
2761                    sizeSelect.appendChild(option);
2762                });
2763
2764                sizeSelect.addEventListener('change', (e) => {
2765                    this.settings.fontSize = parseInt(e.target.value);
2766                });
2767
2768                // Create alignment options
```

```
                const alignmentDiv = document.createElement('div');
                alignmentDiv.style.display = 'flex';
                alignmentDiv.style.marginTop = '5px';

                ['left', 'center', 'right'].forEach(align => {
                    const button = document.createElement('button');
                    button.textContent = align[0].toUpperCase();
                    button.style.flex = '1';
                    button.style.padding = '2px 5px';
                    button.style.backgroundColor = this.settings.textAlign === align ? '#4CAF50' : '#f1f1f1';
                    button.style.color = this.settings.textAlign === align ? 'white' : 'black';
                    button.style.border = '1px solid #ccc';
                    button.style.cursor = 'pointer';

                    button.addEventListener('click', () => {
                        this.settings.textAlign = align;
                        alignmentDiv.querySelectorAll('button').forEach(btn => {
                            btn.style.backgroundColor = '#f1f1f1';
                            btn.style.color = 'black';
                        });
                        button.style.backgroundColor = '#4CAF50';
                        button.style.color = 'white';
                    });

                    alignmentDiv.appendChild(button);
                });

                // Add UI elements to the container
                this.textInput.appendChild(input);
                this.textInput.appendChild(document.createElement('br'));
                this.textInput.appendChild(document.createTextNode('Size: '));
                this.textInput.appendChild(sizeSelect);
                this.textInput.appendChild(document.createElement('br'));
                this.textInput.appendChild(alignmentDiv);

                // Add buttons container
                const buttons = document.createElement('div');
                buttons.style.display = 'flex';
                buttons.style.marginTop = '5px';
                buttons.style.gap = '5px';

                // Add button
                const addButton = document.createElement('button');
                addButton.textContent = 'Add Text';
                addButton.style.flex = '1';
                addButton.style.padding = '5px';
                addButton.style.backgroundColor = '#4CAF50';
                addButton.style.color = 'white';
```

```
            addButton.style.border = 'none';
            addButton.style.borderRadius = '3px';
            addButton.style.cursor = 'pointer';

            addButton.addEventListener('click', () => {
                this.addText(x, y, input.value);
            });

            // Cancel button
            const cancelButton = document.createElement('button');
            cancelButton.textContent = 'Cancel';
            cancelButton.style.flex = '1';
            cancelButton.style.padding = '5px';
            cancelButton.style.backgroundColor = '#f44336';
            cancelButton.style.color = 'white';
            cancelButton.style.border = 'none';
            cancelButton.style.borderRadius = '3px';
            cancelButton.style.cursor = 'pointer';

            cancelButton.addEventListener('click', () => {
                this.removeTextInput();
            });

            buttons.appendChild(addButton);
            buttons.appendChild(cancelButton);
            this.textInput.appendChild(buttons);

            // Add to document
            document.body.appendChild(this.textInput);

            // Focus the input
            input.focus();

            // Handle enter key
            input.addEventListener('keydown', (e) => {
                if (e.key === 'Enter') {
                    this.addText(x, y, input.value);
                } else if (e.key === 'Escape') {
                    this.removeTextInput();
                }
            });
        }

        removeTextInput() {
            if (this.textInput && this.textInput.parentNode) {
                this.textInput.parentNode.removeChild(this.textInput);
                this.textInput = null;
            }
        }

```

```
            addText(x, y, text) {
                if (!text.trim()) {
                    this.removeTextInput();
                    return;
                }

                // Create text object
                const textObj = new TextObject({
                    x: x,
                    y: y,
                    text: text,
                    color: this.settings.color,
                    fontSize: this.settings.fontSize,
                    fontFamily: this.settings.fontFamily,
                    align: this.settings.textAlign
                });

                // Add to layer
                this.layerSystem.addObject(textObj);

                // Remove text input
                this.removeTextInput();
            }
        }

        // Image tool
        class ImageTool extends DrawingTool {
            constructor(toolSystem) {
                super(toolSystem);
                this.fileInput = null;
            }

            attachEvents() {
                const canvas = this.layerSystem.getActiveLayer().canvas;

                canvas.addEventListener('click', this.handleClick);
            }

            detachEvents() {
                const canvas = this.layerSystem.getActiveLayer().canvas;

                canvas.removeEventListener('click', this.handleClick);
            }

            handleClick = (e) => {
                // Show file upload dialog
                this.showFileDialog();
            }

            showFileDialog() {
```

```
2917                // Create hidden file input if it doesn't exist
2918                if (!this.fileInput) {
2919                    this.fileInput = document.createElement('input');
2920                    this.fileInput.type = 'file';
2921                    this.fileInput.accept = 'image/*';
2922                    this.fileInput.style.display = 'none';
2923                    document.body.appendChild(this.fileInput);
2924
2925                    this.fileInput.addEventListener('change', (e) => {
2926                        if (e.target.files && e.target.files[0]) {
2927                            this.handleFileSelect(e.target.files[0]);
2928                        }
2929                    });
2930                }
2931
2932                // Trigger file dialog
2933                this.fileInput.click();
2934            }
2935
2936            handleFileSelect(file) {
2937                // Read the file and create a data URL
2938                const reader = new FileReader();
2939
2940                reader.onload = (e) => {
2941                    const imageUrl = e.target.result;
2942
2943                    // Show image placement UI
2944                    this.showImagePlacementUI(imageUrl);
2945                };
2946
2947                reader.readAsDataURL(file);
2948            }
2949
2950            showImagePlacementUI(imageUrl) {
2951                // Create a preview image to get dimensions
2952                const img = new Image();
2953
2954                img.onload = () => {
2955                    // Calculate dimensions (max size 300px width/height while maintaining
    aspect ratio)
2956                    let width = img.width;
2957                    let height = img.height;
2958
2959                    const maxSize = 300;
2960                    if (width > maxSize || height > maxSize) {
2961                        if (width > height) {
2962                            height = (height / width) * maxSize;
2963                            width = maxSize;
2964                        } else {
2965                            width = (width / height) * maxSize;
```

```
                        height = maxSize;
                    }
                }

                // Create placement UI
                const placementUI = document.createElement('div');
                placementUI.style.position = 'fixed';
                placementUI.style.top = '50%';
                placementUI.style.left = '50%';
                placementUI.style.transform = 'translate(-50%, -50%)';
                placementUI.style.backgroundColor = 'white';
                placementUI.style.padding = '20px';
                placementUI.style.borderRadius = '5px';
                placementUI.style.boxShadow = '0 0 10px rgba(0,0,0,0.3)';
                placementUI.style.zIndex = '1000';

                // Add heading
                const heading = document.createElement('h3');
                heading.textContent = 'Place Image';
                heading.style.margin = '0 0 10px 0';

                // Add image preview
                const preview = document.createElement('img');
                preview.src = imageUrl;
                preview.style.maxWidth = '300px';
                preview.style.maxHeight = '300px';
                preview.style.display = 'block';
                preview.style.marginBottom = '10px';

                // Size controls
                const sizeControls = document.createElement('div');
                sizeControls.style.marginBottom = '10px';

                const widthLabel = document.createElement('label');
                widthLabel.textContent = 'Width: ';
                const widthInput = document.createElement('input');
                widthInput.type = 'number';
                widthInput.value = Math.round(width);
                widthInput.style.width = '60px';

                const heightLabel = document.createElement('label');
                heightLabel.textContent = 'Height: ';
                heightLabel.style.marginLeft = '10px';
                const heightInput = document.createElement('input');
                heightInput.type = 'number';
                heightInput.value = Math.round(height);
                heightInput.style.width = '60px';

                // Maintain aspect ratio
                const aspectRatio = img.width / img.height;
```

```
3016
3017                    widthInput.addEventListener('input', () => {
3018                        const newWidth = parseInt(widthInput.value);
3019                        if (!isNaN(newWidth)) {
3020                            heightInput.value = Math.round(newWidth / aspectRatio);
3021                        }
3022                    });
3023
3024                    heightInput.addEventListener('input', () => {
3025                        const newHeight = parseInt(heightInput.value);
3026                        if (!isNaN(newHeight)) {
3027                            widthInput.value = Math.round(newHeight * aspectRatio);
3028                        }
3029                    });
3030
3031                    sizeControls.appendChild(widthLabel);
3032                    sizeControls.appendChild(widthInput);
3033                    sizeControls.appendChild(heightLabel);
3034                    sizeControls.appendChild(heightInput);
3035
3036                    // Buttons container
3037                    const buttons = document.createElement('div');
3038                    buttons.style.display = 'flex';
3039                    buttons.style.justifyContent = 'space-between';
3040                    buttons.style.marginTop = '15px';
3041
3042                    // Place button
3043                    const placeButton = document.createElement('button');
3044                    placeButton.textContent = 'Place Image';
3045                    placeButton.style.padding = '8px 15px';
3046                    placeButton.style.backgroundColor = '#4CAF50';
3047                    placeButton.style.color = 'white';
3048                    placeButton.style.border = 'none';
3049                    placeButton.style.borderRadius = '4px';
3050                    placeButton.style.cursor = 'pointer';
3051
3052                    placeButton.addEventListener('click', () => {
3053                        // Get center of view as placement position
3054                        const containerRect =
    this.layerSystem.container.getBoundingClientRect();
3055                        const x = containerRect.width / 2;
3056                        const y = containerRect.height / 2;
3057
3058                        // Get dimensions from inputs
3059                        const finalWidth = parseInt(widthInput.value);
3060                        const finalHeight = parseInt(heightInput.value);
3061
3062                        // Create and add image object
3063                        const imageObj = new ImageObject({
3064                            x: x - finalWidth / 2,
```

```
                            y: y - finalHeight / 2,
                            width: finalWidth,
                            height: finalHeight,
                            imageUrl: imageUrl
                        });

                        this.layerSystem.addObject(imageObj);

                        // Remove UI
                        document.body.removeChild(placementUI);
                    });

                    // Cancel button
                    const cancelButton = document.createElement('button');
                    cancelButton.textContent = 'Cancel';
                    cancelButton.style.padding = '8px 15px';
                    cancelButton.style.backgroundColor = '#f44336';
                    cancelButton.style.color = 'white';
                    cancelButton.style.border = 'none';
                    cancelButton.style.borderRadius = '4px';
                    cancelButton.style.cursor = 'pointer';

                    cancelButton.addEventListener('click', () => {
                        document.body.removeChild(placementUI);
                    });

                    buttons.appendChild(cancelButton);
                    buttons.appendChild(placeButton);

                    // Assemble UI
                    placementUI.appendChild(heading);
                    placementUI.appendChild(preview);
                    placementUI.appendChild(sizeControls);
                    placementUI.appendChild(buttons);

                    // Add to document
                    document.body.appendChild(placementUI);
                };

                img.src = imageUrl;
            }
        }

        // Symbol tool
        class SymbolTool extends DrawingTool {
            constructor(toolSystem) {
                super(toolSystem);
                this.symbolSelector = null;
            }

```

```
3115            attachEvents() {
3116                const canvas = this.layerSystem.getActiveLayer().canvas;
3117
3118                canvas.addEventListener('click', this.handleClick);
3119            }
3120
3121            detachEvents() {
3122                const canvas = this.layerSystem.getActiveLayer().canvas;
3123
3124                canvas.removeEventListener('click', this.handleClick);
3125
3126                // Remove any active symbol selector
3127                this.removeSymbolSelector();
3128            }
3129
3130            handleClick = (e) => {
3131                const coords = this.layerSystem.screenToCanvas(e.clientX, e.clientY);
3132
3133                // Show symbol selector
3134                this.showSymbolSelector(coords.x, coords.y);
3135            }
3136
3137            showSymbolSelector(x, y) {
3138                // Remove existing selector if any
3139                this.removeSymbolSelector();
3140
3141                // Create symbol selector
3142                this.symbolSelector = document.createElement('div');
3143                this.symbolSelector.style.position = 'absolute';
3144                this.symbolSelector.style.zIndex = '100';
3145
3146                // Position relative to canvas
3147                const canvasRect = this.layerSystem.container.getBoundingClientRect();
3148                this.symbolSelector.style.left = (canvasRect.left + x + 10) + 'px';
3149                this.symbolSelector.style.top = (canvasRect.top + y + 10) + 'px';
3150
3151                // Style
3152                this.symbolSelector.style.backgroundColor = 'white';
3153                this.symbolSelector.style.border = '1px solid #ccc';
3154                this.symbolSelector.style.padding = '10px';
3155                this.symbolSelector.style.borderRadius = '5px';
3156                this.symbolSelector.style.boxShadow = '0 2px 10px rgba(0,0,0,0.1)';
3157
3158                // Add title
3159                const title = document.createElement('h4');
3160                title.textContent = 'Select Animation Symbol';
3161                title.style.margin = '0 0 10px 0';
3162                this.symbolSelector.appendChild(title);
3163
3164                // Symbol grid
```

```
            const symbolGrid = document.createElement('div');
            symbolGrid.style.display = 'grid';
            symbolGrid.style.gridTemplateColumns = 'repeat(3, 1fr)';
            symbolGrid.style.gap = '10px';
            symbolGrid.style.marginBottom = '10px';

            // Available symbols
            const symbols = [
                { type: 'anticipation', name: 'Anticipation' },
                { type: 'impact', name: 'Impact' },
                { type: 'keyframe', name: 'Keyframe' },
                { type: 'inbetween', name: 'Inbetween' },
                { type: 'hold', name: 'Hold' },
                { type: 'default', name: 'Default' }
            ];

            // Create symbol preview for each type
            symbols.forEach(symbol => {
                const symbolItem = document.createElement('div');
                symbolItem.style.display = 'flex';
                symbolItem.style.flexDirection = 'column';
                symbolItem.style.alignItems = 'center';
                symbolItem.style.cursor = 'pointer';
                symbolItem.style.padding = '5px';
                symbolItem.style.border = '1px solid #ddd';
                symbolItem.style.borderRadius = '3px';

                // Create canvas for symbol preview
                const canvas = document.createElement('canvas');
                canvas.width = 50;
                canvas.height = 50;
                const ctx = canvas.getContext('2d');

                // Draw symbol preview
                ctx.save();
                ctx.strokeStyle = this.settings.color;
                ctx.fillStyle = this.settings.color;
                ctx.lineWidth = 2;

                // Draw centered preview
                const symbolObj = new SymbolObject({
                    x: 25,
                    y: 25,
                    symbolType: symbol.type,
                    color: this.settings.color,
                    lineWidth: 2,
                    scale: 1.5
                });

                symbolObj.draw(ctx);
```

```
                    ctx.restore();

                    const name = document.createElement('div');
                    name.textContent = symbol.name;
                    name.style.marginTop = '5px';
                    name.style.fontSize = '12px';

                    symbolItem.appendChild(canvas);
                    symbolItem.appendChild(name);

                    // Add click handler
                    symbolItem.addEventListener('click', () => {
                        this.addSymbol(x, y, symbol.type);
                        this.removeSymbolSelector();
                    });

                    symbolGrid.appendChild(symbolItem);
                });

                this.symbolSelector.appendChild(symbolGrid);

                // Scale control
                const scaleContainer = document.createElement('div');
                scaleContainer.style.display = 'flex';
                scaleContainer.style.alignItems = 'center';
                scaleContainer.style.marginBottom = '10px';

                const scaleLabel = document.createElement('label');
                scaleLabel.textContent = 'Scale: ';

                const scaleInput = document.createElement('input');
                scaleInput.type = 'range';
                scaleInput.min = '0.5';
                scaleInput.max = '3';
                scaleInput.step = '0.1';
                scaleInput.value = this.settings.symbolScale;
                scaleInput.style.flex = '1';
                scaleInput.style.marginLeft = '5px';

                const scaleValue = document.createElement('span');
                scaleValue.textContent = this.settings.symbolScale + 'x';
                scaleValue.style.marginLeft = '5px';
                scaleValue.style.width = '30px';

                scaleInput.addEventListener('input', () => {
                    this.settings.symbolScale = parseFloat(scaleInput.value);
                    scaleValue.textContent = this.settings.symbolScale + 'x';
                });

                scaleContainer.appendChild(scaleLabel);
```

```
3265                scaleContainer.appendChild(scaleInput);
3266                scaleContainer.appendChild(scaleValue);
3267
3268                this.symbolSelector.appendChild(scaleContainer);
3269
3270                // Add cancel button
3271                const cancelButton = document.createElement('button');
3272                cancelButton.textContent = 'Cancel';
3273                cancelButton.style.width = '100%';
3274                cancelButton.style.padding = '5px';
3275                cancelButton.style.backgroundColor = '#f44336';
3276                cancelButton.style.color = 'white';
3277                cancelButton.style.border = 'none';
3278                cancelButton.style.borderRadius = '3px';
3279                cancelButton.style.cursor = 'pointer';
3280
3281                cancelButton.addEventListener('click', () => {
3282                    this.removeSymbolSelector();
3283                });
3284
3285                this.symbolSelector.appendChild(cancelButton);
3286
3287                // Add to document
3288                document.body.appendChild(this.symbolSelector);
3289            }
3290
3291            removeSymbolSelector() {
3292                if (this.symbolSelector && this.symbolSelector.parentNode) {
3293                    this.symbolSelector.parentNode.removeChild(this.symbolSelector);
3294                    this.symbolSelector = null;
3295                }
3296            }
3297
3298            addSymbol(x, y, symbolType) {
3299                // Create symbol object
3300                const symbolObj = new SymbolObject({
3301                    x: x,
3302                    y: y,
3303                    symbolType: symbolType,
3304                    color: this.settings.color,
3305                    lineWidth: this.settings.lineWidth,
3306                    scale: this.settings.symbolScale
3307                });
3308
3309                // Add to layer
3310                this.layerSystem.addObject(symbolObj);
3311            }
3312        }
3313
3314        // Frame-spanning line tool (for connecting cells across frames)
```

```
3315        class FrameSpanningLineTool extends DrawingTool {
3316            constructor(toolSystem) {
3317                super(toolSystem);
3318                this.startCell = null;
3319                this.currentLine = null;
3320                this.drawing = false;
3321            }
3322
3323            attachEvents() {
3324                // This tool works with the table cells, not the canvas
3325                this.setupCellEvents();
3326            }
3327
3328            detachEvents() {
3329                this.removeCellEvents();
3330
3331                // Finish any in-progress drawing
3332                this.finishDrawing();
3333            }
3334
3335            setupCellEvents() {
3336                // Find all table cells and add mousedown handler
3337                const cells = document.querySelectorAll('#xsheet-table td');
3338                cells.forEach(cell => {
3339                    cell.addEventListener('mousedown', this.handleCellMouseDown);
3340                    cell.addEventListener('mouseup', this.handleCellMouseUp);
3341
3342                    // Add hover effect
3343                    cell.addEventListener('mouseover', this.handleCellHover);
3344                    cell.addEventListener('mouseout', this.handleCellOut);
3345
3346                    // Store original background for hover effect
3347                    if (!cell.dataset.originalBg) {
3348                        cell.dataset.originalBg = cell.style.backgroundColor || '';
3349                    }
3350                });
3351            }
3352
3353            removeCellEvents() {
3354                const cells = document.querySelectorAll('#xsheet-table td');
3355                cells.forEach(cell => {
3356                    cell.removeEventListener('mousedown', this.handleCellMouseDown);
3357                    cell.removeEventListener('mouseup', this.handleCellMouseUp);
3358                    cell.removeEventListener('mouseover', this.handleCellHover);
3359                    cell.removeEventListener('mouseout', this.handleCellOut);
3360
3361                    // Restore original background
3362                    if (cell.dataset.originalBg) {
3363                        cell.style.backgroundColor = cell.dataset.originalBg;
3364                    }
```

```
3365                 });
3366             }
3367
3368             handleCellHover = (e) => {
3369                 const cell = e.currentTarget;
3370                 cell.style.backgroundColor = 'rgba(0, 123, 255, 0.2)';
3371             }
3372
3373             handleCellOut = (e) => {
3374                 const cell = e.currentTarget;
3375                 if (!this.drawing || cell !== this.startCell) {
3376                     cell.style.backgroundColor = cell.dataset.originalBg || '';
3377                 }
3378             }
3379
3380             handleCellMouseDown = (e) => {
3381                 const cell = e.currentTarget;
3382
3383                 // Find the row and column index
3384                 const row = cell.parentElement;
3385                 const frameNumber = parseInt(row.getAttribute('data-frame') ||
    row.className.replace('frame-', ''));
3386                 const columnIndex = Array.from(row.children).indexOf(cell) + 1;
3387
3388                 if (isNaN(frameNumber) || frameNumber <= 0) return;
3389
3390                 // Store as start cell
3391                 this.startCell = cell;
3392                 this.drawing = true;
3393
3394                 // Highlight the cell
3395                 cell.style.backgroundColor = 'rgba(0, 123, 255, 0.4)';
3396
3397                 // Create temporary frame-spanning line object
3398                 this.currentLine = new FrameSpanningLineObject({
3399                     startFrame: frameNumber,
3400                     startColumn: columnIndex,
3401                     endFrame: frameNumber,
3402                     endColumn: columnIndex,
3403                     color: this.settings.color,
3404                     lineWidth: this.settings.lineWidth,
3405                     arrowEnd: true
3406                 });
3407
3408                 // Add to layer
3409                 this.layerSystem.addObject(this.currentLine);
3410             }
3411
3412             handleCellMouseUp = (e) => {
3413                 if (!this.drawing || !this.currentLine) return;
```

```
3414
3415                const cell = e.currentTarget;
3416
3417                // Skip if this is the same cell
3418                if (cell === this.startCell) {
3419                    this.finishDrawing();
3420                    return;
3421                }
3422
3423                // Find the row and column index
3424                const row = cell.parentElement;
3425                const frameNumber = parseInt(row.getAttribute('data-frame') ||
    row.className.replace('frame-', ''));
3426                const columnIndex = Array.from(row.children).indexOf(cell) + 1;
3427
3428                if (isNaN(frameNumber) || frameNumber <= 0) {
3429                    this.finishDrawing();
3430                    return;
3431                }
3432
3433                // Update the end point of the line
3434                this.currentLine.endFrame = frameNumber;
3435                this.currentLine.endColumn = columnIndex;
3436
3437                // Redraw
3438                this.layerSystem.redrawLayer(this.layerSystem.activeLayerIndex);
3439
3440                // Finish drawing
3441                this.finishDrawing();
3442            }
3443
3444            finishDrawing() {
3445                if (this.drawing) {
3446                    // Reset flags
3447                    this.drawing = false;
3448
3449                    // If the line is too short (same start and end), remove it
3450                    if (this.currentLine &&
3451                        this.currentLine.startFrame === this.currentLine.endFrame &&
3452                        this.currentLine.startColumn === this.currentLine.endColumn) {
3453                        this.layerSystem.removeObject(this.currentLine);
3454                    }
3455
3456                    // Clear current line reference
3457                    this.currentLine = null;
3458
3459                    // Reset cell highlights
3460                    if (this.startCell) {
3461                        this.startCell.style.backgroundColor =
    this.startCell.dataset.originalBg || '';
```

```
                        this.startCell = null;
                    }
                }
            }
        }

        // Eraser tool
        class EraserTool extends DrawingTool {
            constructor(toolSystem) {
                super(toolSystem);
            }

            attachEvents() {
                const canvas = this.layerSystem.getActiveLayer().canvas;

                canvas.addEventListener('mousedown', this.handleMouseDown);
            }

            detachEvents() {
                const canvas = this.layerSystem.getActiveLayer().canvas;

                canvas.removeEventListener('mousedown', this.handleMouseDown);
            }

            handleMouseDown = (e) => {
                const coords = this.layerSystem.screenToCanvas(e.clientX, e.clientY);

                // Find object at click position
                const hit = this.layerSystem.findObjectAt(coords.x, coords.y);

                if (hit) {
                    // Remove the object
                    this.layerSystem.removeObject(hit.object, hit.layerIndex);
                }
            }
        }

        /**
         * INTEGRATION WITH X-SHEET APPLICATION
         * Initialize the drawing system, save/load integration, and PDF/print handling
         */

        // Initialize drawing system when document is loaded
        function initDrawingSystem() {
            // Setup after the table is rendered
            setTimeout(() => {
                // Get the X-Sheet table element
                const xsheetTable = document.getElementById('xsheet-table');
                if (!xsheetTable) {
                    console.error('X-Sheet table not found');
```

```
                    return;
                }

                // Create drawing systems
                const drawingLayerSystem = new DrawingLayerSystem(xsheetTable);
                const drawingToolSystem = new DrawingToolSystem(drawingLayerSystem);

                // Store global reference
                window.xsheetDrawing = {
                    layerSystem: drawingLayerSystem,
                    toolSystem: drawingToolSystem
                };

                // Add to status message
                const statusElement = document.getElementById('status-message');
                if (statusElement) {
                    statusElement.textContent = 'Drawing tools initialized';
                }

                // Integrate with X-Sheet save/load system
                integrateWithXSheetSaveLoad();

                // Integrate with X-Sheet PDF and printing
                integrateWithXSheetExport();

                // Custom event for when drawing objects change
                document.addEventListener('xsheet-redraw', function() {
                    drawingLayerSystem.redrawAll();
                });

                console.log('Drawing tools initialized successfully');
            }, 500);
        }

        // Integrate drawing data with X-Sheet save/load system
        function integrateWithXSheetSaveLoad() {
            // Store original functions
            const originalCollectData = window.collectData;
            const originalRestoreData = window.restoreData;

            // Override collectData to include drawings
            window.collectData = function() {
                // Call original function to get base data
                const data = originalCollectData ? originalCollectData() : {};

                // Add drawing data if drawing system is initialized
                if (window.xsheetDrawing && window.xsheetDrawing.layerSystem) {
                    data.drawingLayers = [];

                    // Collect objects from each layer
```

```
3562                    window.xsheetDrawing.layerSystem.layers.forEach(layer => {
3563                        const layerData = {
3564                            name: layer.name,
3565                            visible: layer.visible,
3566                            objects: layer.objects.map(obj => obj.toJSON())
3567                        };
3568
3569                        data.drawingLayers.push(layerData);
3570                    });
3571                }
3572
3573                return data;
3574            };
3575
3576            // Override restoreData to handle drawings
3577            window.restoreData = function(data) {
3578                // Call original function to restore base data
3579                if (originalRestoreData) {
3580                    originalRestoreData(data);
3581                }
3582
3583                // Restore drawing data if available
3584                if (data.drawingLayers && window.xsheetDrawing &&
    window.xsheetDrawing.layerSystem) {
3585                    const layerSystem = window.xsheetDrawing.layerSystem;
3586
3587                    // Clear existing layers
3588                    layerSystem.clearAllLayers();
3589
3590                    // Restore each layer
3591                    data.drawingLayers.forEach((layerData, index) => {
3592                        // Create layer if needed
3593                        if (index >= layerSystem.layers.length) {
3594                            layerSystem.addLayer(layerData.name);
3595                        } else {
3596                            layerSystem.layers[index].name = layerData.name;
3597                            layerSystem.layers[index].visible = layerData.visible;
3598                        }
3599
3600                        // Restore objects
3601                        layerData.objects.forEach(objData => {
3602                            const newObj = DrawingObjectFactory.createFromJSON(objData);
3603                            if (newObj) {
3604                                layerSystem.layers[index].objects.push(newObj);
3605                            }
3606                        });
3607                    });
3608
3609                    // Redraw
3610                    layerSystem.redrawAll();
```

```
3611                 }
3612             };
3613         }
3614
3615        // Integrate drawing layers with PDF export and printing
3616         function integrateWithXSheetExport() {
3617             // Store original functions
3618             const originalExportToPDF = window.exportToPDF;
3619             const originalPrintSheet = window.printSheet;
3620
3621             // Override PDF export to include drawings
3622             window.exportToPDF = function () {
3623                 if (!window.xsheetDrawing || !window.xsheetDrawing.layerSystem) {
3624                     return originalExportToPDF ? originalExportToPDF() : null;
3625                 }
3626
3627                 // Save current state
3628                 const layerSystem = window.xsheetDrawing.layerSystem;
3629                 const layerContainer = layerSystem.container;
3630
3631                 // 1) Save original CSS & DOM properties
3632                 const originalDisplay = layerContainer.style.display;
3633                 const originalPos = layerContainer.style.position;
3634                 const originalTop = layerContainer.style.top;
3635                 const originalLeft = layerContainer.style.left;
3636                 const originalZIndex = layerContainer.style.zIndex;
3637                 const originalWidth = layerContainer.style.width;
3638                 const originalHeight = layerContainer.style.height;
3639                 const originalParent = layerContainer.parentNode;
3640
3641                 // 2) Make it visible for capture
3642                 layerContainer.style.display = 'block';
3643
3644                 // 3) Compute offsets - FIXED POSITIONING LOGIC
3645                 const printableArea = document.getElementById('printable-area');
3646                 const printableRect = printableArea.getBoundingClientRect();
3647                 const tableRect = document.getElementById('xsheet-
table').getBoundingClientRect();
3648
3649                 // 4) Reparent into the printable area
3650                 printableArea.appendChild(layerContainer);
3651
3652                 // 5) Force absolute positioning & explicit size - FIXED POSITIONING LOGIC
3653                 layerContainer.style.position = 'absolute';
3654                 layerContainer.style.top = `${tableRect.top - printableRect.top}px`;
3655                 layerContainer.style.left = `${tableRect.left - printableRect.left}px`;
3656                 layerContainer.style.width = `${tableRect.width}px`;
3657                 layerContainer.style.height = `${tableRect.height}px`;
3658                 layerContainer.style.zIndex = '1000';
3659
```

```
                // Force a redraw of all drawing objects
                layerSystem.redrawAll();

                // 6) Wait a brief moment to ensure drawings are rendered
                setTimeout(() => {
                    // 7) Generate the PDF
                    const result = originalExportToPDF ? originalExportToPDF() : null;

                    // 8) Restore everything
                    layerContainer.style.display = originalDisplay;
                    layerContainer.style.position = originalPos;
                    layerContainer.style.top = originalTop;
                    layerContainer.style.left = originalLeft;
                    layerContainer.style.zIndex = originalZIndex;
                    layerContainer.style.width = originalWidth;
                    layerContainer.style.height = originalHeight;
                    originalParent.appendChild(layerContainer);
                }, 50);

                return true; // Will be async due to setTimeout
            };

            // Override print to include drawings
            window.printSheet = function () {
                // 1) Fallback if no drawing system
                if (!window.xsheetDrawing || !window.xsheetDrawing.layerSystem) {
                    return originalPrintSheet ? originalPrintSheet() : null;
                }

                // 2) Grab the container and save its original state
                const layerSystem = window.xsheetDrawing.layerSystem;
                const layerContainer = layerSystem.container;
                const originalDisplay = layerContainer.style.display;
                const originalPos = layerContainer.style.position;
                const originalTop = layerContainer.style.top;
                const originalLeft = layerContainer.style.left;
                const originalZIndex = layerContainer.style.zIndex;
                const originalWidth = layerContainer.style.width;
                const originalHeight = layerContainer.style.height;
                const originalParent = layerContainer.parentNode;

                // 3) Make it visible for print capture
                layerContainer.style.display = 'block';

                // 4) Compute offsets relative to the sheet - FIXED POSITIONING LOGIC
                const printableArea = document.getElementById('printable-area');
                const printableRect = printableArea.getBoundingClientRect();
                const tableRect = document.getElementById('xsheet-
table').getBoundingClientRect();

```

```
3709                // 5) Reparent into the printable-area element
3710                printableArea.appendChild(layerContainer);
3711
3712                // 6) Force absolute positioning & lock size - FIXED POSITIONING LOGIC
3713                layerContainer.style.position = 'absolute';
3714                layerContainer.style.top = `${tableRect.top - printableRect.top}px`;
3715                layerContainer.style.left = `${tableRect.left - printableRect.left}px`;
3716                layerContainer.style.width = `${tableRect.width}px`;
3717                layerContainer.style.height = `${tableRect.height}px`;
3718                layerContainer.style.zIndex = '1000';
3719
3720                // 7) Force a redraw of all drawing objects
3721                layerSystem.redrawAll();
3722
3723                // 8) Add a print class to the container for print-specific CSS
3724                layerContainer.classList.add('printing');
3725
3726                // 9) Call the original print with a slight delay to ensure drawing render
3727                setTimeout(() => {
3728                    const result = originalPrintSheet ? originalPrintSheet() :
    window.print();
3729
3730                    // 10) Restore everything back
3731                    layerContainer.style.display = originalDisplay;
3732                    layerContainer.style.position = originalPos;
3733                    layerContainer.style.top = originalTop;
3734                    layerContainer.style.left = originalLeft;
3735                    layerContainer.style.zIndex = originalZIndex;
3736                    layerContainer.style.width = originalWidth;
3737                    layerContainer.style.height = originalHeight;
3738                    layerContainer.classList.remove('printing');
3739                    originalParent.appendChild(layerContainer);
3740                }, 50);
3741
3742                return true; // Will be async due to setTimeout
3743            };
3744        }
3745
3746
3747        // Setup event listeners for table updates
3748        function setupXSheetUpdateHandling() {
3749            // Hook into existing update functions
3750            if (typeof window.generateTable === 'function') {
3751                const originalGenerateTable = window.generateTable;
3752                window.generateTable = function() {
3753                    // Call original function
3754                    const result = originalGenerateTable.apply(this, arguments);
3755
3756                    // Fire update event
3757                    document.dispatchEvent(new Event('xsheet-updated'));
```

```
                    return result;
                };
            }

            if (typeof window.addEightRows === 'function') {
                const originalAddEightRows = window.addEightRows;
                window.addEightRows = function() {
                    // Call original function
                    const result = originalAddEightRows.apply(this, arguments);

                    // Fire update event
                    document.dispatchEvent(new Event('xsheet-updated'));

                    return result;
                };
            }

            if (typeof window.updateTemplate === 'function') {
                const originalUpdateTemplate = window.updateTemplate;
                window.updateTemplate = function() {
                    // Call original function
                    const result = originalUpdateTemplate.apply(this, arguments);

                    // Fire update event
                    document.dispatchEvent(new Event('xsheet-updated'));

                    return result;
                };
            }
        }

        // Original X-Sheet code
        // Wait for page load and libraries to initialize
        document.addEventListener('DOMContentLoaded', function () {
            // Global variables
            let frameCount = 96;
            let projectName = 'Animation_XSheet_' + new Date().toISOString().split('T')[0];
            let modified = false;
            let currentTemplate = 'large';

            // Audio variables
            let audioContext = null;
            let audioBuffer = null;
            let audioSource = null;
            let audioData = null;
            let isPlaying = false;
            let startTime = 0;
            let startOffset = 0;
            let audioFileName = '';
```

```
3808            let waveformData = [];
3809            let waveformCanvases = [];
3810            let phonetics = [];
3811            let frameDuration = 1 / 24; // 24fps
3812            let currentFrame = 0;
3813            let phoneticEditPosition = null;
3814
3815            // Initialize UI elements
3816            const templateSelector = document.getElementById('template-selector');
3817            const frameCountInput = document.getElementById('frame-count');
3818            const saveButton = document.getElementById('save-button');
3819            const loadButton = document.getElementById('load-button');
3820            const pdfButton = document.getElementById('pdf-button');
3821            const printButton = document.getElementById('print-button');
3822            const addRowsButton = document.getElementById('add-rows-button');
3823            const clearButton = document.getElementById('clear-button');
3824            const tableBody = document.getElementById('xsheet-body');
3825            const statusMessage = document.getElementById('status-message');
3826            const audioButton = document.getElementById('audio-button');
3827            const audioUpload = document.getElementById('audio-upload');
3828            const playAudioButton = document.getElementById('play-audio');
3829            const stopAudioButton = document.getElementById('stop-audio');
3830            const audioScrubber = document.getElementById('audio-scrubber');
3831            const audioInfo = document.getElementById('audio-info');
3832            const addPhoneticButton = document.getElementById('add-phonetic');
3833            const phoneticInput = document.getElementById('phonetic-input');
3834            const phoneticText = document.getElementById('phonetic-text');
3835            const savePhoneticButton = document.getElementById('save-phonetic');
3836            const cancelPhoneticButton = document.getElementById('cancel-phonetic');
3837
3838            // Date field - set today by default
3839            const dateField = document.getElementById('project-date');
3840            dateField.valueAsDate = new Date();
3841
3842            // Initialize the table
3843            generateTable(frameCount);
3844
3845            // Initialize the drawing system
3846            initDrawingSystem();
3847            setupXSheetUpdateHandling();
3848
3849            // Event Listeners
3850            templateSelector.addEventListener('change', function () {
3851                currentTemplate = this.value;
3852                updateTemplate();
3853            });
3854
3855            frameCountInput.addEventListener('change', function () {
3856                frameCount = parseInt(this.value);
3857                if (frameCount < 8) frameCount = 8;
```

```
3858                generateTable(frameCount);
3859                updateStatusMessage('Frame count updated to ' + frameCount);
3860
3861                // Re-render waveform if audio is loaded
3862                if (audioBuffer) {
3863                    renderWaveform();
3864                }
3865            });
3866
3867            saveButton.addEventListener('click', saveProject);
3868            loadButton.addEventListener('click', loadProject);
3869            pdfButton.addEventListener('click', exportToPDF);
3870            printButton.addEventListener('click', printSheet);
3871            addRowsButton.addEventListener('click', addEightRows);
3872            clearButton.addEventListener('click', clearSheet);
3873
3874            // Audio related event listeners
3875            audioButton.addEventListener('click', function () {
3876                audioUpload.click();
3877            });
3878
3879            audioUpload.addEventListener('change', function (e) {
3880                if (e.target.files.length > 0) {
3881                    const file = e.target.files[0];
3882                    audioFileName = file.name;
3883                    loadAudioFile(file);
3884                }
3885            });
3886
3887            playAudioButton.addEventListener('click', togglePlayAudio);
3888            stopAudioButton.addEventListener('click', stopAudio);
3889            audioScrubber.addEventListener('input', scrubAudio);
3890
3891            addPhoneticButton.addEventListener('click', function () {
3892                if (audioBuffer) {
3893                    showPhoneticInput(null);
3894                } else {
3895                    updateStatusMessage('Please load audio first');
3896                }
3897            });
3898
3899            savePhoneticButton.addEventListener('click', savePhoneticMarker);
3900            cancelPhoneticButton.addEventListener('click', function () {
3901                phoneticInput.style.display = 'none';
3902            });
3903
3904            // Monitor for changes to set modified flag
3905            document.addEventListener('input', function (e) {
3906                if (!e.target.matches('input, [contenteditable="true"]')) return;
3907                modified = true;
```

```
3908
3909                if (e.target.matches('[contenteditable="true"]')) {
3910                    // Add the modified class when content is added
3911                    e.target.classList.add('modified');
3912
3913                    // Remove the modified class when the cell is empty
3914                    if (e.target.textContent.trim() === '') {
3915                        e.target.classList.remove('modified');
3916                    }
3917                }
3918
3919                updateStatusMessage('Changes detected - not saved');
3920            });
3921
3922            // Variables for cell selection
3923            let selectedCells = [];
3924            let isSelecting = false;
3925            let selectionStart = null;
3926            let hasMovedDuringSelection = false;
3927
3928            // Set up multi-cell selection functionality
3929            function setupCellSelection() {
3930                const editableCells = document.querySelectorAll('[contenteditable="true"]');
3931
3932                // Clear selected cells when clicking outside
3933                document.addEventListener('click', function (e) {
3934                    // If this was part of a drag operation, skip the click handler
3935                    if (hasMovedDuringSelection) {
3936                        // We removed this line to fix the selection issue
3937                        // hasMovedDuringSelection = false;
3938                        return;
3939                    }
3940
3941                    // Get the clicked element and check if it's inside an editable cell or
    a selected cell
3942                    const clickedCell = e.target.closest('[contenteditable="true"]');
3943                    const clickedInSelection = e.target.closest('.selected-cell');
3944
3945                    // If not clicking in a cell or selection and not using modifier keys
3946                    if (!clickedCell && !clickedInSelection && !e.ctrlKey && !e.metaKey) {
3947                        clearCellSelection();
3948                    }
3949                });
3950
3951                // Handle clicks on the table for better cell focusing
3952                document.getElementById('xsheet-table').addEventListener('click', function
    (e) {
3953                    // Find the closest editable cell from the click point
3954                    const targetCell = e.target.closest('[contenteditable="true"]');
3955
```

```
3956                    // If we found a cell and we're not in multi-select mode
3957                    if (targetCell && !hasMovedDuringSelection && !e.ctrlKey && !e.metaKey)
    {
3958                        // Focus the cell and position cursor at the end
3959                        if (selectedCells.length === 1 && selectedCells[0] === targetCell) {
3960                            // If already selected, just ensure focus
3961                            targetCell.focus();
3962                            placeCaretAtEnd(targetCell);
3963                        }
3964                    }
3965                });
3966
3967                // Helper function to place caret at the end of content
3968                function placeCaretAtEnd(el) {
3969                    if (document.createRange) {
3970                        const range = document.createRange();
3971                        range.selectNodeContents(el);
3972                        range.collapse(false); // false means collapse to end
3973                        const selection = window.getSelection();
3974                        selection.removeAllRanges();
3975                        selection.addRange(range);
3976                    }
3977                }
3978
3979                // Handle mousedown for selection start
3980                editableCells.forEach(cell => {
3981                    // Mousedown - start potential selection
3982                    cell.addEventListener('mousedown', function (e) {
3983                        // Only handle left mouse button
3984                        if (e.button !== 0) return;
3985
3986                        // If using modifier keys for multi-select
3987                        if (e.ctrlKey || e.metaKey) {
3988                            toggleCellSelection(cell);
3989                            e.preventDefault(); // Prevent text cursor
3990                            return;
3991                        }
3992
3993                        // Clear previous selection unless clicking on already selected cell
3994                        if (!cell.classList.contains('selected-cell')) {
3995                            clearCellSelection();
3996                        }
3997
3998                        // Store starting cell and reset movement flag
3999                        selectionStart = cell;
4000                        hasMovedDuringSelection = false;
4001                        isSelecting = true;
4002
4003                        // Initially add this cell to selection
4004                        if (!cell.classList.contains('selected-cell')) {
```

```
                        toggleCellSelection(cell, true);
                    }

                    // If it's a single click (not start of drag), focus the cell
                    cell.focus();

                    // Don't prevent default here to allow normal focus behavior
                });

                // Additional keyboard events for selected cells
                cell.addEventListener('keydown', function (e) {
                    // If Delete or Backspace key and we have selected cells
                    if ((e.key === 'Delete' || e.key === 'Backspace') &&
selectedCells.length > 1) {
                        e.preventDefault();
                        clearSelectedCellsContent();
                    }

                    // Copy with Ctrl+C or Cmd+C
                    if ((e.ctrlKey || e.metaKey) && e.key === 'c' &&
selectedCells.length > 0) {
                        e.preventDefault();
                        copySelectedCells();
                    }

                    // Select all cells in row with Ctrl+A or Cmd+A
                    if ((e.ctrlKey || e.metaKey) && e.key === 'a') {
                        e.preventDefault();
                        selectRowCells(cell.parentElement);
                    }
                });
            });

            // Global mousemove handler for extending selection
            document.addEventListener('mousemove', function (e) {
                if (!isSelecting) return;

                // Set the flag that we've moved during this selection
                hasMovedDuringSelection = true;

                // Find the element under the cursor
                const elementUnderCursor = document.elementFromPoint(e.clientX,
e.clientY);
                if (!elementUnderCursor) return;

                // Find the closest editable cell
                const targetCell =
elementUnderCursor.closest('[contenteditable="true"]');
                if (targetCell) {
                    // Add to selection
                    toggleCellSelection(targetCell, true);
```

```
4052
4053                        // Prevent text selection during drag
4054                        e.preventDefault();
4055                        if (window.getSelection) {
4056                            window.getSelection().removeAllRanges();
4057                        }
4058                    }
4059                });
4060
4061                // Global mouseup to end selection - THIS IS THE FIXED VERSION
4062                document.addEventListener('mouseup', function (e) {
4063                    // End selection mode but keep selected cells
4064                    if (isSelecting) {
4065                        isSelecting = false;
4066
4067                        // FIX: Use setTimeout to delay resetting the hasMovedDuringSelect-
    ion flag
4068                        // This gives the click handler a chance to see the flag first
4069                        if (hasMovedDuringSelection) {
4070                            setTimeout(function () {
4071                                hasMovedDuringSelection = false;
4072                            }, 10);
4073                        }
4074
4075                        // If this wasn't a drag and there's only one cell selected, focus
    it
4076                        if (!hasMovedDuringSelection && selectedCells.length === 1) {
4077                            const cell = selectedCells[0];
4078                            cell.focus();
4079                            placeCaretAtEnd(cell);
4080                        }
4081                    }
4082
4083                    selectionStart = null;
4084                });
4085
4086                // Listen for paste events
4087                document.addEventListener('paste', function (e) {
4088                    if (selectedCells.length > 0) {
4089                        handlePaste(e);
4090                    }
4091                });
4092            }
4093
4094            function toggleCellSelection(cell, addOnly = false) {
4095                const index = selectedCells.indexOf(cell);
4096
4097                if (index === -1) {
4098                    // Add to selection
4099                    selectedCells.push(cell);
```

```
                    cell.classList.add('selected-cell');
                } else if (!addOnly) {
                    // Remove from selection if not add-only mode
                    selectedCells.splice(index, 1);
                    cell.classList.remove('selected-cell');
                }
            }

            function clearCellSelection() {
                selectedCells.forEach(cell => {
                    cell.classList.remove('selected-cell');
                });
                selectedCells = [];
            }

            function clearSelectedCellsContent() {
                selectedCells.forEach(cell => {
                    cell.textContent = '';
                    cell.classList.remove('modified');
                });
                modified = true;
                updateStatusMessage('Cleared selected cells');
            }

            function copySelectedCells() {
                if (selectedCells.length === 0) return;

                // Create a text representation of selected cells
                const cellData = selectedCells.map(cell => cell.textContent ||
''').join('\t');

                // Copy to clipboard using Clipboard API
                navigator.clipboard.writeText(cellData)
                    .then(() => {
                        updateStatusMessage('Copied selected cells to clipboard');
                    })
                    .catch(err => {
                        updateStatusMessage('Failed to copy: ' + err);
                    });
            }

            function handlePaste(e) {
                // Prevent default paste behavior
                e.preventDefault();

                // Get clipboard data
                const clipboardData = e.clipboardData || window.clipboardData;
                const pastedText = clipboardData.getData('text');

                // If we have a focused element within selection, paste there
```

```
4149                const activeElement = document.activeElement;
4150                if (activeElement && selectedCells.includes(activeElement)) {
4151                    activeElement.textContent = pastedText;
4152                    activeElement.classList.add('modified');
4153                    modified = true;
4154                } else if (selectedCells.length > 0) {
4155                    // Otherwise paste into first selected cell
4156                    selectedCells[0].textContent = pastedText;
4157                    selectedCells[0].classList.add('modified');
4158                    modified = true;
4159                }
4160
4161                if (pastedText.trim() === '') {
4162                    selectedCells.forEach(cell => {
4163                        cell.classList.remove('modified');
4164                    });
4165                }
4166
4167                updateStatusMessage('Pasted content into selected cell');
4168            }
4169
4170            function selectRowCells(row) {
4171                // Clear previous selection
4172                clearCellSelection();
4173
4174                // Select all editable cells in the row
4175                const cells = Array.from(row.cells).filter(cell => cell.contentEditable ===
    'true');
4176                cells.forEach(cell => {
4177                    toggleCellSelection(cell, true);
4178                });
4179            }
4180
4181            // Functions
4182            function generateTable(frames) {
4183                tableBody.innerHTML = '';
4184                waveformCanvases = [];
4185
4186                // Create a column container for the waveform
4187                const waveformColContainer = document.createElement('div');
4188                waveformColContainer.className = 'waveform-col-container';
4189
4190                for (let i = 1; i <= frames; i++) {
4191                    const row = document.createElement('tr');
4192                    row.className = `frame-${i}`;
4193                    row.setAttribute('data-frame', i);
4194
4195                    // Create standard cell
4196                    function createCell(className, isEditable = true, frameNum = null) {
4197                        const cell = document.createElement('td');
```

```
4198                        cell.className = className;
4199
4200                        if (isEditable) {
4201                            cell.contentEditable = true;
4202                            cell.setAttribute('data-placeholder', '');
4203                            cell.setAttribute('tabindex', '0');
4204                        } else if (frameNum !== null) {
4205                            cell.textContent = frameNum;
4206                            cell.className += ' frame-number';
4207
4208                            // Color the first frame green for reference (like in the
    example image)
4209                            if (frameNum === 1) {
4210                                cell.style.backgroundColor = '#00cc00';
4211                            } else {
4212                                cell.style.backgroundColor = '#cccccc';
4213                            }
4214                        }
4215
4216                        return cell;
4217                    }
4218
4219                    // Create waveform cell (placeholder for the vertical waveform)
4220                    function createWaveformCell(frameNum) {
4221                        const cell = document.createElement('td');
4222                        cell.className = 'waveform-col';
4223                        cell.setAttribute('data-frame', frameNum);
4224                        return cell;
4225                    }
4226
4227                    // Add all cells to the row
4228                    row.appendChild(createCell('action-col'));
4229                    row.appendChild(createCell('frame-col', false, i));
4230                    row.appendChild(createWaveformCell(i));
4231                    row.appendChild(createCell('dialogue-col'));
4232                    row.appendChild(createCell('sound-col'));
4233                    row.appendChild(createCell('technical-col'));
4234                    row.appendChild(createCell('extra1-col'));
4235                    row.appendChild(createCell('extra2-col'));
4236                    row.appendChild(createCell('frame-col', false, i));
4237                    row.appendChild(createCell('camera-col'));
4238
4239                    // Add special styling for 8-frame and 24-frame intervals
4240                    if (i % 24 === 0) {
4241                        row.style.borderBottom = '4px double #000';
4242                        for (let cell of row.cells) {
4243                            cell.style.borderBottom = '4px double #000';
4244                            cell.style.fontWeight = 'bold';
4245                        }
4246                    } else if (i % 8 === 0) {
```

```
                        row.style.borderBottom = '2px solid #000';
                        for (let cell of row.cells) {
                            cell.style.borderBottom = '2px solid #000';
                            cell.style.fontWeight = 'bold';
                        }
                    }

                    tableBody.appendChild(row);
                }

                setupCellNavigation();
                updateStatusMessage('Table generated with ' + frames + ' frames');

                // Fire custom event for drawing system to update
                document.dispatchEvent(new Event('xsheet-updated'));
            }

            function updateTemplate() {
                if (currentTemplate === 'large') {
                    // 11"x17" template (96 frames)
                    document.body.style.maxWidth = '11in';
                    document.body.style.maxHeight = '17in';
                    document.body.style.fontSize = '9pt';
                    frameCountInput.value = 96;
                    frameCount = 96;
                } else {
                    // 8"x10" template (48 frames)
                    document.body.style.maxWidth = '8in';
                    document.body.style.maxHeight = '10in';
                    document.body.style.fontSize = '8pt';
                    frameCountInput.value = 48;
                    frameCount = 48;
                }

                generateTable(frameCount);
                updateStatusMessage('Template switched to ' + (currentTemplate === 'large' ? '11"x17"' : '8"x10"'));

                // Re-render waveform if audio is loaded
                if (audioBuffer) {
                    renderWaveform();
                }
            }

            function setupCellNavigation() {
                const editableCells = document.querySelectorAll('[contenteditable="true"]');

                editableCells.forEach(cell => {
                    cell.addEventListener('keydown', function (e) {
                        // Tab navigation
```

```
4296                        if (e.key === 'Tab') {
4297                            e.preventDefault();
4298                            const currentRow = this.parentElement;
4299                            const currentIndex = Array.from(currentRow.cells).indexOf(this);
4300
4301                            if (e.shiftKey) {
4302                                // Shift+Tab moves backward
4303                                let prevCell = null;
4304                                if (currentIndex > 0) {
4305                                    // Find previous editable cell in same row
4306                                    for (let i = currentIndex - 1; i >= 0; i--) {
4307                                        if (currentRow.cells[i].contentEditable === 'true')
    {
4308                                            prevCell = currentRow.cells[i];
4309                                            break;
4310                                        }
4311                                    }
4312                                }
4313
4314                                if (!prevCell) {
4315                                    // Move to previous row, last cell
4316                                    const prevRow = currentRow.previousElementSibling;
4317                                    if (prevRow) {
4318                                        const cells = Array.from(prevRow.cells).filter(c =>
    c.contentEditable === 'true');
4319                                        prevCell = cells[cells.length - 1];
4320                                    }
4321                                }
4322
4323                                if (prevCell) {
4324                                    prevCell.focus();
4325                                    // Place cursor at end of content
4326                                    const range = document.createRange();
4327                                    const sel = window.getSelection();
4328                                    range.selectNodeContents(prevCell);
4329                                    range.collapse(false);
4330                                    sel.removeAllRanges();
4331                                    sel.addRange(range);
4332                                }
4333                            } else {
4334                                // Tab moves forward
4335                                let nextCell = null;
4336                                if (currentIndex < currentRow.cells.length - 1) {
4337                                    // Find next editable cell in same row
4338                                    for (let i = currentIndex + 1; i <
    currentRow.cells.length; i++) {
4339                                        if (currentRow.cells[i].contentEditable === 'true')
    {
4340                                            nextCell = currentRow.cells[i];
4341                                            break;
4342                                        }
```

```
4343                                    }
4344                                }
4345
4346                                if (!nextCell) {
4347                                    // Move to next row, first cell
4348                                    const nextRow = currentRow.nextElementSibling;
4349                                    if (nextRow) {
4350                                        nextCell = Array.from(nextRow.cells).find(c =>
    c.contentEditable === 'true');
4351                                    }
4352                                }
4353
4354                                if (nextCell) {
4355                                    nextCell.focus();
4356                                }
4357                            }
4358                        }
4359
4360                        // Enter key to move down
4361                        if (e.key === 'Enter' && !e.shiftKey) {
4362                            e.preventDefault();
4363                            const currentRow = this.parentElement;
4364                            const nextRow = currentRow.nextElementSibling;
4365                            const currentIndex = Array.from(currentRow.cells).indexOf(this);
4366
4367                            if (nextRow) {
4368                                const cells = Array.from(nextRow.cells);
4369                                const samePositionCell = cells[currentIndex];
4370                                if (samePositionCell && samePositionCell.contentEditable ===
    'true') {
4371                                    samePositionCell.focus();
4372                                }
4373                            }
4374                        }
4375                    });
4376
4377                    // Add listener to remove the modified class when cell is emptied
4378                    cell.addEventListener('keyup', function (e) {
4379                        if ((e.key === 'Delete' || e.key === 'Backspace') &&
    this.textContent.trim() === '') {
4380                            this.classList.remove('modified');
4381                        }
4382                    });
4383                });
4384
4385                // Setup cell selection functionality
4386                setupCellSelection();
4387            }
4388
4389            // Audio functions
4390            function loadAudioFile(file) {
```

```
4391                if (!audioContext) {
4392                    try {
4393                        window.AudioContext = window.AudioContext ||
    window.webkitAudioContext;
4394                        audioContext = new AudioContext();
4395                    } catch (e) {
4396                        updateStatusMessage('Web Audio API is not supported in this
    browser');
4397                        return;
4398                    }
4399                }
4400
4401                const reader = new FileReader();
4402
4403                reader.onload = function (e) {
4404                    const audioData = e.target.result;
4405
4406                    // Update status
4407                    updateStatusMessage('Decoding audio...');
4408
4409                    // Decode the audio data
4410                    audioContext.decodeAudioData(audioData, function (buffer) {
4411                        audioBuffer = buffer;
4412
4413                        // Update audio info
4414                        const duration = buffer.duration;
4415                        const minutes = Math.floor(duration / 60);
4416                        const seconds = Math.floor(duration % 60);
4417                        const frameCount = Math.ceil(duration * 24); // Assuming 24fps
4418
4419                        audioInfo.textContent = `${audioFileName}
    (${minutes}:${seconds.toString().padStart(2, '0')}, ${frameCount} frames @ 24fps)`;
4420
4421                        // Enable audio controls
4422                        playAudioButton.disabled = false;
4423                        stopAudioButton.disabled = false;
4424                        audioScrubber.disabled = false;
4425
4426                        // Generate waveform visualization
4427                        generateWaveformData(buffer);
4428
4429                        // Update status
4430                        updateStatusMessage('Audio loaded: ' + audioFileName);
4431
4432                        // If the x-sheet has fewer frames than the audio, suggest
    increasing
4433                        if (frameCount > frameCountInput.value) {
4434                            if (confirm(`This audio is ${frameCount} frames long at 24fps,
    but your X-sheet only has ${frameCountInput.value} frames. Do you want to increase the frame
    count?`)) {
4435                                frameCountInput.value = frameCount;
```

```
                                frameCount = frameCount;
                                generateTable(frameCount);
                                renderWaveform();
                            } else {
                                renderWaveform();
                            }
                        } else {
                            renderWaveform();
                        }
                    }, function (e) {
                        updateStatusMessage('Error decoding audio: ' + e.message);
                    });
                };

                reader.onerror = function () {
                    updateStatusMessage('Error reading audio file');
                };

                reader.readAsArrayBuffer(file);
            }

            function generateWaveformData(buffer) {
                // Get the raw audio data from the buffer
                const rawData = buffer.getChannelData(0); // Use first channel

                // Calculate how many samples we need for our visualization
                const totalSamples = rawData.length;

                // Process for visualization - we need to reduce the resolution
                // to make it efficient to display
                waveformData = [];

                // Calculate desired number of points for the visualization
                // For vertical waveform, we want more detail
                const pointsPerSecond = 100; // Increase detail for vertical display
                const totalPoints = Math.ceil(buffer.duration * pointsPerSecond);

                // Calculate step size
                const step = Math.floor(totalSamples / totalPoints);

                // Build the waveform data array
                for (let i = 0; i < totalPoints; i++) {
                    const index = Math.floor(i * step);
                    if (index < totalSamples) {
                        // Get the absolute value for a nicer visual
                        waveformData.push(Math.abs(rawData[index]));
                    }
                }
            }

```

```
4486            function renderWaveform() {
4487                if (!audioBuffer || waveformData.length === 0) return;
4488
4489                // Clear existing markers and components
4490                const existingMarkers = document.querySelectorAll('.waveform-marker');
4491                existingMarkers.forEach(marker => marker.remove());
4492
4493                const existingLabels = document.querySelectorAll('.phonetic-label');
4494                existingLabels.forEach(label => label.remove());
4495
4496                const existingContainer = document.querySelector('.waveform-container');
4497                if (existingContainer) {
4498                    existingContainer.remove();
4499                }
4500
4501                // Get the table and first waveform column cell for positioning
4502                const table = document.getElementById('xsheet-table');
4503                const firstCell = document.querySelector('.waveform-col[data-frame="1"]');
4504                if (!firstCell) return;
4505
4506                // Calculate total height needed for the waveform
4507                const totalRows = document.querySelectorAll('tr[data-frame]').length;
4508                const rowHeight = firstCell.offsetHeight;
4509                const totalHeight = totalRows * rowHeight;
4510
4511                // Create a container for the vertical waveform that spans the entire table
4512                const waveformContainer = document.createElement('div');
4513                waveformContainer.className = 'waveform-container';
4514
4515                // Create a canvas for the waveform
4516                const canvas = document.createElement('canvas');
4517                canvas.className = 'waveform-canvas';
4518                canvas.width = firstCell.offsetWidth;
4519                canvas.height = totalHeight;
4520                waveformContainer.appendChild(canvas);
4521
4522                // Add overlay for event handling
4523                const overlay = document.createElement('div');
4524                overlay.className = 'waveform-overlay';
4525                overlay.style.height = totalHeight + 'px';
4526
4527                // Add event listener for clicking on the waveform
4528                overlay.addEventListener('click', function (e) {
4529                    if (!audioBuffer) return;
4530
4531                    const rect = overlay.getBoundingClientRect();
4532                    const y = e.clientY - rect.top;
4533                    const percentage = y / rect.height;
4534
4535                    // Calculate time point in the audio
```

```
                    const timePoint = percentage * audioBuffer.duration;

                    // Update audio position
                    audioScrubber.value = (timePoint / audioBuffer.duration) * 100;
                    if (isPlaying) {
                        stopAudio();
                        startOffset = timePoint;
                        playAudio();
                    } else {
                        startOffset = timePoint;
                    }

                    // Update the UI to show frame position
                    updateFrameMarker();
                });

                // Variables for scrubbing
                let isScrubbing = false;

                // Add scrubbing functionality (dragging while holding mouse button)
                overlay.addEventListener('mousedown', function (e) {
                    if (!audioBuffer) return;
                    if (e.button !== 0) return; // Only respond to left mouse button

                    isScrubbing = true;

                    // Pause any playing audio
                    if (isPlaying) {
                        pauseAudio();
                    }
                    // — NEW: allow pen (pointerType="pen") to scrub as well —
                    overlay.addEventListener('pointerdown', function (e) {
                        if (!audioBuffer || e.pointerType !== 'pen') return;
                        isScrubbing = true;
                        if (isPlaying) pauseAudio();          // pause continuous playback
                        updateScrubPosition(e);               // scrub to this location
                        e.preventDefault();                   // prevent default pen
behavior
                    });
                    overlay.addEventListener('pointermove', function (e) {
                        // — FIX: only scrub while the pen is physically pressing
(pressure>0)
                        if (isScrubbing && e.pointerType === 'pen' && e.pressure > 0) {
                            updateScrubPosition(e);
                        }
                    });
                    overlay.addEventListener('pointerup', function (e) {
                        if (e.pointerType === 'pen') {
                            isScrubbing = false;
                        }
```

```
                    });

                    // Create a scrub audio context if needed
                    if (!audioContext) {
                        try {
                            window.AudioContext = window.AudioContext ||
window.webkitAudioContext;
                            audioContext = new AudioContext();
                        } catch (e) {
                            console.error('Web Audio API not supported');
                            return;
                        }
                    }

                    // Initial scrub to current position
                    updateScrubPosition(e);

                    // Prevent text selection during drag
                    e.preventDefault();
                });

                // Handle scrubbing movement
                overlay.addEventListener('mousemove', function (e) {
                    if (!isScrubbing || !audioBuffer) return;
                    updateScrubPosition(e);
                });

                // Function to update position during scrubbing
                function updateScrubPosition(e) {
                    const rect = overlay.getBoundingClientRect();
                    const y = e.clientY - rect.top;
                    const percentage = Math.max(0, Math.min(1, y / rect.height));

                    // Calculate time point in the audio
                    const timePoint = percentage * audioBuffer.duration;
                    startOffset = timePoint;

                    // Update audio scrubber
                    audioScrubber.value = percentage * 100;

                    // Play a short snippet of audio at this position
                    playScrubAudio(timePoint);

                    // Update the marker
                    updateFrameMarker();
                }

                // End scrubbing when mouse is released or leaves element
                overlay.addEventListener('mouseup', function () {
                    isScrubbing = false;
```

```
4633                });
4634
4635                overlay.addEventListener('mouseleave', function () {
4636                    isScrubbing = false;
4637                });
4638
4639                // Function to play a short snippet at the scrub position
4640                let scrubSource = null;
4641                function playScrubAudio(timePoint) {
4642                    if (scrubSource) {
4643                        try {
4644                            scrubSource.stop();
4645                        } catch (e) {
4646                            // Ignore errors when stopping already stopped source
4647                        }
4648                    }
4649
4650                    // Play a very short snippet at the current position
4651                    scrubSource = audioContext.createBufferSource();
4652                    scrubSource.buffer = audioBuffer;
4653                    scrubSource.connect(audioContext.destination);
4654
4655                    // Play just a short snippet (equivalent to 1-2 frames at 24fps)
4656                    const snippetDuration = 1 / 12; // 1/12 of a second (2 frames at 24fps)
4657                    scrubSource.start(0, timePoint, snippetDuration);
4658                }
4659
4660                // Right-click to add phonetic marker
4661                overlay.addEventListener('contextmenu', function (e) {
4662                    e.preventDefault();
4663                    if (!audioBuffer) return;
4664
4665                    const rect = overlay.getBoundingClientRect();
4666                    const y = e.clientY - rect.top;
4667                    const percentage = y / rect.height;
4668
4669                    // Calculate time point in the audio
4670                    const timePoint = percentage * audioBuffer.duration;
4671
4672                    showPhoneticInput(timePoint);
4673                    return false;
4674                });
4675
4676                waveformContainer.appendChild(overlay);
4677
4678                // Add the container to the table
4679                document.body.appendChild(waveformContainer);
4680
4681                // Position the container over the waveform column
4682                const tableRect = table.getBoundingClientRect();
```

```
4683                const cellRect = firstCell.getBoundingClientRect();
4684
4685                waveformContainer.style.left = (cellRect.left - 1) + 'px';
4686                waveformContainer.style.top = (cellRect.top) + 'px';
4687                waveformContainer.style.width = (cellRect.width) + 'px';
4688                waveformContainer.style.height = totalHeight + 'px';
4689
4690                // Draw the waveform on the canvas
4691                const ctx = canvas.getContext('2d');
4692                const width = canvas.width;
4693                const height = canvas.height;
4694
4695                // Clear and set background
4696                ctx.clearRect(0, 0, width, height);
4697                ctx.fillStyle = '#ffffff';
4698                ctx.fillRect(0, 0, width, height);
4699
4700                // Calculate scaling factors
4701                const totalDuration = audioBuffer.duration;
4702                const framesPerSecond = 24;
4703                const totalFrames = Math.ceil(totalDuration * framesPerSecond);
4704
4705                // Draw center line
4706                ctx.beginPath();
4707                ctx.strokeStyle = '#cccccc';
4708                ctx.moveTo(width / 2, 0);
4709                ctx.lineTo(width / 2, height);
4710                ctx.stroke();
4711
4712                // Draw the waveform
4713                ctx.beginPath();
4714                ctx.strokeStyle = '#000000';
4715                ctx.lineWidth = 1;
4716
4717                // Map waveform data to vertical height
4718                for (let i = 0; i < waveformData.length; i++) {
4719                    const y = (i / waveformData.length) * height;
4720                    const amplitude = waveformData[i] * (width * 0.4); // Scale amplitude to
    40% of width
4721                    const x = (width / 2) + amplitude;
4722
4723                    if (i === 0) {
4724                        ctx.moveTo(x, y);
4725                    } else {
4726                        ctx.lineTo(x, y);
4727                    }
4728                }
4729
4730                // Draw mirrored waveform for visual effect
4731                for (let i = waveformData.length - 1; i >= 0; i--) {
```

```
                    const y = (i / waveformData.length) * height;
                    const amplitude = waveformData[i] * (width * 0.4);
                    const x = (width / 2) - amplitude;

                    ctx.lineTo(x, y);
                }

                ctx.stroke();

                // Draw frame markers
                for (let i = 1; i <= totalFrames && i <= totalRows; i++) {
                    const y = (i / totalRows) * height;

                    // Draw horizontal line at frame boundary
                    if (i % 8 === 0) {
                        ctx.beginPath();
                        ctx.strokeStyle = '#999999';
                        ctx.lineWidth = 1;
                        ctx.moveTo(0, y);
                        ctx.lineTo(width, y);
                        ctx.stroke();
                    }
                }

                // Render phonetic markers
                renderPhoneticMarkers();

                // Add the moving marker for current frame
                updateFrameMarker();
            }

            function renderPhoneticMarkers() {
                if (!phonetics || !audioBuffer) return;

                // Get the waveform container
                const waveformContainer = document.querySelector('.waveform-container');
                if (!waveformContainer) return;

                // Get container dimensions
                const containerHeight = waveformContainer.offsetHeight;

                // Add phonetic markers
                phonetics.forEach(phonetic => {
                    // Calculate vertical position based on time
                    const percentage = phonetic.time / audioBuffer.duration;
                    const yPosition = percentage * containerHeight;

                    // Create and position the marker
                    const label = document.createElement('div');
                    label.className = 'phonetic-label';
```

```
                    label.textContent = phonetic.text;
                    label.style.position = 'absolute';
                    label.style.left = '2px';
                    label.style.top = yPosition + 'px';
                    label.style.zIndex = '25';

                    // Add event listener to edit the phonetic marker
                    label.addEventListener('dblclick', function () {
                        showPhoneticInput(phonetic.time, phonetic.text,
phonetics.indexOf(phonetic));
                    });

                    waveformContainer.appendChild(label);
                });
            }

            function updateFrameMarker() {
                if (!audioBuffer) return;

                // Remove existing markers
                const existingMarkers = document.querySelectorAll('.waveform-marker');
                existingMarkers.forEach(marker => marker.remove());

                // Calculate which frame we're on
                const time = isPlaying ?
                    (audioContext.currentTime - startTime + startOffset) :
                    startOffset;

                const frame = Math.floor(time / frameDuration) + 1;
                currentFrame = frame;

                // Get the waveform container
                const waveformContainer = document.querySelector('.waveform-container');
                if (!waveformContainer) return;

                // Calculate vertical position based on time
                const containerHeight = waveformContainer.offsetHeight;
                const percentage = time / audioBuffer.duration;
                const yPosition = percentage * containerHeight;

                // Create horizontal marker line
                const marker = document.createElement('div');
                marker.className = 'waveform-marker';
                marker.style.position = 'absolute';
                marker.style.top = yPosition + 'px';
                marker.style.left = '0';
                marker.style.width = '100%';
                marker.style.height = '2px';
                marker.style.backgroundColor = 'rgba(255, 0, 0, 0.7)';
                marker.style.zIndex = '30';
```

```
4831
4832                // Add frame number
4833                marker.textContent = `Frame ${frame}`;
4834                marker.style.lineHeight = '0';
4835                marker.style.textAlign = 'right';
4836                marker.style.color = 'red';
4837                marker.style.fontWeight = 'bold';
4838                marker.style.fontSize = '8pt';
4839
4840                waveformContainer.appendChild(marker);
4841
4842                // Scroll to the marker if it's not visible and if we're playing
4843                if (isPlaying) {
4844                    const table = document.getElementById('xsheet-table');
4845                    const tableRect = table.getBoundingClientRect();
4846                    const markerY = tableRect.top + yPosition;
4847
4848                    // Check if marker is outside visible area
4849                    if (markerY < window.scrollY || markerY > window.scrollY +
    window.innerHeight) {
4850                        window.scrollTo({
4851                            top: markerY - (window.innerHeight / 2),
4852                            behavior: 'smooth'
4853                        });
4854                    }
4855                }
4856
4857                // Update scrubber position
4858                if (audioBuffer) {
4859                    const scrubPercentage = (time / audioBuffer.duration) * 100;
4860                    audioScrubber.value = scrubPercentage;
4861                }
4862
4863                // If playing, schedule the next update
4864                if (isPlaying) {
4865                    requestAnimationFrame(updateFrameMarker);
4866                }
4867            }
4868
4869            function togglePlayAudio() {
4870                if (isPlaying) {
4871                    pauseAudio();
4872                } else {
4873                    playAudio();
4874                }
4875            }
4876
4877            function playAudio() {
4878                if (!audioBuffer) return;
4879
```

```
4880                try {
4881                    // Create a new source node
4882                    audioSource = audioContext.createBufferSource();
4883                    audioSource.buffer = audioBuffer;
4884                    audioSource.connect(audioContext.destination);
4885
4886                    // Calculate start position
4887                    startTime = audioContext.currentTime;
4888
4889                    // Start playback from the current offset
4890                    audioSource.start(0, startOffset);
4891                    isPlaying = true;
4892
4893                    // Set up animation loop
4894                    updateFrameMarker();
4895
4896                    // Update UI
4897                    playAudioButton.textContent = 'Pause';
4898
4899                    // Set up ended event
4900                    audioSource.onended = function () {
4901                        if (isPlaying) {
4902                            stopAudio();
4903                        }
4904                    };
4905                } catch (e) {
4906                    updateStatusMessage('Error playing audio: ' + e.message);
4907                }
4908            }
4909
4910            function pauseAudio() {
4911                if (!isPlaying || !audioSource) return;
4912
4913                // Stop the audio
4914                audioSource.stop();
4915
4916                // Calculate current position
4917                startOffset += audioContext.currentTime - startTime;
4918
4919                isPlaying = false;
4920
4921                // Update UI
4922                playAudioButton.textContent = 'Play';
4923            }
4924
4925            function stopAudio() {
4926                if (audioSource) {
4927                    try {
4928                        audioSource.stop();
4929                    } catch (e) {
```

```
                        // Ignore errors when stopping already stopped source
                    }
                }

                isPlaying = false;
                startOffset = 0;

                // Update UI
                playAudioButton.textContent = 'Play';
                audioScrubber.value = 0;

                // Clear frame marker
                updateFrameMarker();
            }

            function scrubAudio() {
                if (!audioBuffer) return;

                // Calculate time from scrubber position
                const percentage = audioScrubber.value / 100;
                const newTime = percentage * audioBuffer.duration;

                // If playing, restart from new position
                if (isPlaying) {
                    pauseAudio();
                    startOffset = newTime;
                    playAudio();
                } else {
                    startOffset = newTime;
                    updateFrameMarker();
                }
            }

            function showPhoneticInput(time, initialText = '', editIndex = -1) {
                if (time === null) {
                    // If no time provided, use current position
                    time = isPlaying ?
                        (audioContext.currentTime - startTime + startOffset) :
                        startOffset;
                }

                // Store position for later use
                phoneticEditPosition = {
                    time: time,
                    editIndex: editIndex
                };

                // Set initial text if editing existing marker
                phoneticText.value = initialText;

```

```
                // Get the waveform container and calculate position
                const waveformContainer = document.querySelector('.waveform-container');
                if (waveformContainer) {
                    const containerRect = waveformContainer.getBoundingClientRect();
                    const containerHeight = waveformContainer.offsetHeight;
                    const percentage = time / audioBuffer.duration;
                    const yPosition = percentage * containerHeight;

                    // Position the input near the click position
                    phoneticInput.style.top = (containerRect.top + yPosition + window.scrollY) + 'px';
                    phoneticInput.style.left = (containerRect.right + window.scrollX + 5) + 'px';
                } else {
                    // Default position if container not found
                    phoneticInput.style.top = '200px';
                    phoneticInput.style.left = '200px';
                }

                // Show the input and focus it
                phoneticInput.style.display = 'block';
                phoneticText.focus();
            }

            function savePhoneticMarker() {
                if (!phoneticEditPosition) return;

                const text = phoneticText.value.trim();
                if (text === '') {
                    // If empty text and editing an existing marker, remove it
                    if (phoneticEditPosition.editIndex >= 0) {
                        phonetics.splice(phoneticEditPosition.editIndex, 1);
                    }
                } else {
                    // Save or update the phonetic marker
                    if (phoneticEditPosition.editIndex >= 0) {
                        // Update existing
                        phonetics[phoneticEditPosition.editIndex].text = text;
                    } else {
                        // Add new
                        phonetics.push({
                            time: phoneticEditPosition.time,
                            text: text
                        });
                    }
                }

                // Hide input
                phoneticInput.style.display = 'none';

```

```
                // Re-render markers
                renderWaveform();

                // Mark as modified
                modified = true;
            }

            function collectData() {
                const data = {
                    template: currentTemplate,
                    frameCount: frameCount,
                    metadata: {
                        projectNumber: document.getElementById('project-number').value,
                        date: document.getElementById('project-date').value,
                        pageNumber: document.getElementById('page-number').value,
                        animatorName: document.getElementById('animator-name').value,
                        versionNumber: document.getElementById('version-number').value,
                        shotNumber: document.getElementById('shot-number').value
                    },
                    audio: {
                        fileName: audioFileName,
                        phonetics: phonetics
                    },
                    rows: []
                };

                // Collect data from all rows
                const rows = tableBody.querySelectorAll('tr');
                rows.forEach(row => {
                    const rowData = {
                        action: row.cells[0].innerText,
                        frame: row.cells[1].innerText,
                        // Skip waveform cell (2)
                        dialogue: row.cells[3].innerText,
                        sound: row.cells[4].innerText,
                        technical: row.cells[5].innerText,
                        extra1: row.cells[6].innerText,
                        extra2: row.cells[7].innerText,
                        frameRepeat: row.cells[8].innerText,
                        camera: row.cells[9].innerText
                    };
                    data.rows.push(rowData);
                });

                return data;
            }

            function restoreData(data) {
                if (!data) return;

```

```
5078                // Update template and frame count
5079                currentTemplate = data.template || 'large';
5080                frameCount = data.frameCount || 96;
5081
5082                // Update UI to match
5083                templateSelector.value = currentTemplate;
5084                frameCountInput.value = frameCount;
5085
5086                // Restore metadata
5087                if (data.metadata) {
5088                    document.getElementById('project-number').value =
    data.metadata.projectNumber || '';
5089                    document.getElementById('project-date').value = data.metadata.date ||
    '';
5090                    document.getElementById('page-number').value = data.metadata.pageNumber
    || '';
5091                    document.getElementById('animator-name').value =
    data.metadata.animatorName || '';
5092                    document.getElementById('version-number').value =
    data.metadata.versionNumber || '';
5093                    document.getElementById('shot-number').value = data.metadata.shotNumber
    || '';
5094                }
5095
5096                // Generate the table with the right frame count
5097                generateTable(frameCount);
5098                updateTemplate();
5099
5100                // Restore audio data
5101                if (data.audio) {
5102                    audioFileName = data.audio.fileName || '';
5103                    phonetics = data.audio.phonetics || [];
5104
5105                    if (audioBuffer && phonetics.length > 0) {
5106                        renderWaveform();
5107                    }
5108                }
5109
5110                // Restore row data
5111                if (data.rows && data.rows.length > 0) {
5112                    const rows = tableBody.querySelectorAll('tr');
5113                    data.rows.forEach((rowData, index) => {
5114                        if (index < rows.length) {
5115                            rows[index].cells[0].innerText = rowData.action || '';
5116                            // Don't restore frame number cells as they're auto-generated
5117                            rows[index].cells[3].innerText = rowData.dialogue || '';
5118                            rows[index].cells[4].innerText = rowData.sound || '';
5119                            rows[index].cells[5].innerText = rowData.technical || '';
5120                            rows[index].cells[6].innerText = rowData.extra1 || '';
5121                            rows[index].cells[7].innerText = rowData.extra2 || '';
5122                            // Don't restore the second frame number cell
```

```
                            rows[index].cells[9].innerText = rowData.camera || '';
                        }
                    });
                }

                modified = false;
                updateStatusMessage('Project loaded successfully');
            }

            function saveProject() {
                const data = collectData();

                // Save to localStorage
                try {
                    localStorage.setItem('animationXSheet', JSON.stringify(data));
                    modified = false;
                    updateStatusMessage('Project saved successfully');

                    // Also download as JSON file for backup
                    const filename = projectName + '.json';
                    const dataStr = "data:text/json;charset=utf-8," +
encodeURIComponent(JSON.stringify(data));
                    const downloadAnchorNode = document.createElement('a');
                    downloadAnchorNode.setAttribute("href", dataStr);
                    downloadAnchorNode.setAttribute("download", filename);
                    document.body.appendChild(downloadAnchorNode);
                    downloadAnchorNode.click();
                    downloadAnchorNode.remove();
                } catch (e) {
                    updateStatusMessage('Error saving project: ' + e.message);
                }
            }

            function loadProject() {
                // First try to load from localStorage
                try {
                    const savedData = localStorage.getItem('animationXSheet');
                    if (savedData) {
                        restoreData(JSON.parse(savedData));
                        return;
                    }
                } catch (e) {
                    updateStatusMessage('Error loading saved project: ' + e.message);
                }

                // If no localStorage data, create file input for uploading JSON
                const fileInput = document.createElement('input');
                fileInput.type = 'file';
                fileInput.accept = '.json';
                fileInput.style.display = 'none';
```

```
5172                document.body.appendChild(fileInput);
5173
5174                fileInput.addEventListener('change', function (e) {
5175                    const file = e.target.files[0];
5176                    if (!file) return;
5177
5178                    const reader = new FileReader();
5179                    reader.onload = function (e) {
5180                        try {
5181                            const data = JSON.parse(e.target.result);
5182                            restoreData(data);
5183                        } catch (error) {
5184                            updateStatusMessage('Error parsing file: ' + error.message);
5185                        }
5186                    };
5187                    reader.readAsText(file);
5188                });
5189
5190                fileInput.click();
5191                fileInput.remove();
5192            }
5193
5194            function exportToPDF() {
5195                // Save the current state before PDF export
5196                const savedData = collectData();
5197
5198                updateStatusMessage('Preparing PDF. Please wait...');
5199
5200                // Remove controls temporarily for PDF generation
5201                const controls = document.querySelector('.controls');
5202                const audioControls = document.querySelector('#audio-controls');
5203                const statusMsg = document.querySelector('.status');
5204                const phoneticInputEl = document.querySelector('#phonetic-input');
5205
5206                controls.style.display = 'none';
5207                audioControls.style.display = 'none';
5208                statusMsg.style.display = 'none';
5209                phoneticInputEl.style.display = 'none';
5210
5211                // Clean up any previous waveform elements
5212                const previousContainers = document.querySelectorAll('.cell-waveform-
    window');
5213                previousContainers.forEach(container => container.remove());
5214
5215                // Draw the waveform directly into the cells
5216                if (audioBuffer && waveformData.length > 0) {
5217                    drawWaveformInCells();
5218                }
5219
5220                // Wait a moment for DOM updates to complete
```

```
                setTimeout(() => {
                    try {
                        // Force a redraw of the drawing layer before capture
                        if (window.xsheetDrawing && window.xsheetDrawing.layerSystem) {
                            window.xsheetDrawing.layerSystem.redrawAll();
                        }

                        // Use html2canvas to capture the printable area
                        html2canvas(document.getElementById('printable-area'), {
                            scale: 2, // Higher resolution
                            useCORS: true,
                            logging: false,
                            allowTaint: true,
                            onclone: function (clonedDoc) {
                                // Ensure drawing elements are visible in the cloned document
                                const clonedDrawingLayer = clonedDoc.querySelector('.drawing-layer-container');
                                if (clonedDrawingLayer) {
                                    clonedDrawingLayer.style.display = 'block';
                                    clonedDrawingLayer.style.visibility = 'visible';
                                    clonedDrawingLayer.style.opacity = '1';
                                }
                            }
                        }).then(canvas => {
                            const imgData = canvas.toDataURL('image/png');

                            // Determine PDF size based on template
                            let pdfWidth, pdfHeight;
                            if (currentTemplate === 'large') {
                                // 11"x17"
                                pdfWidth = 279.4; // mm
                                pdfHeight = 431.8; // mm
                            } else {
                                // 8.5"x11"
                                pdfWidth = 215.9; // mm
                                pdfHeight = 279.4; // mm
                            }

                            // Create PDF with jsPDF
                            const { jsPDF } = window.jspdf;
                            const pdf = new jsPDF({
                                orientation: 'portrait',
                                unit: 'mm',
                                format: [pdfWidth, pdfHeight]
                            });

                            // Calculate aspect ratio
                            const imgWidth = pdfWidth - 20; // margins
                            const imgHeight = canvas.height * imgWidth / canvas.width;
```

```
                            // Add image to PDF
                            pdf.addImage(imgData, 'PNG', 10, 10, imgWidth, imgHeight);

                            // Save PDF
                            pdf.save(`${projectName}.pdf`);

                            // Clean up after PDF generation using saved data instead of rebuilding
                            cleanupAfterExport(savedData);
                        });
                    } catch (e) {
                        // Clean up if there was an error
                        cleanupAfterExport(savedData);
                        updateStatusMessage('Error exporting PDF: ' + e.message);
                    }
                }, 100);

                function cleanupAfterExport(savedData) {
                    // Restore controls
                    controls.style.display = 'flex';
                    audioControls.style.display = 'flex';
                    statusMsg.style.display = 'block';

                    // Restore data instead of regenerating blank table
                    restoreData(savedData);

                    updateStatusMessage('PDF exported successfully');
                }

                function drawWaveformInCells() {
                    const waveformCells = document.querySelectorAll('.waveform-col');
                    if (waveformCells.length === 0) return;

                    // Calculate the number of data points per frame
                    const totalDuration = audioBuffer.duration;
                    const pointsPerFrame = waveformData.length / (totalDuration * 24);

                    // For each cell, draw its portion of the waveform
                    waveformCells.forEach((cell, index) => {
                        // Get the frame number (1-based)
                        const frameNum = index + 1;

                        // Create a canvas for this cell
                        const canvas = document.createElement('canvas');
                        canvas.width = cell.offsetWidth;
                        canvas.height = cell.offsetHeight;
                        canvas.style.display = 'block';
                        canvas.style.width = '100%';
                        canvas.style.height = '100%';
```

```
5318
5319                        const ctx = canvas.getContext('2d');
5320
5321                        // Fill with white background
5322                        ctx.fillStyle = 'white';
5323                        ctx.fillRect(0, 0, canvas.width, canvas.height);
5324
5325                        // Draw center line
5326                        ctx.beginPath();
5327                        ctx.strokeStyle = '#cccccc';
5328                        ctx.moveTo(canvas.width / 2, 0);
5329                        ctx.lineTo(canvas.width / 2, canvas.height);
5330                        ctx.stroke();
5331
5332                        // Calculate which section of the waveform to draw
5333                        const startPoint = Math.floor((frameNum - 1) * pointsPerFrame);
5334                        const endPoint = Math.floor(frameNum * pointsPerFrame);
5335
5336                        if (startPoint < waveformData.length) {
5337                            // Draw this portion of the waveform
5338                            ctx.beginPath();
5339                            ctx.strokeStyle = '#000000';
5340
5341                            // Check if we have valid data to draw
5342                            if (endPoint > startPoint) {
5343                                // Map the waveform section to this cell
5344                                for (let i = startPoint; i <= endPoint && i <
    waveformData.length; i++) {
5345                                    // Calculate position within this cell
5346                                    const relativePos = (i - startPoint) / (endPoint -
    startPoint);
5347                                    const y = relativePos * canvas.height;
5348
5349                                    // Draw waveform
5350                                    const amplitude = waveformData[i] * (canvas.width *
    0.4);
5351                                    const x = (canvas.width / 2) + amplitude;
5352
5353                                    if (i === startPoint) {
5354                                        ctx.moveTo(x, y);
5355                                    } else {
5356                                        ctx.lineTo(x, y);
5357                                    }
5358                                }
5359
5360                                // Draw left side (mirror)
5361                                for (let i = endPoint; i >= startPoint && i <
    waveformData.length; i--) {
5362                                    const relativePos = (i - startPoint) / (endPoint -
    startPoint);
5363                                    const y = relativePos * canvas.height;
```

```
5364
5365                                    const amplitude = waveformData[i] * (canvas.width *
    0.4);
5366                                    const x = (canvas.width / 2) - amplitude;
5367
5368                                    ctx.lineTo(x, y);
5369                                }
5370
5371                                ctx.stroke();
5372                            }
5373
5374                            // Look for phonetic markers that might be in this frame
5375                            if (phonetics && phonetics.length > 0) {
5376                                phonetics.forEach(phonetic => {
5377                                    // Calculate which frame this phonetic marker belongs to
5378                                    const frameOfMarker = Math.floor(phonetic.time /
    frameDuration) + 1;
5379
5380                                    // If it's in this frame, draw it
5381                                    if (frameOfMarker === frameNum) {
5382                                        // Calculate position within cell
5383                                        const posInFrameRatio = (phonetic.time - ((frameNum
    - 1) * frameDuration)) / frameDuration;
5384                                        const yPos = posInFrameRatio * canvas.height;
5385
5386                                        // Draw marker line
5387                                        ctx.beginPath();
5388                                        ctx.strokeStyle = '#ff0000';
5389                                        ctx.lineWidth = 1;
5390                                        ctx.moveTo(0, yPos);
5391                                        ctx.lineTo(canvas.width, yPos);
5392                                        ctx.stroke();
5393
5394                                        // Add label if it fits
5395                                        if (canvas.width > 30) {
5396                                            ctx.fillStyle = 'rgba(255, 255, 255, 0.7)';
5397                                            const textWidth =
    ctx.measureText(phonetic.text).width;
5398                                            if (textWidth < canvas.width - 4) {
5399                                                ctx.fillRect(2, yPos - 8, textWidth + 4,
    14);
5400                                                ctx.fillStyle = '#ff0000';
5401                                                ctx.font = '8px Arial';
5402                                                ctx.fillText(phonetic.text, 4, yPos + 4);
5403                                            }
5404                                        }
5405                                    }
5406                                });
5407                            }
5408                        }
5409
```

```
                        // Add frame number indicator
                        ctx.fillStyle = 'rgba(255, 255, 255, 0.7)';
                        ctx.fillRect(0, 0, 18, 12);
                        ctx.fillStyle = '#000000';
                        ctx.font = '8px Arial';
                        ctx.fillText(frameNum, 2, 8);

                        // Clear cell content and add the canvas
                        cell.innerHTML = '';
                        cell.appendChild(canvas);
                    });
                }
            }

            function printSheet() {
                updateStatusMessage('Preparing to print. Please wait...');

                // Save current selection state and cell contents
                const tableData = saveSelectionState();

                // Hide the waveform container temporarily but don't remove it
                const originalWf = document.querySelector('.waveform-container');
                if (originalWf) {
                    originalWf.style.display = 'none';
                }

                // Draw the waveform directly into the cells before printing
                if (audioBuffer && waveformData.length > 0) {
                    drawWaveformInCells();
                }

                // Wait a moment for DOM updates to complete
                setTimeout(() => {
                    window.print();

                    // Clean up after printing
                    setTimeout(() => {
                        // Restore the original waveform container
                        if (originalWf) {
                            originalWf.style.display = '';
                        }

                        // Restore original cell content for waveform cells only
                        const waveformCells = document.querySelectorAll('.waveform-col');
                        waveformCells.forEach(cell => {
                            const originalContent = cell.getAttribute('data-original-
content');
                            if (originalContent !== null) {
                                cell.innerHTML = originalContent;
                                cell.removeAttribute('data-original-content');
```

```
                            }
                        });

                        // Restore all cell contents from saved data
                        if (tableData && tableData.cellContents) {
                            tableData.cellContents.forEach(cellData => {
                                if (cellData.rowIndex >= 0 && cellData.rowIndex < tableBody.children.length) {
                                    const row = tableBody.children[cellData.rowIndex];
                                    if (cellData.colIndex >= 0 && cellData.colIndex < row.children.length) {
                                        const cell = row.children[cellData.colIndex];

                                        // Skip waveform cells (they were already restored above)
                                        if (!cell.classList.contains('waveform-col')) {
                                            // Only restore if cell is editable
                                            if (cell.contentEditable === 'true') {
                                                cell.innerHTML = cellData.content;

                                                // Restore modified status
                                                if (cellData.isModified) {
                                                    cell.classList.add('modified');
                                                } else {
                                                    cell.classList.remove('modified');
                                                }
                                            }
                                        }
                                    }
                                }
                            });
                        }

                        // Restore metadata fields
                        if (tableData && tableData.metadata) {
                            document.getElementById('project-number').value = tableData.metadata.projectNumber || '';
                            document.getElementById('project-date').value = tableData.metadata.date || '';
                            document.getElementById('page-number').value = tableData.metadata.pageNumber || '';
                            document.getElementById('animator-name').value = tableData.metadata.animatorName || '';
                            document.getElementById('version-number').value = tableData.metadata.versionNumber || '';
                            document.getElementById('shot-number').value = tableData.metadata.shotNumber || '';
                        }

                        // Finally restore cell selection state
                        restoreSelectionState(tableData);
```

```
5501
5502                        updateStatusMessage('Print complete');
5503                    }, 500);
5504                }, 100);
5505
5506                function drawWaveformInCells() {
5507                    const waveformCells = document.querySelectorAll('.waveform-col');
5508                    if (waveformCells.length === 0) return;
5509
5510                    // Calculate the number of data points per frame
5511                    const totalDuration = audioBuffer.duration;
5512                    const pointsPerFrame = waveformData.length / (totalDuration * 24);
5513
5514                    // For each cell, draw its portion of the waveform
5515                    waveformCells.forEach((cell, index) => {
5516                        // Get the frame number (1-based)
5517                        const frameNum = index + 1;
5518
5519                        // Create a canvas for this cell
5520                        const canvas = document.createElement('canvas');
5521                        canvas.width = cell.offsetWidth;
5522                        canvas.height = cell.offsetHeight;
5523                        canvas.style.display = 'block';
5524                        canvas.style.width = '100%';
5525                        canvas.style.height = '100%';
5526
5527                        const ctx = canvas.getContext('2d');
5528
5529                        // Fill with white background
5530                        ctx.fillStyle = 'white';
5531                        ctx.fillRect(0, 0, canvas.width, canvas.height);
5532
5533                        // Draw center line
5534                        ctx.beginPath();
5535                        ctx.strokeStyle = '#cccccc';
5536                        ctx.moveTo(canvas.width / 2, 0);
5537                        ctx.lineTo(canvas.width / 2, canvas.height);
5538                        ctx.stroke();
5539
5540                        // Calculate which section of the waveform to draw
5541                        const startPoint = Math.floor((frameNum - 1) * pointsPerFrame);
5542                        const endPoint = Math.floor(frameNum * pointsPerFrame);
5543
5544                        if (startPoint < waveformData.length) {
5545                            // Draw this portion of the waveform
5546                            ctx.beginPath();
5547                            ctx.strokeStyle = '#000000';
5548
5549                            // Check if we have valid data to draw
5550                            if (endPoint > startPoint) {
```

```
                                // Map the waveform section to this cell
                                for (let i = startPoint; i <= endPoint && i < waveformData.length; i++) {
                                    // Calculate position within this cell
                                    const relativePos = (i - startPoint) / (endPoint - startPoint);
                                    const y = relativePos * canvas.height;

                                    // Draw waveform
                                    const amplitude = waveformData[i] * (canvas.width * 0.4);
                                    const x = (canvas.width / 2) + amplitude;

                                    if (i === startPoint) {
                                        ctx.moveTo(x, y);
                                    } else {
                                        ctx.lineTo(x, y);
                                    }
                                }

                                // Draw left side (mirror)
                                for (let i = endPoint; i >= startPoint && i < waveformData.length; i--) {
                                    const relativePos = (i - startPoint) / (endPoint - startPoint);
                                    const y = relativePos * canvas.height;

                                    const amplitude = waveformData[i] * (canvas.width * 0.4);
                                    const x = (canvas.width / 2) - amplitude;

                                    ctx.lineTo(x, y);
                                }

                                ctx.stroke();
                            }

                            // Look for phonetic markers that might be in this frame
                            if (phonetics && phonetics.length > 0) {
                                phonetics.forEach(phonetic => {
                                    // Calculate which frame this phonetic marker belongs to
                                    const frameOfMarker = Math.floor(phonetic.time / frameDuration) + 1;

                                    // If it's in this frame, draw it
                                    if (frameOfMarker === frameNum) {
                                        // Calculate position within cell
                                        const posInFrameRatio = (phonetic.time - ((frameNum - 1) * frameDuration)) / frameDuration;
                                        const yPos = posInFrameRatio * canvas.height;

```

```
                                    // Draw marker line
                                    ctx.beginPath();
                                    ctx.strokeStyle = '#ff0000';
                                    ctx.lineWidth = 1;
                                    ctx.moveTo(0, yPos);
                                    ctx.lineTo(canvas.width, yPos);
                                    ctx.stroke();

                                    // Add label if it fits
                                    if (canvas.width > 30) {
                                        ctx.fillStyle = 'rgba(255, 255, 255, 0.7)';
                                        const textWidth =
ctx.measureText(phonetic.text).width;
                                        if (textWidth < canvas.width - 4) {
                                            ctx.fillRect(2, yPos - 8, textWidth + 4,
14);
                                            ctx.fillStyle = '#ff0000';
                                            ctx.font = '8px Arial';
                                            ctx.fillText(phonetic.text, 4, yPos + 4);
                                        }
                                    }
                                }
                            });
                        }
                    }

                    // Add frame number indicator
                    ctx.fillStyle = 'rgba(255, 255, 255, 0.7)';
                    ctx.fillRect(0, 0, 18, 12);
                    ctx.fillStyle = '#000000';
                    ctx.font = '8px Arial';
                    ctx.fillText(frameNum, 2, 8);

                    // Store original content and add the canvas
                    cell.setAttribute('data-original-content', cell.innerHTML);
                    cell.innerHTML = '';
                    cell.appendChild(canvas);
                });
            }

            function saveSelectionState() {
                // Save current selection and cell contents
                const tableData = {
                    // Save which cells are selected
                    selectedIndices: selectedCells.map(cell => {
                        const row = cell.closest('tr');
                        const rowIndex = Array.from(tableBody.children).indexOf(row);
                        const colIndex = Array.from(row.children).indexOf(cell);
                        return { rowIndex, colIndex };
                    }),
```

```
                        // Save metadata fields
                        metadata: {
                            projectNumber: document.getElementById('project-number').value,
                            date: document.getElementById('project-date').value,
                            pageNumber: document.getElementById('page-number').value,
                            animatorName: document.getElementById('animator-name').value,
                            versionNumber: document.getElementById('version-number').value,
                            shotNumber: document.getElementById('shot-number').value
                        },
                        // Store editable cells' content
                        cellContents: []
                    };

                    // Store all editable cells' content
                    const allEditableCells = document.querySelectorAll('[contenteditable="true"]');
                    allEditableCells.forEach(cell => {
                        const row = cell.closest('tr');
                        if (row) {
                            const rowIndex = Array.from(tableBody.children).indexOf(row);
                            const colIndex = Array.from(row.children).indexOf(cell);
                            tableData.cellContents.push({
                                rowIndex,
                                colIndex,
                                content: cell.innerHTML,
                                isModified: cell.classList.contains('modified')
                            });
                        }
                    });

                    return tableData;
                }

                function restoreSelectionState(tableData) {
                    if (!tableData || !tableData.selectedIndices) return;

                    // Clear current selection
                    clearCellSelection();

                    // Restore selection
                    tableData.selectedIndices.forEach(index => {
                        if (index.rowIndex >= 0 && index.rowIndex < tableBody.children.length) {
                            const row = tableBody.children[index.rowIndex];
                            if (index.colIndex >= 0 && index.colIndex < row.children.length) {
                                const cell = row.children[index.colIndex];
                                if (cell.contentEditable === 'true') {
                                    toggleCellSelection(cell, true);
                                }
                            }
```

```
                        }
                    });
                }
            }

            function addEightRows() {
                frameCount += 8;
                frameCountInput.value = frameCount;
                generateTable(frameCount);
                updateStatusMessage('Added 8 rows. Total frames: ' + frameCount);

                // Re-render waveform if audio is loaded
                if (audioBuffer) {
                    renderWaveform();
                }
            }

            function clearSheet() {
                if (confirm('Are you sure you want to clear all data? This cannot be undone.')) {
                    // Clear metadata
                    document.getElementById('project-number').value = '';
                    document.getElementById('page-number').value = '';
                    document.getElementById('animator-name').value = '';
                    document.getElementById('version-number').value = '';
                    document.getElementById('shot-number').value = '';

                    // Reset date to today
                    document.getElementById('project-date').valueAsDate = new Date();

                    // Clear all editable cells
                    const editableCells = document.querySelectorAll('[contenteditable="true"]');
                    editableCells.forEach(cell => {
                        cell.innerText = '';
                        cell.classList.remove('modified');
                    });

                    // Clear audio
                    audioBuffer = null;
                    audioSource = null;
                    waveformData = [];
                    phonetics = [];
                    audioFileName = '';
                    audioInfo.textContent = 'No audio loaded';

                    // Stop any playing audio
                    stopAudio();

                    // Clear waveform visualization
```

```
                    waveformCanvases.forEach(canvas => {
                        const ctx = canvas.getContext('2d');
                        ctx.clearRect(0, 0, canvas.width, canvas.height);
                    });

                    // Clear phonetic markers
                    const labels = document.querySelectorAll('.phonetic-label');
                    labels.forEach(label => label.remove());

                    // Clear drawings
                    if (window.xsheetDrawing && window.xsheetDrawing.layerSystem) {
                        window.xsheetDrawing.layerSystem.clearAllLayers();
                    }

                    modified = false;
                    updateStatusMessage('Sheet cleared');
                }
            }

            function updateStatusMessage(message) {
                statusMessage.textContent = message;
                console.log(message);

                // Clear status message after 3 seconds
                setTimeout(() => {
                    if (statusMessage.textContent === message) {
                        if (modified) {
                            statusMessage.textContent = 'Unsaved changes';
                        } else {
                            statusMessage.textContent = '';
                        }
                    }
                }, 3000);
            }

            // Auto-save timer every 2 minutes
            setInterval(() => {
                if (modified) {
                    try {
                        const data = collectData();
                        localStorage.setItem('animationXSheet_autosave',
JSON.stringify(data));
                        updateStatusMessage('Auto-saved');
                    } catch (e) {
                        console.error('Auto-save failed:', e);
                    }
                }
            }, 120000);

            // Check for auto-saved data on load
```

```
            try {
                const autoSavedData = localStorage.getItem('animationXSheet_autosave');
                if (autoSavedData && !localStorage.getItem('animationXSheet')) {
                    if (confirm('Found auto-saved data. Would you like to restore it?')) {
                        restoreData(JSON.parse(autoSavedData));
                    }
                }
            } catch (e) {
                console.error('Error checking for auto-saved data:', e);
            }

            // Initial status
            updateStatusMessage('X-Sheet ready');
        });
    </script>
</body>

</html>
```