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;
            page-break-after: avoid !important;
        }

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

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

<body>
    <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>

    <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();
            }

            init() {
                // Create container aligned with table
                const tableRect = this.xsheetTable.getBoundingClientRect();
                this.container = document.createElement('div');
                this.container.className = 'drawing-layer-container';
                this.container.style.position = 'absolute';
                this.container.style.left = `${tableRect.left}px`;
                this.container.style.top = `${tableRect.top}px`;
                this.container.style.width = `${tableRect.width}px`;
                this.container.style.height = `${tableRect.height}px`;
                this.container.style.pointerEvents = 'none'; // Initially pass events
through
                this.container.style.zIndex = '5';
                document.body.appendChild(this.container);

                // Create default background and foreground layers
                this.addLayer('background');
                this.addLayer('foreground');
                this.setActiveLayer(1); // Set foreground as active by default

                // Handle window resize and table changes
                this.setupResizeHandling();
            }

            addLayer(name) {
                const canvas = document.createElement('canvas');
                canvas.className = `drawing-layer-${name}`;
                canvas.width = this.container.clientWidth;
                canvas.height = this.container.clientHeight;
                canvas.style.position = 'absolute';
                canvas.style.left = '0';
                canvas.style.top = '0';
                canvas.style.pointerEvents = 'none';

                this.container.appendChild(canvas);

                const layer = {
                    name: name,
                    canvas: canvas,
                    context: canvas.getContext('2d'),
                    objects: [],
```

```
                    visible: true
                };

                this.layers.push(layer);
                return this.layers.length - 1; // Return index of new layer
            }

            setActiveLayer(index) {
                if (index >= 0 && index < this.layers.length) {
                    this.activeLayerIndex = index;
                    return true;
                }
                return false;
            }

            getActiveLayer() {
                return this.layers[this.activeLayerIndex];
            }
           updateLayoutSize() {
                const tableRect = this.xsheetTable.getBoundingClientRect();

                // Store current drawings from each layer
                const tempCanvases = this.layers.map(layer => {
                    const tempCanvas = document.createElement('canvas');
                    tempCanvas.width = layer.canvas.width;
                    tempCanvas.height = layer.canvas.height;
                    const tempCtx = tempCanvas.getContext('2d');
                    tempCtx.drawImage(layer.canvas, 0, 0);
                    return tempCanvas;
                });

                // Update container position and dimensions
                this.container.style.position = 'absolute';
                this.container.style.left = `${tableRect.left}px`;
                this.container.style.top = `${tableRect.top}px`;
                this.container.style.width = `${tableRect.width}px`;
                this.container.style.height = `${tableRect.height}px`;

                // Update each layer canvas
                this.layers.forEach((layer, i) => {
                    const scaleX = tableRect.width / layer.canvas.width;
                    const scaleY = tableRect.height / layer.canvas.height;

                    layer.canvas.width = tableRect.width;
                    layer.canvas.height = tableRect.height;

                    // Redraw with scaling
                    layer.context.save();
                    layer.context.scale(scaleX, scaleY);
                    layer.context.drawImage(tempCanvases[i], 0, 0);
```

```
                    layer.context.restore();
                });

                // Force redraw of all objects
                this.redrawAll();

                // Dispatch a custom event that the layout was updated
                document.dispatchEvent(new Event('drawing-layout-updated'));
            }

            enableDrawing() {
                this.layers.forEach(layer => {
                    layer.canvas.style.pointerEvents = 'auto';
                });
            }

            disableDrawing() {
                this.layers.forEach(layer => {
                    layer.canvas.style.pointerEvents = 'none';
                });
            }

            clearLayer(layerIndex) {
                if (layerIndex >= 0 && layerIndex < this.layers.length) {
                    const layer = this.layers[layerIndex];
                    layer.context.clearRect(0, 0, layer.canvas.width, layer.canvas.height);
                    layer.objects = [];
                }
            }

            clearAllLayers() {
                this.layers.forEach((layer, index) => {
                    this.clearLayer(index);
                });
            }

            // Convert screen coordinates to canvas coordinates
            screenToCanvas(screenX, screenY) {
                const containerRect = this.container.getBoundingClientRect();
                return {
                    x: screenX - containerRect.left,
                    y: screenY - containerRect.top
                };
            }

            // Convert frame/column to canvas coordinates (for multi-frame spanning)
            gridToCanvas(frame, column) {
                const cell = document.querySelector(`tr.frame-${frame} td:nth-
child(${column})`);
                if (!cell) return null;
```

```
691
692                 const cellRect = cell.getBoundingClientRect();
693                 const containerRect = this.container.getBoundingClientRect();
694
695                 return {
696                     x: cellRect.left - containerRect.left + cellRect.width / 2,
697                     y: cellRect.top - containerRect.top + cellRect.height / 2
698                 };
699             }
700
701             // Add a drawing object to the active layer
702             addObject(object) {
703                 const layer = this.getActiveLayer();
704                 layer.objects.push(object);
705                 this.redrawLayer(this.activeLayerIndex);
706                 return object;
707             }
708
709             // Redraw a specific layer
710             redrawLayer(layerIndex) {
711                 if (layerIndex >= 0 && layerIndex < this.layers.length) {
712                     const layer = this.layers[layerIndex];
713                     layer.context.clearRect(0, 0, layer.canvas.width, layer.canvas.height);
714
715                     // Draw all objects in this layer
716                     layer.objects.forEach(obj => {
717                         if (obj.visible) {
718                             obj.draw(layer.context);
719                         }
720                     });
721                 }
722             }
723
724             // Redraw all layers
725             redrawAll() {
726                 this.layers.forEach((_, index) => {
727                     this.redrawLayer(index);
728                 });
729             }
730
731             // Find object under point
732             findObjectAt(x, y) {
733                 // Check active layer first, then others in reverse order (top to bottom)
734                 const activeLayer = this.getActiveLayer();
735
736                 // Check active layer
737                 for (let i = activeLayer.objects.length - 1; i >= 0; i--) {
738                     const obj = activeLayer.objects[i];
739                     if (obj.containsPoint(x, y)) {
740                         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;
```

```
789                 this.type = 'drawingObject'; // Base type
790             }
791
792             draw(context) {
793                 // Base drawing functionality
794                 if (this.selected) {
795                     this.drawSelectionMarkers(context);
796                 }
797             }
798
799             drawSelectionMarkers(context) {
800                 // Draw selection handles (default implementation)
801                 context.save();
802                 context.strokeStyle = '#0099ff';
803                 context.lineWidth = 1;
804                 context.setLineDash([5, 3]);
805
806                 // Default is to draw a box around the object
807                 // This should be overridden by subclasses with specific bounds
808                 const bounds = this.getBounds();
809                 context.strokeRect(
810                     bounds.x - 2,
811                     bounds.y - 2,
812                     bounds.width + 4,
813                     bounds.height + 4
814                 );
815
816                 context.restore();
817             }
818
819             getBounds() {
820                 // Default implementation - should be overridden
821                 return { x: this.x, y: this.y, width: 0, height: 0 };
822             }
823
824             containsPoint(x, y) {
825                 // Default implementation - should be overridden
826                 return false;
827             }
828
829             move(dx, dy) {
830                 this.x += dx;
831                 this.y += dy;
832             }
833
834             toJSON() {
835                 return {
836                     type: this.type,
837                     x: this.x,
838                     y: this.y,
```

```
                    color: this.color,
                    lineWidth: this.lineWidth,
                    visible: this.visible
                };
            }

            static fromJSON(data) {
                // Factory method to create objects from JSON
                // This will be overridden by subclasses
                return new DrawingObject(data);
            }
        }

        // Line object
        class LineObject extends DrawingObject {
            constructor(props = {}) {
                super(props);
                this.x2 = props.x2 || 0;
                this.y2 = props.y2 || 0;
                this.type = 'line';
                this.dashPattern = props.dashPattern || [];
            }

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

                if (this.dashPattern.length > 0) {
                    context.setLineDash(this.dashPattern);
                }

                context.moveTo(this.x, this.y);
                context.lineTo(this.x2, this.y2);
                context.stroke();
                context.restore();

                super.draw(context);
            }

            getBounds() {
                const minX = Math.min(this.x, this.x2);
                const minY = Math.min(this.y, this.y2);
                const width = Math.abs(this.x2 - this.x);
                const height = Math.abs(this.y2 - this.y);

                return { x: minX, y: minY, width, height };
            }

```

```
889             containsPoint(x, y) {
890                 // Check if point is near the line
891                 const lineLength = Math.sqrt(
892                     Math.pow(this.x2 - this.x, 2) + Math.pow(this.y2 - this.y, 2)
893                 );
894 
895                 // If line is too short, use a minimum distance
896                 if (lineLength < 1) {
897                     const dx = x - this.x;
898                     const dy = y - this.y;
899                     return Math.sqrt(dx * dx + dy * dy) <= 5;
900                 }
901 
902                 // Calculate distance from point to line segment
903                 const t = ((x - this.x) * (this.x2 - this.x) + (y - this.y) * (this.y2 - this.y)) / (lineLength * lineLength);
904 
905                 if (t < 0) {
906                     // Point is beyond start point
907                     const dx = x - this.x;
908                     const dy = y - this.y;
909                     return Math.sqrt(dx * dx + dy * dy) <= 5;
910                 }
911 
912                 if (t > 1) {
913                     // Point is beyond end point
914                     const dx = x - this.x2;
915                     const dy = y - this.y2;
916                     return Math.sqrt(dx * dx + dy * dy) <= 5;
917                 }
918 
919                 // Calculate perpendicular distance
920                 const px = this.x + t * (this.x2 - this.x);
921                 const py = this.y + t * (this.y2 - this.y);
922                 const dx = x - px;
923                 const dy = y - py;
924                 return Math.sqrt(dx * dx + dy * dy) <= 5;
925             }
926 
927             move(dx, dy) {
928                 super.move(dx, dy);
929                 this.x2 += dx;
930                 this.y2 += dy;
931             }
932 
933             toJSON() {
934                 const json = super.toJSON();
935                 return {
936                     ...json,
937                     x2: this.x2,
```

```
                    y2: this.y2,
                    dashPattern: this.dashPattern
                };
            }

            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
        class RectangleObject extends DrawingObject {
            constructor(props = {}) {
                super(props);
                this.width = props.width || 0;
                this.height = props.height || 0;
                this.fill = props.fill || false;
                this.fillColor = props.fillColor || this.color;
                this.type = 'rectangle';
            }

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

                // Draw rectangle
                if (this.fill) {
                    context.fillStyle = this.fillColor;
                    context.fillRect(this.x, this.y, this.width, this.height);
                }
                context.strokeRect(this.x, this.y, this.width, this.height);
                context.restore();

                super.draw(context);
            }

            getBounds() {
                return {
                    x: this.x,
                    y: this.y,
                    width: this.width,
                    height: this.height
                };
            }

            containsPoint(x, y) {
                // Check if point is inside or near the edge of the rectangle
                if (this.fill) {
                    // For filled rectangles, check if point is inside
                    return (
                        x >= this.x && x <= this.x + this.width &&
                        y >= this.y && y <= this.y + this.height
                    );
                } else {
                    // For unfilled rectangles, check if point is near the edges
```

```
                    const nearLeft = Math.abs(x - this.x) <= 5;
                    const nearRight = Math.abs(x - (this.x + this.width)) <= 5;
                    const nearTop = Math.abs(y - this.y) <= 5;
                    const nearBottom = Math.abs(y - (this.y + this.height)) <= 5;

                    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
                );

                if (this.fill) {
                    context.fillStyle = this.fillColor;
                    context.fill();
                }
                context.stroke();
                context.restore();

                super.draw(context);
            }

            getBounds() {
                return {
                    x: this.x - this.radiusX,
                    y: this.y - this.radiusY,
                    width: this.radiusX * 2,
                    height: this.radiusY * 2
                };
            }

            containsPoint(x, y) {
                // Check if point is inside or near the edge of the ellipse
                const normalizedX = (x - this.x) / this.radiusX;
                const normalizedY = (y - this.y) / this.radiusY;
                const distance = Math.sqrt(normalizedX * normalizedX + normalizedY *
normalizedY);

                if (this.fill) {
                    // For filled ellipses, check if point is inside
                    return distance <= 1.0;
                } else {
                    // For unfilled ellipses, check if point is near the edge
                    return Math.abs(distance - 1.0) <= 5 / this.radiusX;
                }
            }

            toJSON() {
                const json = super.toJSON();
                return {
                    ...json,
                    radiusX: this.radiusX,
                    radiusY: this.radiusY,
                    fill: this.fill,
                    fillColor: this.fillColor
```

```
            };
        }

        static fromJSON(data) {
            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;
                }
```

```
1236
1237                    // Trigger redraw
1238                    document.dispatchEvent(new Event('xsheet-redraw'));
1239                };
1240                this.image.src = url;
1241            }
1242
1243            draw(context) {
1244                if (this.loaded && this.image) {
1245                    context.save();
1246                    context.drawImage(this.image, this.x, this.y, this.width, this.height);
1247                    context.restore();
1248                } else if (!this.loaded) {
1249                    // Draw placeholder while loading
1250                    context.save();
1251                    context.strokeStyle = '#999999';
1252                    context.lineWidth = 1;
1253                    context.strokeRect(this.x, this.y, this.width, this.height);
1254                    context.font = '10px Arial';
1255                    context.fillStyle = '#999999';
1256                    context.fillText('Loading Image...', this.x + 5, this.y + 15);
1257                    context.restore();
1258                }
1259
1260                super.draw(context);
1261            }
1262
1263            getBounds() {
1264                return {
1265                    x: this.x,
1266                    y: this.y,
1267                    width: this.width,
1268                    height: this.height
1269                };
1270            }
1271
1272            containsPoint(x, y) {
1273                return (
1274                    x >= this.x && x <= this.x + this.width &&
1275                    y >= this.y && y <= this.y + this.height
1276                );
1277            }
1278
1279            toJSON() {
1280                const json = super.toJSON();
1281                return {
1282                    ...json,
1283                    width: this.width,
1284                    height: this.height,
1285                    imageUrl: this.imageUrl
```

```
            };
        }

        static fromJSON(data) {
            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();
```

```
1336                context.translate(this.x, this.y);
1337                context.scale(this.scale, this.scale);
1338
1339                // Draw curved arrow going back
1340                context.beginPath();
1341                context.moveTo(0, 0);
1342                context.bezierCurveTo(-20, -5, -25, 10, -10, 15);
1343                context.stroke();
1344
1345                // Draw arrowhead
1346                context.beginPath();
1347                context.moveTo(-10, 15);
1348                context.lineTo(-5, 10);
1349                context.lineTo(-15, 5);
1350                context.closePath();
1351                context.fill();
1352
1353                context.restore();
1354            }
1355
1356            drawImpact(context) {
1357                context.save();
1358                context.translate(this.x, this.y);
1359                context.scale(this.scale, this.scale);
1360
1361                // Draw impact star
1362                for (let i = 0; i < 8; i++) {
1363                    const angle = (i / 8) * Math.PI * 2;
1364                    const innerRadius = 5;
1365                    const outerRadius = 15;
1366
1367                    context.beginPath();
1368                    context.moveTo(
1369                        innerRadius * Math.cos(angle),
1370                        innerRadius * Math.sin(angle)
1371                    );
1372                    context.lineTo(
1373                        outerRadius * Math.cos(angle),
1374                        outerRadius * Math.sin(angle)
1375                    );
1376                    context.stroke();
1377                }
1378
1379                context.restore();
1380            }
1381
1382            drawKeyframe(context) {
1383                context.save();
1384                context.translate(this.x, this.y);
1385                context.scale(this.scale, this.scale);
```

```
1386
1387                // Draw diamond
1388                context.beginPath();
1389                context.moveTo(0, -10);
1390                context.lineTo(10, 0);
1391                context.lineTo(0, 10);
1392                context.lineTo(-10, 0);
1393                context.closePath();
1394                context.stroke();
1395                context.fill();
1396
1397                context.restore();
1398            }
1399
1400            drawInbetween(context) {
1401                context.save();
1402                context.translate(this.x, this.y);
1403                context.scale(this.scale, this.scale);
1404
1405                // Draw circle
1406                context.beginPath();
1407                context.arc(0, 0, 7, 0, Math.PI * 2);
1408                context.stroke();
1409
1410                context.restore();
1411            }
1412
1413            drawHold(context) {
1414                context.save();
1415                context.translate(this.x, this.y);
1416                context.scale(this.scale, this.scale);
1417
1418                // Draw horizontal bar
1419                context.beginPath();
1420                context.moveTo(-15, 0);
1421                context.lineTo(15, 0);
1422                context.lineWidth = this.lineWidth * 2;
1423                context.stroke();
1424
1425                context.restore();
1426            }
1427
1428            drawDefault(context) {
1429                context.save();
1430                context.translate(this.x, this.y);
1431                context.scale(this.scale, this.scale);
1432
1433                // Draw square
1434                context.beginPath();
1435                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';
```

```
1486            }
1487
1488            addPoint(x, y) {
1489                this.points.push({ x, y });
1490            }
1491
1492            draw(context) {
1493                if (this.points.length < 2) return;
1494
1495                context.save();
1496                context.beginPath();
1497                context.strokeStyle = this.color;
1498                context.lineWidth = this.lineWidth;
1499                context.lineJoin = 'round';
1500                context.lineCap = 'round';
1501
1502                // Start from first point
1503                context.moveTo(this.points[0].x, this.points[0].y);
1504
1505                if (this.smoothing && this.points.length > 2) {
1506                    // Draw using bezier curves for smoothing
1507                    for (let i = 1; i < this.points.length - 1; i++) {
1508                        const p1 = this.points[i];
1509                        const p2 = this.points[i + 1];
1510
1511                        const xc = (p1.x + p2.x) / 2;
1512                        const yc = (p1.y + p2.y) / 2;
1513
1514                        context.quadraticCurveTo(p1.x, p1.y, xc, yc);
1515                    }
1516
1517                    // Connect to the last point
1518                    const last = this.points[this.points.length - 1];
1519                    context.lineTo(last.x, last.y);
1520                } else {
1521                    // Simple line segments
1522                    for (let i = 1; i < this.points.length; i++) {
1523                        context.lineTo(this.points[i].x, this.points[i].y);
1524                    }
1525                }
1526
1527                if (this.closed) {
1528                    context.closePath();
1529                }
1530
1531                if (this.fill) {
1532                    context.fillStyle = this.fillColor;
1533                    context.fill();
1534                }
1535
```

```
1536                context.stroke();
1537                context.restore();
1538
1539                super.draw(context);
1540            }
1541
1542            getBounds() {
1543                if (this.points.length === 0) {
1544                    return { x: this.x, y: this.y, width: 0, height: 0 };
1545                }
1546
1547                let minX = this.points[0].x;
1548                let maxX = this.points[0].x;
1549                let minY = this.points[0].y;
1550                let maxY = this.points[0].y;
1551
1552                // Find min/max coordinates
1553                for (let i = 1; i < this.points.length; i++) {
1554                    const point = this.points[i];
1555                    minX = Math.min(minX, point.x);
1556                    maxX = Math.max(maxX, point.x);
1557                    minY = Math.min(minY, point.y);
1558                    maxY = Math.max(maxY, point.y);
1559                }
1560
1561                return {
1562                    x: minX,
1563                    y: minY,
1564                    width: maxX - minX,
1565                    height: maxY - minY
1566                };
1567            }
1568
1569            containsPoint(x, y) {
1570                if (this.points.length < 2) return false;
1571
1572                // Check if point is near any line segment
1573                for (let i = 0; i < this.points.length - 1; i++) {
1574                    const p1 = this.points[i];
1575                    const p2 = this.points[i + 1];
1576
1577                    const lineLength = Math.sqrt(
1578                        Math.pow(p2.x - p1.x, 2) + Math.pow(p2.y - p1.y, 2)
1579                    );
1580
1581                    // If line is too short, check distance to point
1582                    if (lineLength < 1) {
1583                        const dx = x - p1.x;
1584                        const dy = y - p1.y;
1585                        if (Math.sqrt(dx * dx + dy * dy) <= 5) {
```

```
1586                            return true;
1587                        }
1588                        continue;
1589                    }
1590
1591                    // Calculate distance from point to line segment
1592                    const t = ((x - p1.x) * (p2.x - p1.x) + (y - p1.y) * (p2.y - p1.y)) /
    (lineLength * lineLength);
1593
1594                    if (t < 0) {
1595                        // Point is beyond start point
1596                        const dx = x - p1.x;
1597                        const dy = y - p1.y;
1598                        if (Math.sqrt(dx * dx + dy * dy) <= 5) {
1599                            return true;
1600                        }
1601                    } else if (t > 1) {
1602                        // Point is beyond end point
1603                        const dx = x - p2.x;
1604                        const dy = y - p2.y;
1605                        if (Math.sqrt(dx * dx + dy * dy) <= 5) {
1606                            return true;
1607                        }
1608                    } else {
1609                        // Calculate perpendicular distance
1610                        const px = p1.x + t * (p2.x - p1.x);
1611                        const py = p1.y + t * (p2.y - p1.y);
1612                        const dx = x - px;
1613                        const dy = y - py;
1614                        if (Math.sqrt(dx * dx + dy * dy) <= 5) {
1615                            return true;
1616                        }
1617                    }
1618                }
1619
1620                // If closed and filled, also check if point is inside
1621                if (this.closed && this.fill) {
1622                    // Use point-in-polygon algorithm
1623                    let inside = false;
1624                    for (let i = 0, j = this.points.length - 1; i < this.points.length; j =
    i++) {
1625                        const xi = this.points[i].x;
1626                        const yi = this.points[i].y;
1627                        const xj = this.points[j].x;
1628                        const yj = this.points[j].y;
1629
1630                        const intersect = ((yi > y) !== (yj > y)) &&
1631                            (x < (xj - xi) * (y - yi) / (yj - yi) + xi);
1632
1633                        if (intersect) inside = !inside;
```

```
                    }
                    return inside;
                }

                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);
                const endPos = layerSystem.gridToCanvas(this.endFrame, this.endColumn);

                if (!startPos || !endPos) return; // Skip if cells not found

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

                if (this.dashPattern.length > 0) {
                    context.setLineDash(this.dashPattern);
                }

                context.moveTo(startPos.x, startPos.y);
                context.lineTo(endPos.x, endPos.y);
                context.stroke();

                // Draw arrows if needed
                if (this.arrowStart) {
                    this.drawArrow(context, endPos.x, endPos.y, startPos.x, startPos.y);
                }

                if (this.arrowEnd) {
                    this.drawArrow(context, startPos.x, startPos.y, endPos.x, endPos.y);
                }

                context.restore();

                // Store computed coordinates for selection/hit testing
                this.computedStart = startPos;
                this.computedEnd = endPos;

                super.draw(context);
            }

            drawArrow(context, fromX, fromY, toX, toY) {
                const angle = Math.atan2(toY - fromY, toX - fromX);

                context.save();
                context.fillStyle = this.color;
                context.beginPath();
                context.moveTo(toX, toY);
                context.lineTo(
                    toX - this.arrowSize * Math.cos(angle - Math.PI/6),
                    toY - this.arrowSize * Math.sin(angle - Math.PI/6)
```

```
                );
                context.lineTo(
                    toX - this.arrowSize * Math.cos(angle + Math.PI/6),
                    toY - this.arrowSize * Math.sin(angle + Math.PI/6)
                );
                context.closePath();
                context.fill();
                context.restore();
            }

            getBounds() {
                if (!this.computedStart || !this.computedEnd) return { x: 0, y: 0, width: 0, height: 0 };

                const minX = Math.min(this.computedStart.x, this.computedEnd.x);
                const minY = Math.min(this.computedStart.y, this.computedEnd.y);
                const width = Math.abs(this.computedEnd.x - this.computedStart.x);
                const height = Math.abs(this.computedEnd.y - this.computedStart.y);

                return { x: minX, y: minY, width, height };
            }

            containsPoint(x, y) {
                if (!this.computedStart || !this.computedEnd) return false;

                // Same algorithm as LineObject
                const lineLength = Math.sqrt(
                    Math.pow(this.computedEnd.x - this.computedStart.x, 2) +
                    Math.pow(this.computedEnd.y - this.computedStart.y, 2)
                );

                if (lineLength < 1) {
                    const dx = x - this.computedStart.x;
                    const dy = y - this.computedStart.y;
                    return Math.sqrt(dx * dx + dy * dy) <= 5;
                }

                const t = ((x - this.computedStart.x) * (this.computedEnd.x - this.computedStart.x) +
                           (y - this.computedStart.y) * (this.computedEnd.y - this.computedStart.y)) /
                          (lineLength * lineLength);

                if (t < 0) {
                    const dx = x - this.computedStart.x;
                    const dy = y - this.computedStart.y;
                    return Math.sqrt(dx * dx + dy * dy) <= 5;
                }

                if (t > 1) {
                    const dx = x - this.computedEnd.x;
```

```
                    const dy = y - this.computedEnd.y;
                    return Math.sqrt(dx * dx + dy * dy) <= 5;
                }

                const px = this.computedStart.x + t * (this.computedEnd.x - this.computedStart.x);
                const py = this.computedStart.y + t * (this.computedEnd.y - this.computedStart.y);
                const dx = x - px;
                const dy = y - py;
                return Math.sqrt(dx * dx + dy * dy) <= 5;
            }

            // This object type doesn't use the regular move method
            // Instead, it updates the frame/column values

            toJSON() {
                const json = super.toJSON();
                return {
                    ...json,
                    startFrame: this.startFrame,
                    startColumn: this.startColumn,
                    endFrame: this.endFrame,
                    endColumn: this.endColumn,
                    dashPattern: this.dashPattern,
                    arrowStart: this.arrowStart,
                    arrowEnd: this.arrowEnd,
                    arrowSize: this.arrowSize
                };
            }

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

        // Register all object types in a factory
        const DrawingObjectFactory = {
            types: {
                'drawingObject': DrawingObject,
                'line': LineObject,
                'arrow': ArrowObject,
                'rectangle': RectangleObject,
                'ellipse': EllipseObject,
                'text': TextObject,
                'image': ImageObject,
                'symbol': SymbolObject,
                'path': PathObject,
                'frameSpanningLine': FrameSpanningLineObject
            },
```

```
1829
1830            createFromJSON(data) {
1831                const Type = this.types[data.type];
1832                if (Type) {
1833                    return Type.fromJSON(data);
1834                }
1835                return null;
1836            }
1837        };
1838
1839        /**
1840         * DRAWING TOOL SYSTEM
1841         * Manages the drawing tools and interfaces with the layer system
1842         */
1843        class DrawingToolSystem {
1844            constructor(layerSystem) {
1845                this.layerSystem = layerSystem;
1846                this.activeTool = null;
1847                this.toolSettings = {
1848                    color: '#ff0000',
1849                    lineWidth: 2,
1850                    fill: false,
1851                    fillColor: '#ff8080',
1852                    fontSize: 16,
1853                    fontFamily: 'Arial, sans-serif',
1854                    textAlign: 'left',
1855                    symbolType: 'default',
1856                    symbolScale: 1.0
1857                };
1858
1859                this.availableTools = {
1860                    'select': new SelectTool(this),
1861                    'pen': new PenTool(this),
1862                    'line': new LineTool(this),
1863                    'arrow': new ArrowTool(this),
1864                    'rectangle': new RectangleTool(this),
1865                    'ellipse': new EllipseTool(this),
1866                    'text': new TextTool(this),
1867                    'image': new ImageTool(this),
1868                    'symbol': new SymbolTool(this),
1869                    'frameLine': new FrameSpanningLineTool(this),
1870                    'eraser': new EraserTool(this)
1871                };
1872
1873                // Default to select tool
1874                this.setActiveTool('select');
1875
1876                // Set up toolbar UI
1877                this.createToolbar();
1878            }
```

```
1879
1880            setActiveTool(toolName) {
1881                if (this.activeTool) {
1882                    this.activeTool.deactivate();
1883                }
1884
1885                if (this.availableTools[toolName]) {
1886                    this.activeTool = this.availableTools[toolName];
1887                    this.activeTool.activate();
1888                    return true;
1889                }
1890
1891                return false;
1892            }
1893
1894            createToolbar() {
1895                // Create toolbar container
1896                const toolbar = document.createElement('div');
1897                toolbar.className = 'drawing-toolbar';
1898                toolbar.style.display = 'flex';
1899                toolbar.style.flexWrap = 'wrap';
1900                toolbar.style.gap = '5px';
1901                toolbar.style.padding = '10px';
1902                toolbar.style.backgroundColor = '#f5f5f5';
1903                toolbar.style.marginBottom = '10px';
1904                toolbar.style.borderRadius = '5px';
1905
1906                // Add tool buttons
1907                this.addToolButton(toolbar, 'select', 'Select', '👆');
1908                this.addToolButton(toolbar, 'pen', 'Freehand Drawing', '🖍️');
1909                this.addToolButton(toolbar, 'line', 'Line', '—');
1910                this.addToolButton(toolbar, 'arrow', 'Arrow', '→');
1911                this.addToolButton(toolbar, 'rectangle', 'Rectangle', '□');
1912                this.addToolButton(toolbar, 'ellipse', 'Circle/Ellipse', 'o');
1913                this.addToolButton(toolbar, 'text', 'Text', 'T');
1914                this.addToolButton(toolbar, 'image', 'Insert Image', '🖼️');
1915                this.addToolButton(toolbar, 'symbol', 'Animation Symbol', '⭐');
1916                this.addToolButton(toolbar, 'frameLine', 'Multi-Frame Line', '↕️');
1917
1918                // Add separator
1919                toolbar.appendChild(document.createElement('div')).style.borderLeft = '1px
    solid #ccc';
1920                toolbar.lastChild.style.height = '30px';
1921
1922                // Add color picker
1923                const colorContainer = document.createElement('div');
1924                colorContainer.style.display = 'flex';
1925                colorContainer.style.alignItems = 'center';
1926                colorContainer.style.gap = '5px';
1927
```

```
                const colorLabel = document.createElement('label');
                colorLabel.textContent = 'Color:';
                colorLabel.htmlFor = 'drawing-color';

                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');
```

```
1978                fillContainer.style.display = 'flex';
1979                fillContainer.style.alignItems = 'center';
1980                fillContainer.style.gap = '5px';
1981
1982                const fillCheck = document.createElement('input');
1983                fillCheck.type = 'checkbox';
1984                fillCheck.id = 'drawing-fill';
1985                fillCheck.checked = this.toolSettings.fill;
1986
1987                const fillLabel = document.createElement('label');
1988                fillLabel.textContent = 'Fill';
1989                fillLabel.htmlFor = 'drawing-fill';
1990
1991                fillCheck.addEventListener('change', (e) => {
1992                    this.toolSettings.fill = e.target.checked;
1993                    fillColorPicker.disabled = !e.target.checked;
1994                });
1995
1996                const fillColorPicker = document.createElement('input');
1997                fillColorPicker.type = 'color';
1998                fillColorPicker.id = 'drawing-fill-color';
1999                fillColorPicker.value = this.toolSettings.fillColor;
2000                fillColorPicker.disabled = !this.toolSettings.fill;
2001
2002                fillColorPicker.addEventListener('input', (e) => {
2003                    this.toolSettings.fillColor = e.target.value;
2004                });
2005
2006                fillContainer.appendChild(fillCheck);
2007                fillContainer.appendChild(fillLabel);
2008                fillContainer.appendChild(fillColorPicker);
2009                toolbar.appendChild(fillContainer);
2010
2011                // Add separator
2012                toolbar.appendChild(document.createElement('div')).style.borderLeft = '1px
    solid #ccc';
2013                toolbar.lastChild.style.height = '30px';
2014
2015                // Add layer selector
2016                const layerSelector = document.createElement('select');
2017                layerSelector.id = 'drawing-layer-selector';
2018
2019                // Add options for each layer
2020                this.layerSystem.layers.forEach((layer, index) => {
2021                    const option = document.createElement('option');
2022                    option.value = index;
2023                    option.textContent = layer.name;
2024                    if (index === this.layerSystem.activeLayerIndex) {
2025                        option.selected = true;
2026                    }
```

```
                    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;
            }
```

```
2126
2127            activate() {
2128                this.active = true;
2129                this.layerSystem.enableDrawing();
2130                this.attachEvents();
2131            }
2132
2133            deactivate() {
2134                this.active = false;
2135                // turn pointer-events back off so clicks go to the table cells
2136                this.toolSystem.layerSystem.disableDrawing();
2137                this.detachEvents();
2138            }
2139
2140            attachEvents() {
2141                // Override in subclasses
2142            }
2143
2144            detachEvents() {
2145                // Override in subclasses
2146            }
2147        }
2148
2149        // Select tool for manipulating objects
2150        class SelectTool extends DrawingTool {
2151
2152            constructor(toolSystem) {
2153                super(toolSystem);
2154                this.selectedObject = null;
2155                this.selectedLayer = null;
2156                this.dragging = false;
2157                this.dragStart = { x: 0, y: 0 };
2158                this.objectStart = { x: 0, y: 0 };
2159            }
2160            activate() {
2161                this.active = true;
2162                // let pointer-events go to the table cells instead of the canvases
2163                this.layerSystem.disableDrawing();
2164                this.attachEvents();
2165            }
2166
2167            deactivate() {
2168                this.active = false;
2169                this.layerSystem.disableDrawing();
2170                this.detachEvents();
2171            }
2172            attachEvents() {
2173                const canvas = this.layerSystem.getActiveLayer().canvas;
2174
2175                canvas.addEventListener('mousedown', this.handleMouseDown);
```

```
2176                canvas.addEventListener('pointerdown', this.handleMouseDown);
2177                document.addEventListener('mousemove', this.handleMouseMove);
2178                document.addEventListener('pointermove', this.handleMouseMove);
2179                document.addEventListener('mouseup', this.handleMouseUp);
2180                document.addEventListener('pointerup',   this.handleMouseUp);
2181                document.addEventListener('keydown', this.handleKeyDown);
2182            }
2183
2184            detachEvents() {
2185                const canvas = this.layerSystem.getActiveLayer().canvas;
2186
2187                canvas.removeEventListener('mousedown', this.handleMouseDown);
2188                canvas.removeEventListener('pointerdown', this.handleMouseDown);
2189                document.removeEventListener('mousemove', this.handleMouseMove);
2190                document.removeEventListener('pointermove', this.handleMouseMove);
2191                document.removeEventListener('mouseup', this.handleMouseUp);
2192                document.removeEventListener('pointerup',   this.handleMouseUp);
2193                document.removeEventListener('keydown', this.handleKeyDown);
2194
2195                // Clear selection
2196                this.clearSelection();
2197            }
2198
2199            handleMouseDown = (e) => {
2200                // Convert to canvas coordinates
2201                const coords = this.layerSystem.screenToCanvas(e.clientX, e.clientY);
2202
2203                // Check if clicked on an object
2204                const hit = this.layerSystem.findObjectAt(coords.x, coords.y);
2205
2206                if (hit) {
2207                    // Select the object
2208                    this.selectObject(hit.object, hit.layerIndex);
2209
2210                    // Start drag
2211                    this.dragging = true;
2212                    this.dragStart = { x: coords.x, y: coords.y };
2213                    this.objectStart = { x: hit.object.x, y: hit.object.y };
2214                } else {
2215                    // Clear selection if clicked empty space
2216                    this.clearSelection();
2217                }
2218            }
2219
2220            handleMouseMove = (e) => {
2221                if (!this.dragging || !this.selectedObject) return;
2222
2223                const coords = this.layerSystem.screenToCanvas(e.clientX, e.clientY);
2224
2225                // Calculate move distance
```

```
                const dx = coords.x - this.dragStart.x;
                const dy = coords.y - this.dragStart.y;

                // 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);
```

```
2324
2325                // Finish any in-progress drawing
2326                this.finishDrawing();
2327            }
2328
2329            handleMouseDown = (e) => {
2330                e.preventDefault();
2331                if (e.pointerId != null) {
2332                e.target.setPointerCapture(e.pointerId);
2333                }
2334
2335                const coords = this.layerSystem.screenToCanvas(e.clientX, e.clientY);
2336
2337                // Start a new path
2338                this.currentPath = new PathObject({
2339                    color: this.settings.color,
2340                    lineWidth: this.settings.lineWidth,
2341                    x: coords.x,
2342                    y: coords.y
2343                });
2344
2345                this.currentPath.addPoint(coords.x, coords.y);
2346                this.drawing = true;
2347
2348                // Add to layer
2349                this.layerSystem.addObject(this.currentPath);
2350            }
2351
2352            handleMouseMove = (e) => {
2353                if (!this.drawing || !this.currentPath) return;
2354
2355                const coords = this.layerSystem.screenToCanvas(e.clientX, e.clientY);
2356
2357                // Add point to path
2358                this.currentPath.addPoint(coords.x, coords.y);
2359
2360                // Redraw
2361                this.layerSystem.redrawLayer(this.layerSystem.activeLayerIndex);
2362            }
2363
2364            handleMouseUp = (e) => {
2365                if (e.pointerId != null) {
2366                  e.target.releasePointerCapture(e.pointerId);
2367                }
2368                this.finishDrawing();
2369            }
2370
2371            finishDrawing() {
2372                if (this.drawing && this.currentPath) {
2373                    // Finish the path
```

```
                    this.drawing = false;
                    this.currentPath = null;
                }
            }
        }

        // 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;
```

```
2424
2425                // Add to layer
2426                this.layerSystem.addObject(this.currentLine);
2427            }
2428
2429            handleMouseMove = (e) => {
2430                if (!this.drawing || !this.currentLine) return;
2431
2432                const coords = this.layerSystem.screenToCanvas(e.clientX, e.clientY);
2433
2434                // Update end point
2435                this.currentLine.x2 = coords.x;
2436                this.currentLine.y2 = coords.y;
2437
2438                // Redraw
2439                this.layerSystem.redrawLayer(this.layerSystem.activeLayerIndex);
2440            }
2441
2442            handleMouseUp = () => {
2443                this.finishDrawing();
2444            }
2445
2446            finishDrawing() {
2447                if (this.drawing && this.currentLine) {
2448                    // Finish the line
2449                    this.drawing = false;
2450                    this.startPoint = null;
2451                    this.currentLine = null;
2452                }
2453            }
2454        }
2455
2456        // Arrow tool (extends Line tool)
2457        class ArrowTool extends LineTool {
2458            handleMouseDown = (e) => {
2459                const coords = this.layerSystem.screenToCanvas(e.clientX, e.clientY);
2460
2461                this.startPoint = coords;
2462
2463                // Create temporary arrow
2464                this.currentLine = new ArrowObject({
2465                    x: coords.x,
2466                    y: coords.y,
2467                    x2: coords.x,
2468                    y2: coords.y,
2469                    color: this.settings.color,
2470                    lineWidth: this.settings.lineWidth
2471                });
2472
2473                this.drawing = true;
```

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

        // 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
                });
```

```
2524
2525                this.drawing = true;
2526
2527                // Add to layer
2528                this.layerSystem.addObject(this.currentRect);
2529            }
2530
2531            handleMouseMove = (e) => {
2532                if (!this.drawing || !this.currentRect) return;
2533
2534                const coords = this.layerSystem.screenToCanvas(e.clientX, e.clientY);
2535
2536                // Update dimensions
2537                const width = coords.x - this.startPoint.x;
2538                const height = coords.y - this.startPoint.y;
2539
2540                if (width < 0) {
2541                    this.currentRect.x = coords.x;
2542                    this.currentRect.width = Math.abs(width);
2543                } else {
2544                    this.currentRect.x = this.startPoint.x;
2545                    this.currentRect.width = width;
2546                }
2547
2548                if (height < 0) {
2549                    this.currentRect.y = coords.y;
2550                    this.currentRect.height = Math.abs(height);
2551                } else {
2552                    this.currentRect.y = this.startPoint.y;
2553                    this.currentRect.height = height;
2554                }
2555
2556                // Redraw
2557                this.layerSystem.redrawLayer(this.layerSystem.activeLayerIndex);
2558            }
2559
2560            handleMouseUp = () => {
2561                this.finishDrawing();
2562            }
2563
2564            finishDrawing() {
2565                if (this.drawing && this.currentRect) {
2566                    // Finish the rectangle
2567                    this.drawing = false;
2568                    this.startPoint = null;
2569                    this.currentRect = null;
2570                }
2571            }
2572        }
2573
```

```
        // Ellipse tool
        class EllipseTool extends DrawingTool {
            constructor(toolSystem) {
                super(toolSystem);
                this.center = null;
                this.currentEllipse = 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.center = coords;

                // Create temporary ellipse
                this.currentEllipse = new EllipseObject({
                    x: coords.x,
                    y: coords.y,
                    radiusX: 0,
                    radiusY: 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.currentEllipse);
            }
```

```
2624
2625            handleMouseMove = (e) => {
2626                if (!this.drawing || !this.currentEllipse) return;
2627
2628                const coords = this.layerSystem.screenToCanvas(e.clientX, e.clientY);
2629
2630                // Update radii
2631                this.currentEllipse.radiusX = Math.abs(coords.x - this.center.x);
2632                this.currentEllipse.radiusY = Math.abs(coords.y - this.center.y);
2633
2634                // Redraw
2635                this.layerSystem.redrawLayer(this.layerSystem.activeLayerIndex);
2636            }
2637
2638            handleMouseUp = () => {
2639                this.finishDrawing();
2640            }
2641
2642            finishDrawing() {
2643                if (this.drawing && this.currentEllipse) {
2644                    // Finish the ellipse
2645                    this.drawing = false;
2646                    this.center = null;
2647                    this.currentEllipse = null;
2648                }
2649            }
2650        }
2651
2652        // Text tool
2653        class TextTool extends DrawingTool {
2654            constructor(toolSystem) {
2655                super(toolSystem);
2656                this.textInput = null;
2657            }
2658
2659            attachEvents() {
2660                const canvas = this.layerSystem.getActiveLayer().canvas;
2661
2662                canvas.addEventListener('click', this.handleClick);
2663            }
2664
2665            detachEvents() {
2666                const canvas = this.layerSystem.getActiveLayer().canvas;
2667
2668                canvas.removeEventListener('click', this.handleClick);
2669
2670                // Remove any active text input
2671                this.removeTextInput();
2672            }
2673
```

```
2674            handleClick = (e) => {
2675                const coords = this.layerSystem.screenToCanvas(e.clientX, e.clientY);
2676
2677                // Show text input at click position
2678                this.showTextInput(coords.x, coords.y);
2679            }
2680
2681            showTextInput(x, y) {
2682                // Remove any existing text input
2683                this.removeTextInput();
2684
2685                // Create text input element
2686                this.textInput = document.createElement('div');
2687                this.textInput.style.position = 'absolute';
2688                this.textInput.style.zIndex = '100';
2689
2690                // Position relative to canvas
2691                const canvasRect = this.layerSystem.container.getBoundingClientRect();
2692                this.textInput.style.left = (canvasRect.left + x) + 'px';
2693                this.textInput.style.top = (canvasRect.top + y) + 'px';
2694
2695                // Style the input
2696                this.textInput.style.backgroundColor = 'white';
2697                this.textInput.style.border = '1px solid #ccc';
2698                this.textInput.style.padding = '5px';
2699                this.textInput.style.borderRadius = '3px';
2700                this.textInput.style.boxShadow = '0 2px 5px rgba(0,0,0,0.1)';
2701
2702                // Create the actual input
2703                const input = document.createElement('input');
2704                input.type = 'text';
2705                input.placeholder = 'Enter text...';
2706                input.style.width = '200px';
2707                input.style.padding = '5px';
2708                input.style.border = '1px solid #ddd';
2709                input.style.borderRadius = '3px';
2710
2711                // Create font size selector
2712                const sizeSelect = document.createElement('select');
2713                [8, 10, 12, 14, 16, 18, 20, 24, 36].forEach(size => {
2714                    const option = document.createElement('option');
2715                    option.value = size;
2716                    option.textContent = size + 'px';
2717                    if (size === this.settings.fontSize) {
2718                        option.selected = true;
2719                    }
2720                    sizeSelect.appendChild(option);
2721                });
2722
2723                sizeSelect.addEventListener('change', (e) => {
```

```
                    this.settings.fontSize = parseInt(e.target.value);
                });

                // 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
```

```
2872                 this.showFileDialog();
2873             }
2874 
2875             showFileDialog() {
2876                 // Create hidden file input if it doesn't exist
2877                 if (!this.fileInput) {
2878                     this.fileInput = document.createElement('input');
2879                     this.fileInput.type = 'file';
2880                     this.fileInput.accept = 'image/*';
2881                     this.fileInput.style.display = 'none';
2882                     document.body.appendChild(this.fileInput);
2883 
2884                     this.fileInput.addEventListener('change', (e) => {
2885                         if (e.target.files && e.target.files[0]) {
2886                             this.handleFileSelect(e.target.files[0]);
2887                         }
2888                     });
2889                 }
2890 
2891                 // Trigger file dialog
2892                 this.fileInput.click();
2893             }
2894 
2895             handleFileSelect(file) {
2896                 // Read the file and create a data URL
2897                 const reader = new FileReader();
2898 
2899                 reader.onload = (e) => {
2900                     const imageUrl = e.target.result;
2901 
2902                     // Show image placement UI
2903                     this.showImagePlacementUI(imageUrl);
2904                 };
2905 
2906                 reader.readAsDataURL(file);
2907             }
2908 
2909             showImagePlacementUI(imageUrl) {
2910                 // Create a preview image to get dimensions
2911                 const img = new Image();
2912 
2913                 img.onload = () => {
2914                     // Calculate dimensions (max size 300px width/height while maintaining
    aspect ratio)
2915                     let width = img.width;
2916                     let height = img.height;
2917 
2918                     const maxSize = 300;
2919                     if (width > maxSize || height > maxSize) {
2920                         if (width > height) {
```

```
                        height = (height / width) * maxSize;
                        width = maxSize;
                    } else {
                        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;

                    widthInput.addEventListener('input', () => {
                        const newWidth = parseInt(widthInput.value);
                        if (!isNaN(newWidth)) {
                            heightInput.value = Math.round(newWidth / aspectRatio);
                        }
                    });

                    heightInput.addEventListener('input', () => {
                        const newHeight = parseInt(heightInput.value);
                        if (!isNaN(newHeight)) {
                            widthInput.value = Math.round(newHeight * aspectRatio);
                        }
                    });

                    sizeControls.appendChild(widthLabel);
                    sizeControls.appendChild(widthInput);
                    sizeControls.appendChild(heightLabel);
                    sizeControls.appendChild(heightInput);

                    // Buttons container
                    const buttons = document.createElement('div');
                    buttons.style.display = 'flex';
                    buttons.style.justifyContent = 'space-between';
                    buttons.style.marginTop = '15px';

                    // Place button
                    const placeButton = document.createElement('button');
                    placeButton.textContent = 'Place Image';
                    placeButton.style.padding = '8px 15px';
                    placeButton.style.backgroundColor = '#4CAF50';
                    placeButton.style.color = 'white';
                    placeButton.style.border = 'none';
                    placeButton.style.borderRadius = '4px';
                    placeButton.style.cursor = 'pointer';

                    placeButton.addEventListener('click', () => {
                        // Get center of view as placement position
                        const containerRect =
this.layerSystem.container.getBoundingClientRect();
                        const x = containerRect.width / 2;
                        const y = containerRect.height / 2;

                        // Get dimensions from inputs
                        const finalWidth = parseInt(widthInput.value);
                        const finalHeight = parseInt(heightInput.value);
```

```
                        // Create and add image object
                        const imageObj = new ImageObject({
                            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) {
```

```
3070                super(toolSystem);
3071                this.symbolSelector = null;
3072            }
3073
3074            attachEvents() {
3075                const canvas = this.layerSystem.getActiveLayer().canvas;
3076
3077                canvas.addEventListener('click', this.handleClick);
3078            }
3079
3080            detachEvents() {
3081                const canvas = this.layerSystem.getActiveLayer().canvas;
3082
3083                canvas.removeEventListener('click', this.handleClick);
3084
3085                // Remove any active symbol selector
3086                this.removeSymbolSelector();
3087            }
3088
3089            handleClick = (e) => {
3090                const coords = this.layerSystem.screenToCanvas(e.clientX, e.clientY);
3091
3092                // Show symbol selector
3093                this.showSymbolSelector(coords.x, coords.y);
3094            }
3095
3096            showSymbolSelector(x, y) {
3097                // Remove existing selector if any
3098                this.removeSymbolSelector();
3099
3100                // Create symbol selector
3101                this.symbolSelector = document.createElement('div');
3102                this.symbolSelector.style.position = 'absolute';
3103                this.symbolSelector.style.zIndex = '100';
3104
3105                // Position relative to canvas
3106                const canvasRect = this.layerSystem.container.getBoundingClientRect();
3107                this.symbolSelector.style.left = (canvasRect.left + x + 10) + 'px';
3108                this.symbolSelector.style.top = (canvasRect.top + y + 10) + 'px';
3109
3110                // Style
3111                this.symbolSelector.style.backgroundColor = 'white';
3112                this.symbolSelector.style.border = '1px solid #ccc';
3113                this.symbolSelector.style.padding = '10px';
3114                this.symbolSelector.style.borderRadius = '5px';
3115                this.symbolSelector.style.boxShadow = '0 2px 10px rgba(0,0,0,0.1)';
3116
3117                // Add title
3118                const title = document.createElement('h4');
3119                title.textContent = 'Select Animation Symbol';
```

```
3120                title.style.margin = '0 0 10px 0';
3121                this.symbolSelector.appendChild(title);
3122
3123                // Symbol grid
3124                const symbolGrid = document.createElement('div');
3125                symbolGrid.style.display = 'grid';
3126                symbolGrid.style.gridTemplateColumns = 'repeat(3, 1fr)';
3127                symbolGrid.style.gap = '10px';
3128                symbolGrid.style.marginBottom = '10px';
3129
3130                // Available symbols
3131                const symbols = [
3132                    { type: 'anticipation', name: 'Anticipation' },
3133                    { type: 'impact', name: 'Impact' },
3134                    { type: 'keyframe', name: 'Keyframe' },
3135                    { type: 'inbetween', name: 'Inbetween' },
3136                    { type: 'hold', name: 'Hold' },
3137                    { type: 'default', name: 'Default' }
3138                ];
3139
3140                // Create symbol preview for each type
3141                symbols.forEach(symbol => {
3142                    const symbolItem = document.createElement('div');
3143                    symbolItem.style.display = 'flex';
3144                    symbolItem.style.flexDirection = 'column';
3145                    symbolItem.style.alignItems = 'center';
3146                    symbolItem.style.cursor = 'pointer';
3147                    symbolItem.style.padding = '5px';
3148                    symbolItem.style.border = '1px solid #ddd';
3149                    symbolItem.style.borderRadius = '3px';
3150
3151                    // Create canvas for symbol preview
3152                    const canvas = document.createElement('canvas');
3153                    canvas.width = 50;
3154                    canvas.height = 50;
3155                    const ctx = canvas.getContext('2d');
3156
3157                    // Draw symbol preview
3158                    ctx.save();
3159                    ctx.strokeStyle = this.settings.color;
3160                    ctx.fillStyle = this.settings.color;
3161                    ctx.lineWidth = 2;
3162
3163                    // Draw centered preview
3164                    const symbolObj = new SymbolObject({
3165                        x: 25,
3166                        y: 25,
3167                        symbolType: symbol.type,
3168                        color: this.settings.color,
3169                        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);
                scaleContainer.appendChild(scaleInput);
                scaleContainer.appendChild(scaleValue);

                this.symbolSelector.appendChild(scaleContainer);

                // Add cancel button
                const cancelButton = document.createElement('button');
                cancelButton.textContent = 'Cancel';
                cancelButton.style.width = '100%';
                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.removeSymbolSelector();
                });

                this.symbolSelector.appendChild(cancelButton);

                // Add to document
                document.body.appendChild(this.symbolSelector);
            }

            removeSymbolSelector() {
                if (this.symbolSelector && this.symbolSelector.parentNode) {
                    this.symbolSelector.parentNode.removeChild(this.symbolSelector);
                    this.symbolSelector = null;
                }
            }

            addSymbol(x, y, symbolType) {
                // Create symbol object
                const symbolObj = new SymbolObject({
                    x: x,
                    y: y,
                    symbolType: symbolType,
                    color: this.settings.color,
                    lineWidth: this.settings.lineWidth,
                    scale: this.settings.symbolScale
                });

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

```
3270            }
3271        }
3272
3273        // Frame-spanning line tool (for connecting cells across frames)
3274        class FrameSpanningLineTool extends DrawingTool {
3275            constructor(toolSystem) {
3276                super(toolSystem);
3277                this.startCell = null;
3278                this.currentLine = null;
3279                this.drawing = false;
3280            }
3281
3282            attachEvents() {
3283                // This tool works with the table cells, not the canvas
3284                this.setupCellEvents();
3285            }
3286
3287            detachEvents() {
3288                this.removeCellEvents();
3289
3290                // Finish any in-progress drawing
3291                this.finishDrawing();
3292            }
3293
3294            setupCellEvents() {
3295                // Find all table cells and add mousedown handler
3296                const cells = document.querySelectorAll('#xsheet-table td');
3297                cells.forEach(cell => {
3298                    cell.addEventListener('mousedown', this.handleCellMouseDown);
3299                    cell.addEventListener('mouseup', this.handleCellMouseUp);
3300
3301                    // Add hover effect
3302                    cell.addEventListener('mouseover', this.handleCellHover);
3303                    cell.addEventListener('mouseout', this.handleCellOut);
3304
3305                    // Store original background for hover effect
3306                    if (!cell.dataset.originalBg) {
3307                        cell.dataset.originalBg = cell.style.backgroundColor || '';
3308                    }
3309                });
3310            }
3311
3312            removeCellEvents() {
3313                const cells = document.querySelectorAll('#xsheet-table td');
3314                cells.forEach(cell => {
3315                    cell.removeEventListener('mousedown', this.handleCellMouseDown);
3316                    cell.removeEventListener('mouseup', this.handleCellMouseUp);
3317                    cell.removeEventListener('mouseover', this.handleCellHover);
3318                    cell.removeEventListener('mouseout', this.handleCellOut);
3319
```

```
3320                    // Restore original background
3321                    if (cell.dataset.originalBg) {
3322                        cell.style.backgroundColor = cell.dataset.originalBg;
3323                    }
3324                });
3325            }
3326
3327            handleCellHover = (e) => {
3328                const cell = e.currentTarget;
3329                cell.style.backgroundColor = 'rgba(0, 123, 255, 0.2)';
3330            }
3331
3332            handleCellOut = (e) => {
3333                const cell = e.currentTarget;
3334                if (!this.drawing || cell !== this.startCell) {
3335                    cell.style.backgroundColor = cell.dataset.originalBg || '';
3336                }
3337            }
3338
3339            handleCellMouseDown = (e) => {
3340                const cell = e.currentTarget;
3341
3342                // Find the row and column index
3343                const row = cell.parentElement;
3344                const frameNumber = parseInt(row.getAttribute('data-frame') ||
    row.className.replace('frame-', ''));
3345                const columnIndex = Array.from(row.children).indexOf(cell) + 1;
3346
3347                if (isNaN(frameNumber) || frameNumber <= 0) return;
3348
3349                // Store as start cell
3350                this.startCell = cell;
3351                this.drawing = true;
3352
3353                // Highlight the cell
3354                cell.style.backgroundColor = 'rgba(0, 123, 255, 0.4)';
3355
3356                // Create temporary frame-spanning line object
3357                this.currentLine = new FrameSpanningLineObject({
3358                    startFrame: frameNumber,
3359                    startColumn: columnIndex,
3360                    endFrame: frameNumber,
3361                    endColumn: columnIndex,
3362                    color: this.settings.color,
3363                    lineWidth: this.settings.lineWidth,
3364                    arrowEnd: true
3365                });
3366
3367                // Add to layer
3368                this.layerSystem.addObject(this.currentLine);
```

```
3369            }
3370
3371            handleCellMouseUp = (e) => {
3372                if (!this.drawing || !this.currentLine) return;
3373
3374                const cell = e.currentTarget;
3375
3376                // Skip if this is the same cell
3377                if (cell === this.startCell) {
3378                    this.finishDrawing();
3379                    return;
3380                }
3381
3382                // Find the row and column index
3383                const row = cell.parentElement;
3384                const frameNumber = parseInt(row.getAttribute('data-frame') ||
    row.className.replace('frame-', ''));
3385                const columnIndex = Array.from(row.children).indexOf(cell) + 1;
3386
3387                if (isNaN(frameNumber) || frameNumber <= 0) {
3388                    this.finishDrawing();
3389                    return;
3390                }
3391
3392                // Update the end point of the line
3393                this.currentLine.endFrame = frameNumber;
3394                this.currentLine.endColumn = columnIndex;
3395
3396                // Redraw
3397                this.layerSystem.redrawLayer(this.layerSystem.activeLayerIndex);
3398
3399                // Finish drawing
3400                this.finishDrawing();
3401            }
3402
3403            finishDrawing() {
3404                if (this.drawing) {
3405                    // Reset flags
3406                    this.drawing = false;
3407
3408                    // If the line is too short (same start and end), remove it
3409                    if (this.currentLine &&
3410                        this.currentLine.startFrame === this.currentLine.endFrame &&
3411                        this.currentLine.startColumn === this.currentLine.endColumn) {
3412                        this.layerSystem.removeObject(this.currentLine);
3413                    }
3414
3415                    // Clear current line reference
3416                    this.currentLine = null;
3417
```

```
                    // Reset cell highlights
                    if (this.startCell) {
                        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(() => {
```

```
3467                // Get the X-Sheet table element
3468                const xsheetTable = document.getElementById('xsheet-table');
3469                if (!xsheetTable) {
3470                    console.error('X-Sheet table not found');
3471                    return;
3472                }
3473
3474                // Create drawing systems
3475                const drawingLayerSystem = new DrawingLayerSystem(xsheetTable);
3476                const drawingToolSystem = new DrawingToolSystem(drawingLayerSystem);
3477
3478                // Store global reference
3479                window.xsheetDrawing = {
3480                    layerSystem: drawingLayerSystem,
3481                    toolSystem: drawingToolSystem
3482                };
3483
3484                // Add to status message
3485                const statusElement = document.getElementById('status-message');
3486                if (statusElement) {
3487                    statusElement.textContent = 'Drawing tools initialized';
3488                }
3489
3490                // Integrate with X-Sheet save/load system
3491                integrateWithXSheetSaveLoad();
3492
3493                // Integrate with X-Sheet PDF and printing
3494                integrateWithXSheetExport();
3495
3496                // Custom event for when drawing objects change
3497                document.addEventListener('xsheet-redraw', function() {
3498                    drawingLayerSystem.redrawAll();
3499                });
3500
3501                console.log('Drawing tools initialized successfully');
3502            }, 500);
3503        }
3504
3505        // Integrate drawing data with X-Sheet save/load system
3506        function integrateWithXSheetSaveLoad() {
3507            // Store original functions
3508            const originalCollectData = window.collectData;
3509            const originalRestoreData = window.restoreData;
3510
3511            // Override collectData to include drawings
3512            window.collectData = function() {
3513                // Call original function to get base data
3514                const data = originalCollectData ? originalCollectData() : {};
3515
3516                // Add drawing data if drawing system is initialized
```

```
3517                 if (window.xsheetDrawing && window.xsheetDrawing.layerSystem) {
3518                     data.drawingLayers = [];
3519 
3520                     // Collect objects from each layer
3521                     window.xsheetDrawing.layerSystem.layers.forEach(layer => {
3522                         const layerData = {
3523                             name: layer.name,
3524                             visible: layer.visible,
3525                             objects: layer.objects.map(obj => obj.toJSON())
3526                         };
3527 
3528                         data.drawingLayers.push(layerData);
3529                     });
3530                 }
3531 
3532                 return data;
3533             };
3534 
3535             // Override restoreData to handle drawings
3536             window.restoreData = function(data) {
3537                 // Call original function to restore base data
3538                 if (originalRestoreData) {
3539                     originalRestoreData(data);
3540                 }
3541 
3542                 // Restore drawing data if available
3543                 if (data.drawingLayers && window.xsheetDrawing &&
    window.xsheetDrawing.layerSystem) {
3544                     const layerSystem = window.xsheetDrawing.layerSystem;
3545 
3546                     // Clear existing layers
3547                     layerSystem.clearAllLayers();
3548 
3549                     // Restore each layer
3550                     data.drawingLayers.forEach((layerData, index) => {
3551                         // Create layer if needed
3552                         if (index >= layerSystem.layers.length) {
3553                             layerSystem.addLayer(layerData.name);
3554                         } else {
3555                             layerSystem.layers[index].name = layerData.name;
3556                             layerSystem.layers[index].visible = layerData.visible;
3557                         }
3558 
3559                         // Restore objects
3560                         layerData.objects.forEach(objData => {
3561                             const newObj = DrawingObjectFactory.createFromJSON(objData);
3562                             if (newObj) {
3563                                 layerSystem.layers[index].objects.push(newObj);
3564                             }
3565                         });
```

```
                    });

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

        // Integrate drawing layers with PDF export and printing
        function integrateWithXSheetExport() {
            // Store original functions
            const originalExportToPDF = window.exportToPDF;
            const originalPrintSheet = window.printSheet;

            // Override PDF export to include drawings
            window.exportToPDF = function () {
                if (!window.xsheetDrawing || !window.xsheetDrawing.layerSystem) {
                    return originalExportToPDF ? originalExportToPDF() : null;
                }

                // Save current state
                const layerSystem = window.xsheetDrawing.layerSystem;
                const layerContainer = layerSystem.container;

                // 1) Save original CSS & DOM properties
                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;

                // 2) Make it visible for capture
                layerContainer.style.display = 'block';

                // 3) Compute offsets - FIXED POSITIONING LOGIC
                const printableArea = document.getElementById('printable-area');
                const printableRect = printableArea.getBoundingClientRect();
                const tableRect = document.getElementById('xsheet-
table').getBoundingClientRect();

                // 4) Reparent into the printable area
                printableArea.appendChild(layerContainer);

                // 5) Force absolute positioning & explicit size - FIXED POSITIONING LOGIC
                layerContainer.style.position = 'absolute';
                layerContainer.style.top = `${tableRect.top - printableRect.top}px`;
                layerContainer.style.left = `${tableRect.left - printableRect.left}px`;
```

```
3615                layerContainer.style.width = `${tableRect.width}px`;
3616                layerContainer.style.height = `${tableRect.height}px`;
3617                layerContainer.style.zIndex = '1000';
3618
3619                // Force a redraw of all drawing objects
3620                layerSystem.redrawAll();
3621
3622                // 6) Wait a brief moment to ensure drawings are rendered
3623                setTimeout(() => {
3624                    // 7) Generate the PDF
3625                    const result = originalExportToPDF ? originalExportToPDF() : null;
3626
3627                    // 8) Restore everything
3628                    layerContainer.style.display = originalDisplay;
3629                    layerContainer.style.position = originalPos;
3630                    layerContainer.style.top = originalTop;
3631                    layerContainer.style.left = originalLeft;
3632                    layerContainer.style.zIndex = originalZIndex;
3633                    layerContainer.style.width = originalWidth;
3634                    layerContainer.style.height = originalHeight;
3635                    originalParent.appendChild(layerContainer);
3636                }, 50);
3637
3638                return true; // Will be async due to setTimeout
3639            };
3640
3641            // Override print to include drawings
3642            window.printSheet = function () {
3643                // 1) Fallback if no drawing system
3644                if (!window.xsheetDrawing || !window.xsheetDrawing.layerSystem) {
3645                    return originalPrintSheet ? originalPrintSheet() : null;
3646                }
3647
3648                // 2) Grab the container and save its original state
3649                const layerSystem = window.xsheetDrawing.layerSystem;
3650                const layerContainer = layerSystem.container;
3651                const originalDisplay = layerContainer.style.display;
3652                const originalPos = layerContainer.style.position;
3653                const originalTop = layerContainer.style.top;
3654                const originalLeft = layerContainer.style.left;
3655                const originalZIndex = layerContainer.style.zIndex;
3656                const originalWidth = layerContainer.style.width;
3657                const originalHeight = layerContainer.style.height;
3658                const originalParent = layerContainer.parentNode;
3659
3660                // 3) Make it visible for print capture
3661                layerContainer.style.display = 'block';
3662
3663                // 4) Compute offsets relative to the sheet - FIXED POSITIONING LOGIC
3664                const printableArea = document.getElementById('printable-area');
```

```
3665                const printableRect = printableArea.getBoundingClientRect();
3666                const tableRect = document.getElementById('xsheet-
table').getBoundingClientRect();
3667
3668                // 5) Reparent into the printable-area element
3669                printableArea.appendChild(layerContainer);
3670
3671                // 6) Force absolute positioning & lock size - FIXED POSITIONING LOGIC
3672                layerContainer.style.position = 'absolute';
3673                layerContainer.style.top = `${tableRect.top - printableRect.top}px`;
3674                layerContainer.style.left = `${tableRect.left - printableRect.left}px`;
3675                layerContainer.style.width = `${tableRect.width}px`;
3676                layerContainer.style.height = `${tableRect.height}px`;
3677                layerContainer.style.zIndex = '1000';
3678
3679                // 7) Force a redraw of all drawing objects
3680                layerSystem.redrawAll();
3681
3682                // 8) Add a print class to the container for print-specific CSS
3683                layerContainer.classList.add('printing');
3684
3685                // 9) Call the original print with a slight delay to ensure drawing render
3686                setTimeout(() => {
3687                    const result = originalPrintSheet ? originalPrintSheet() :
window.print();
3688
3689                    // 10) Restore everything back
3690                    layerContainer.style.display = originalDisplay;
3691                    layerContainer.style.position = originalPos;
3692                    layerContainer.style.top = originalTop;
3693                    layerContainer.style.left = originalLeft;
3694                    layerContainer.style.zIndex = originalZIndex;
3695                    layerContainer.style.width = originalWidth;
3696                    layerContainer.style.height = originalHeight;
3697                    layerContainer.classList.remove('printing');
3698                    originalParent.appendChild(layerContainer);
3699                }, 50);
3700
3701                return true; // Will be async due to setTimeout
3702            };
3703        }
3704
3705
3706        // Setup event listeners for table updates
3707        function setupXSheetUpdateHandling() {
3708            // Hook into existing update functions
3709            if (typeof window.generateTable === 'function') {
3710                const originalGenerateTable = window.generateTable;
3711                window.generateTable = function() {
3712                    // Call original function
```

```
3713                    const result = originalGenerateTable.apply(this, arguments);
3714
3715                    // Fire update event
3716                    document.dispatchEvent(new Event('xsheet-updated'));
3717
3718                    return result;
3719                };
3720            }
3721
3722            if (typeof window.addEightRows === 'function') {
3723                const originalAddEightRows = window.addEightRows;
3724                window.addEightRows = function() {
3725                    // Call original function
3726                    const result = originalAddEightRows.apply(this, arguments);
3727
3728                    // Fire update event
3729                    document.dispatchEvent(new Event('xsheet-updated'));
3730
3731                    return result;
3732                };
3733            }
3734
3735            if (typeof window.updateTemplate === 'function') {
3736                const originalUpdateTemplate = window.updateTemplate;
3737                window.updateTemplate = function() {
3738                    // Call original function
3739                    const result = originalUpdateTemplate.apply(this, arguments);
3740
3741                    // Fire update event
3742                    document.dispatchEvent(new Event('xsheet-updated'));
3743
3744                    return result;
3745                };
3746            }
3747        }
3748
3749        // Original X-Sheet code
3750        // Wait for page load and libraries to initialize
3751        document.addEventListener('DOMContentLoaded', function () {
3752            // Global variables
3753            let frameCount = 96;
3754            let projectName = 'Animation_XSheet_' + new Date().toISOString().split('T')[0];
3755            let modified = false;
3756            let currentTemplate = 'large';
3757
3758            // Audio variables
3759            let audioContext = null;
3760            let audioBuffer = null;
3761            let audioSource = null;
3762            let audioData = null;
```

```
3763            let isPlaying = false;
3764            let startTime = 0;
3765            let startOffset = 0;
3766            let audioFileName = '';
3767            let waveformData = [];
3768            let waveformCanvases = [];
3769            let phonetics = [];
3770            let frameDuration = 1 / 24; // 24fps
3771            let currentFrame = 0;
3772            let phoneticEditPosition = null;
3773
3774            // Initialize UI elements
3775            const templateSelector = document.getElementById('template-selector');
3776            const frameCountInput = document.getElementById('frame-count');
3777            const saveButton = document.getElementById('save-button');
3778            const loadButton = document.getElementById('load-button');
3779            const pdfButton = document.getElementById('pdf-button');
3780            const printButton = document.getElementById('print-button');
3781            const addRowsButton = document.getElementById('add-rows-button');
3782            const clearButton = document.getElementById('clear-button');
3783            const tableBody = document.getElementById('xsheet-body');
3784            const statusMessage = document.getElementById('status-message');
3785            const audioButton = document.getElementById('audio-button');
3786            const audioUpload = document.getElementById('audio-upload');
3787            const playAudioButton = document.getElementById('play-audio');
3788            const stopAudioButton = document.getElementById('stop-audio');
3789            const audioScrubber = document.getElementById('audio-scrubber');
3790            const audioInfo = document.getElementById('audio-info');
3791            const addPhoneticButton = document.getElementById('add-phonetic');
3792            const phoneticInput = document.getElementById('phonetic-input');
3793            const phoneticText = document.getElementById('phonetic-text');
3794            const savePhoneticButton = document.getElementById('save-phonetic');
3795            const cancelPhoneticButton = document.getElementById('cancel-phonetic');
3796
3797            // Date field - set today by default
3798            const dateField = document.getElementById('project-date');
3799            dateField.valueAsDate = new Date();
3800
3801            // Initialize the table
3802            generateTable(frameCount);
3803
3804            // Initialize the drawing system
3805            initDrawingSystem();
3806            setupXSheetUpdateHandling();
3807
3808            // Event Listeners
3809            templateSelector.addEventListener('change', function () {
3810                currentTemplate = this.value;
3811                updateTemplate();
3812            });
```

```
3813
3814            frameCountInput.addEventListener('change', function () {
3815                frameCount = parseInt(this.value);
3816                if (frameCount < 8) frameCount = 8;
3817                generateTable(frameCount);
3818                updateStatusMessage('Frame count updated to ' + frameCount);
3819
3820                // Re-render waveform if audio is loaded
3821                if (audioBuffer) {
3822                    renderWaveform();
3823                }
3824            });
3825
3826            saveButton.addEventListener('click', saveProject);
3827            loadButton.addEventListener('click', loadProject);
3828            pdfButton.addEventListener('click', exportToPDF);
3829            printButton.addEventListener('click', printSheet);
3830            addRowsButton.addEventListener('click', addEightRows);
3831            clearButton.addEventListener('click', clearSheet);
3832
3833            // Audio related event listeners
3834            audioButton.addEventListener('click', function () {
3835                audioUpload.click();
3836            });
3837
3838            audioUpload.addEventListener('change', function (e) {
3839                if (e.target.files.length > 0) {
3840                    const file = e.target.files[0];
3841                    audioFileName = file.name;
3842                    loadAudioFile(file);
3843                }
3844            });
3845
3846            playAudioButton.addEventListener('click', togglePlayAudio);
3847            stopAudioButton.addEventListener('click', stopAudio);
3848            audioScrubber.addEventListener('input', scrubAudio);
3849
3850            addPhoneticButton.addEventListener('click', function () {
3851                if (audioBuffer) {
3852                    showPhoneticInput(null);
3853                } else {
3854                    updateStatusMessage('Please load audio first');
3855                }
3856            });
3857
3858            savePhoneticButton.addEventListener('click', savePhoneticMarker);
3859            cancelPhoneticButton.addEventListener('click', function () {
3860                phoneticInput.style.display = 'none';
3861            });
3862
```

```
            // Monitor for changes to set modified flag
            document.addEventListener('input', function (e) {
                if (!e.target.matches('input, [contenteditable="true"]')) return;
                modified = true;

                if (e.target.matches('[contenteditable="true"]')) {
                    // Add the modified class when content is added
                    e.target.classList.add('modified');

                    // Remove the modified class when the cell is empty
                    if (e.target.textContent.trim() === '') {
                        e.target.classList.remove('modified');
                    }
                }

                updateStatusMessage('Changes detected - not saved');
            });

            // Variables for cell selection
            let selectedCells = [];
            let isSelecting = false;
            let selectionStart = null;
            let hasMovedDuringSelection = false;

            // Set up multi-cell selection functionality
            function setupCellSelection() {
                const editableCells = document.querySelectorAll('[contenteditable="true"]');

                // Clear selected cells when clicking outside
                document.addEventListener('click', function (e) {
                    // If this was part of a drag operation, skip the click handler
                    if (hasMovedDuringSelection) {
                        // We removed this line to fix the selection issue
                        // hasMovedDuringSelection = false;
                        return;
                    }

                    // Get the clicked element and check if it's inside an editable cell or a selected cell
                    const clickedCell = e.target.closest('[contenteditable="true"]');
                    const clickedInSelection = e.target.closest('.selected-cell');

                    // If not clicking in a cell or selection and not using modifier keys
                    if (!clickedCell && !clickedInSelection && !e.ctrlKey && !e.metaKey) {
                        clearCellSelection();
                    }
                });

                // Handle clicks on the table for better cell focusing
```

```
3911                document.getElementById('xsheet-table').addEventListener('click', function
    (e) {
3912                    // Find the closest editable cell from the click point
3913                    const targetCell = e.target.closest('[contenteditable="true"]');
3914
3915                    // If we found a cell and we're not in multi-select mode
3916                    if (targetCell && !hasMovedDuringSelection && !e.ctrlKey && !e.metaKey)
    {
3917                        // Focus the cell and position cursor at the end
3918                        if (selectedCells.length === 1 && selectedCells[0] === targetCell) {
3919                            // If already selected, just ensure focus
3920                            targetCell.focus();
3921                            placeCaretAtEnd(targetCell);
3922                        }
3923                    }
3924                });
3925
3926                // Helper function to place caret at the end of content
3927                function placeCaretAtEnd(el) {
3928                    if (document.createRange) {
3929                        const range = document.createRange();
3930                        range.selectNodeContents(el);
3931                        range.collapse(false); // false means collapse to end
3932                        const selection = window.getSelection();
3933                        selection.removeAllRanges();
3934                        selection.addRange(range);
3935                    }
3936                }
3937
3938                // Handle mousedown for selection start
3939                editableCells.forEach(cell => {
3940                    // Mousedown - start potential selection
3941                    cell.addEventListener('mousedown', function (e) {
3942                        // Only handle left mouse button
3943                        if (e.button !== 0) return;
3944
3945                        // If using modifier keys for multi-select
3946                        if (e.ctrlKey || e.metaKey) {
3947                            toggleCellSelection(cell);
3948                            e.preventDefault(); // Prevent text cursor
3949                            return;
3950                        }
3951
3952                        // Clear previous selection unless clicking on already selected cell
3953                        if (!cell.classList.contains('selected-cell')) {
3954                            clearCellSelection();
3955                        }
3956
3957                        // Store starting cell and reset movement flag
3958                        selectionStart = cell;
```

```
3959                        hasMovedDuringSelection = false;
3960                        isSelecting = true;
3961
3962                        // Initially add this cell to selection
3963                        if (!cell.classList.contains('selected-cell')) {
3964                            toggleCellSelection(cell, true);
3965                        }
3966
3967                        // If it's a single click (not start of drag), focus the cell
3968                        cell.focus();
3969
3970                        // Don't prevent default here to allow normal focus behavior
3971                    });
3972
3973                    // Additional keyboard events for selected cells
3974                    cell.addEventListener('keydown', function (e) {
3975                        // If Delete or Backspace key and we have selected cells
3976                        if ((e.key === 'Delete' || e.key === 'Backspace') &&
    selectedCells.length > 1) {
3977                            e.preventDefault();
3978                            clearSelectedCellsContent();
3979                        }
3980
3981                        // Copy with Ctrl+C or Cmd+C
3982                        if ((e.ctrlKey || e.metaKey) && e.key === 'c' &&
    selectedCells.length > 0) {
3983                            e.preventDefault();
3984                            copySelectedCells();
3985                        }
3986
3987                        // Select all cells in row with Ctrl+A or Cmd+A
3988                        if ((e.ctrlKey || e.metaKey) && e.key === 'a') {
3989                            e.preventDefault();
3990                            selectRowCells(cell.parentElement);
3991                        }
3992                    });
3993                });
3994
3995                // Global mousemove handler for extending selection
3996                document.addEventListener('mousemove', function (e) {
3997                    if (!isSelecting) return;
3998
3999                    // Set the flag that we've moved during this selection
4000                    hasMovedDuringSelection = true;
4001
4002                    // Find the element under the cursor
4003                    const elementUnderCursor = document.elementFromPoint(e.clientX,
    e.clientY);
4004                    if (!elementUnderCursor) return;
4005
4006                    // Find the closest editable cell
```

```
4007                    const targetCell =
elementUnderCursor.closest('[contenteditable="true"]');
4008                    if (targetCell) {
4009                        // Add to selection
4010                        toggleCellSelection(targetCell, true);
4011
4012                        // Prevent text selection during drag
4013                        e.preventDefault();
4014                        if (window.getSelection) {
4015                            window.getSelection().removeAllRanges();
4016                        }
4017                    }
4018                });
4019
4020                // Global mouseup to end selection - THIS IS THE FIXED VERSION
4021                document.addEventListener('mouseup', function (e) {
4022                    // End selection mode but keep selected cells
4023                    if (isSelecting) {
4024                        isSelecting = false;
4025
4026                        // FIX: Use setTimeout to delay resetting the hasMovedDuringSelect-
ion flag
4027                        // This gives the click handler a chance to see the flag first
4028                        if (hasMovedDuringSelection) {
4029                            setTimeout(function () {
4030                                hasMovedDuringSelection = false;
4031                            }, 10);
4032                        }
4033
4034                        // If this wasn't a drag and there's only one cell selected, focus
it
4035                        if (!hasMovedDuringSelection && selectedCells.length === 1) {
4036                            const cell = selectedCells[0];
4037                            cell.focus();
4038                            placeCaretAtEnd(cell);
4039                        }
4040                    }
4041
4042                    selectionStart = null;
4043                });
4044
4045                // Listen for paste events
4046                document.addEventListener('paste', function (e) {
4047                    if (selectedCells.length > 0) {
4048                        handlePaste(e);
4049                    }
4050                });
4051            }
4052
4053            function toggleCellSelection(cell, addOnly = false) {
4054                const index = selectedCells.indexOf(cell);
```

```
4055
4056                if (index === -1) {
4057                    // Add to selection
4058                    selectedCells.push(cell);
4059                    cell.classList.add('selected-cell');
4060                } else if (!addOnly) {
4061                    // Remove from selection if not add-only mode
4062                    selectedCells.splice(index, 1);
4063                    cell.classList.remove('selected-cell');
4064                }
4065            }
4066
4067            function clearCellSelection() {
4068                selectedCells.forEach(cell => {
4069                    cell.classList.remove('selected-cell');
4070                });
4071                selectedCells = [];
4072            }
4073
4074            function clearSelectedCellsContent() {
4075                selectedCells.forEach(cell => {
4076                    cell.textContent = '';
4077                    cell.classList.remove('modified');
4078                });
4079                modified = true;
4080                updateStatusMessage('Cleared selected cells');
4081            }
4082
4083            function copySelectedCells() {
4084                if (selectedCells.length === 0) return;
4085
4086                // Create a text representation of selected cells
4087                const cellData = selectedCells.map(cell => cell.textContent ||
    '').join('\t');
4088
4089                // Copy to clipboard using Clipboard API
4090                navigator.clipboard.writeText(cellData)
4091                    .then(() => {
4092                        updateStatusMessage('Copied selected cells to clipboard');
4093                    })
4094                    .catch(err => {
4095                        updateStatusMessage('Failed to copy: ' + err);
4096                    });
4097            }
4098
4099            function handlePaste(e) {
4100                // Prevent default paste behavior
4101                e.preventDefault();
4102
4103                // Get clipboard data
```

```
4104                const clipboardData = e.clipboardData || window.clipboardData;
4105                const pastedText = clipboardData.getData('text');
4106
4107                // If we have a focused element within selection, paste there
4108                const activeElement = document.activeElement;
4109                if (activeElement && selectedCells.includes(activeElement)) {
4110                    activeElement.textContent = pastedText;
4111                    activeElement.classList.add('modified');
4112                    modified = true;
4113                } else if (selectedCells.length > 0) {
4114                    // Otherwise paste into first selected cell
4115                    selectedCells[0].textContent = pastedText;
4116                    selectedCells[0].classList.add('modified');
4117                    modified = true;
4118                }
4119
4120                if (pastedText.trim() === '') {
4121                    selectedCells.forEach(cell => {
4122                        cell.classList.remove('modified');
4123                    });
4124                }
4125
4126                updateStatusMessage('Pasted content into selected cell');
4127            }
4128
4129            function selectRowCells(row) {
4130                // Clear previous selection
4131                clearCellSelection();
4132
4133                // Select all editable cells in the row
4134                const cells = Array.from(row.cells).filter(cell => cell.contentEditable ===
    'true');
4135                cells.forEach(cell => {
4136                    toggleCellSelection(cell, true);
4137                });
4138            }
4139
4140            // Functions
4141            function generateTable(frames) {
4142                tableBody.innerHTML = '';
4143                waveformCanvases = [];
4144
4145                // Create a column container for the waveform
4146                const waveformColContainer = document.createElement('div');
4147                waveformColContainer.className = 'waveform-col-container';
4148
4149                for (let i = 1; i <= frames; i++) {
4150                    const row = document.createElement('tr');
4151                    row.className = `frame-${i}`;
4152                    row.setAttribute('data-frame', i);
```

```
                    // Create standard cell
                    function createCell(className, isEditable = true, frameNum = null) {
                        const cell = document.createElement('td');
                        cell.className = className;

                        if (isEditable) {
                            cell.contentEditable = true;
                            cell.setAttribute('data-placeholder', '');
                            cell.setAttribute('tabindex', '0');
                        } else if (frameNum !== null) {
                            cell.textContent = frameNum;
                            cell.className += ' frame-number';

                            // Color the first frame green for reference (like in the example image)
                            if (frameNum === 1) {
                                cell.style.backgroundColor = '#00cc00';
                            } else {
                                cell.style.backgroundColor = '#cccccc';
                            }
                        }

                        return cell;
                    }

                    // Create waveform cell (placeholder for the vertical waveform)
                    function createWaveformCell(frameNum) {
                        const cell = document.createElement('td');
                        cell.className = 'waveform-col';
                        cell.setAttribute('data-frame', frameNum);
                        return cell;
                    }

                    // Add all cells to the row
                    row.appendChild(createCell('action-col'));
                    row.appendChild(createCell('frame-col', false, i));
                    row.appendChild(createWaveformCell(i));
                    row.appendChild(createCell('dialogue-col'));
                    row.appendChild(createCell('sound-col'));
                    row.appendChild(createCell('technical-col'));
                    row.appendChild(createCell('extra1-col'));
                    row.appendChild(createCell('extra2-col'));
                    row.appendChild(createCell('frame-col', false, i));
                    row.appendChild(createCell('camera-col'));

                    // Add special styling for 8-frame and 24-frame intervals
                    if (i % 24 === 0) {
                        row.style.borderBottom = '4px double #000';
                        for (let cell of row.cells) {
```

```
                            cell.style.borderBottom = '4px double #000';
                            cell.style.fontWeight = 'bold';
                        }
                    } 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"]');
```

```
4251
4252                editableCells.forEach(cell => {
4253                    cell.addEventListener('keydown', function (e) {
4254                        // Tab navigation
4255                        if (e.key === 'Tab') {
4256                            e.preventDefault();
4257                            const currentRow = this.parentElement;
4258                            const currentIndex = Array.from(currentRow.cells).indexOf(this);
4259
4260                            if (e.shiftKey) {
4261                                // Shift+Tab moves backward
4262                                let prevCell = null;
4263                                if (currentIndex > 0) {
4264                                    // Find previous editable cell in same row
4265                                    for (let i = currentIndex - 1; i >= 0; i--) {
4266                                        if (currentRow.cells[i].contentEditable === 'true')
    {
4267                                            prevCell = currentRow.cells[i];
4268                                            break;
4269                                        }
4270                                    }
4271                                }
4272
4273                                if (!prevCell) {
4274                                    // Move to previous row, last cell
4275                                    const prevRow = currentRow.previousElementSibling;
4276                                    if (prevRow) {
4277                                        const cells = Array.from(prevRow.cells).filter(c =>
    c.contentEditable === 'true');
4278                                        prevCell = cells[cells.length - 1];
4279                                    }
4280                                }
4281
4282                                if (prevCell) {
4283                                    prevCell.focus();
4284                                    // Place cursor at end of content
4285                                    const range = document.createRange();
4286                                    const sel = window.getSelection();
4287                                    range.selectNodeContents(prevCell);
4288                                    range.collapse(false);
4289                                    sel.removeAllRanges();
4290                                    sel.addRange(range);
4291                                }
4292                            } else {
4293                                // Tab moves forward
4294                                let nextCell = null;
4295                                if (currentIndex < currentRow.cells.length - 1) {
4296                                    // Find next editable cell in same row
4297                                    for (let i = currentIndex + 1; i <
    currentRow.cells.length; i++) {
```

```
4298                                    if (currentRow.cells[i].contentEditable === 'true')
{
4299                                        nextCell = currentRow.cells[i];
4300                                        break;
4301                                    }
4302                                }
4303                            }
4304
4305                            if (!nextCell) {
4306                                // Move to next row, first cell
4307                                const nextRow = currentRow.nextElementSibling;
4308                                if (nextRow) {
4309                                    nextCell = Array.from(nextRow.cells).find(c =>
c.contentEditable === 'true');
4310                                }
4311                            }
4312
4313                            if (nextCell) {
4314                                nextCell.focus();
4315                            }
4316                        }
4317                    }
4318
4319                    // Enter key to move down
4320                    if (e.key === 'Enter' && !e.shiftKey) {
4321                        e.preventDefault();
4322                        const currentRow = this.parentElement;
4323                        const nextRow = currentRow.nextElementSibling;
4324                        const currentIndex = Array.from(currentRow.cells).indexOf(this);
4325
4326                        if (nextRow) {
4327                            const cells = Array.from(nextRow.cells);
4328                            const samePositionCell = cells[currentIndex];
4329                            if (samePositionCell && samePositionCell.contentEditable ===
'true') {
4330                                samePositionCell.focus();
4331                            }
4332                        }
4333                    }
4334                });
4335
4336                // Add listener to remove the modified class when cell is emptied
4337                cell.addEventListener('keyup', function (e) {
4338                    if ((e.key === 'Delete' || e.key === 'Backspace') &&
this.textContent.trim() === '') {
4339                        this.classList.remove('modified');
4340                    }
4341                });
4342            });
4343
4344            // Setup cell selection functionality
```

```
4345                setupCellSelection();
4346            }
4347
4348            // Audio functions
4349            function loadAudioFile(file) {
4350                if (!audioContext) {
4351                    try {
4352                        window.AudioContext = window.AudioContext ||
    window.webkitAudioContext;
4353                        audioContext = new AudioContext();
4354                    } catch (e) {
4355                        updateStatusMessage('Web Audio API is not supported in this
    browser');
4356                        return;
4357                    }
4358                }
4359
4360                const reader = new FileReader();
4361
4362                reader.onload = function (e) {
4363                    const audioData = e.target.result;
4364
4365                    // Update status
4366                    updateStatusMessage('Decoding audio...');
4367
4368                    // Decode the audio data
4369                    audioContext.decodeAudioData(audioData, function (buffer) {
4370                        audioBuffer = buffer;
4371
4372                        // Update audio info
4373                        const duration = buffer.duration;
4374                        const minutes = Math.floor(duration / 60);
4375                        const seconds = Math.floor(duration % 60);
4376                        const frameCount = Math.ceil(duration * 24); // Assuming 24fps
4377
4378                        audioInfo.textContent = `${audioFileName}
    (${minutes}:${seconds.toString().padStart(2, '0')}, ${frameCount} frames @ 24fps)`;
4379
4380                        // Enable audio controls
4381                        playAudioButton.disabled = false;
4382                        stopAudioButton.disabled = false;
4383                        audioScrubber.disabled = false;
4384
4385                        // Generate waveform visualization
4386                        generateWaveformData(buffer);
4387
4388                        // Update status
4389                        updateStatusMessage('Audio loaded: ' + audioFileName);
4390
4391                        // If the x-sheet has fewer frames than the audio, suggest
    increasing
```

```
                        if (frameCount > frameCountInput.value) {
                            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?`)) {
                                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]));
                }
            }
        }

        function renderWaveform() {
            if (!audioBuffer || waveformData.length === 0) return;

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

            const existingLabels = document.querySelectorAll('.phonetic-label');
            existingLabels.forEach(label => label.remove());

            const existingContainer = document.querySelector('.waveform-container');
            if (existingContainer) {
                existingContainer.remove();
            }

            // Get the table and first waveform column cell for positioning
            const table = document.getElementById('xsheet-table');
            const firstCell = document.querySelector('.waveform-col[data-frame="1"]');
            if (!firstCell) return;

            // Calculate total height needed for the waveform
            const totalRows = document.querySelectorAll('tr[data-frame]').length;
            const rowHeight = firstCell.offsetHeight;
            const totalHeight = totalRows * rowHeight;

            // Create a container for the vertical waveform that spans the entire table
            const waveformContainer = document.createElement('div');
            waveformContainer.className = 'waveform-container';

            // Create a canvas for the waveform
            const canvas = document.createElement('canvas');
            canvas.className = 'waveform-canvas';
            canvas.width = firstCell.offsetWidth;
            canvas.height = totalHeight;
            waveformContainer.appendChild(canvas);

            // Add overlay for event handling
            const overlay = document.createElement('div');
            overlay.className = 'waveform-overlay';
            overlay.style.height = totalHeight + 'px';

            // Add event listener for clicking on the waveform
            overlay.addEventListener('click', function (e) {
                if (!audioBuffer) return;

```

```
                    const rect = overlay.getBoundingClientRect();
                    const y = e.clientY - rect.top;
                    const percentage = y / rect.height;

                    // 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);
                        }
```

```
4538                    });
4539                    overlay.addEventListener('pointerup', function (e) {
4540                        if (e.pointerType === 'pen') {
4541                            isScrubbing = false;
4542                        }
4543                    });
4544
4545                    // Create a scrub audio context if needed
4546                    if (!audioContext) {
4547                        try {
4548                            window.AudioContext = window.AudioContext ||
    window.webkitAudioContext;
4549                            audioContext = new AudioContext();
4550                        } catch (e) {
4551                            console.error('Web Audio API not supported');
4552                            return;
4553                        }
4554                    }
4555
4556                    // Initial scrub to current position
4557                    updateScrubPosition(e);
4558
4559                    // Prevent text selection during drag
4560                    e.preventDefault();
4561                });
4562
4563                // Handle scrubbing movement
4564                overlay.addEventListener('mousemove', function (e) {
4565                    if (!isScrubbing || !audioBuffer) return;
4566                    updateScrubPosition(e);
4567                });
4568
4569                // Function to update position during scrubbing
4570                function updateScrubPosition(e) {
4571                    const rect = overlay.getBoundingClientRect();
4572                    const y = e.clientY - rect.top;
4573                    const percentage = Math.max(0, Math.min(1, y / rect.height));
4574
4575                    // Calculate time point in the audio
4576                    const timePoint = percentage * audioBuffer.duration;
4577                    startOffset = timePoint;
4578
4579                    // Update audio scrubber
4580                    audioScrubber.value = percentage * 100;
4581
4582                    // Play a short snippet of audio at this position
4583                    playScrubAudio(timePoint);
4584
4585                    // Update the marker
4586                    updateFrameMarker();
```

```
4587                }
4588
4589                // End scrubbing when mouse is released or leaves element
4590                overlay.addEventListener('mouseup', function () {
4591                    isScrubbing = false;
4592                });
4593
4594                overlay.addEventListener('mouseleave', function () {
4595                    isScrubbing = false;
4596                });
4597
4598                // Function to play a short snippet at the scrub position
4599                let scrubSource = null;
4600                function playScrubAudio(timePoint) {
4601                    if (scrubSource) {
4602                        try {
4603                            scrubSource.stop();
4604                        } catch (e) {
4605                            // Ignore errors when stopping already stopped source
4606                        }
4607                    }
4608
4609                    // Play a very short snippet at the current position
4610                    scrubSource = audioContext.createBufferSource();
4611                    scrubSource.buffer = audioBuffer;
4612                    scrubSource.connect(audioContext.destination);
4613
4614                    // Play just a short snippet (equivalent to 1-2 frames at 24fps)
4615                    const snippetDuration = 1 / 12; // 1/12 of a second (2 frames at 24fps)
4616                    scrubSource.start(0, timePoint, snippetDuration);
4617                }
4618
4619                // Right-click to add phonetic marker
4620                overlay.addEventListener('contextmenu', function (e) {
4621                    e.preventDefault();
4622                    if (!audioBuffer) return;
4623
4624                    const rect = overlay.getBoundingClientRect();
4625                    const y = e.clientY - rect.top;
4626                    const percentage = y / rect.height;
4627
4628                    // Calculate time point in the audio
4629                    const timePoint = percentage * audioBuffer.duration;
4630
4631                    showPhoneticInput(timePoint);
4632                    return false;
4633                });
4634
4635                waveformContainer.appendChild(overlay);
4636
```

```
4637                // Add the container to the table
4638                document.body.appendChild(waveformContainer);
4639
4640                // Position the container over the waveform column
4641                const tableRect = table.getBoundingClientRect();
4642                const cellRect = firstCell.getBoundingClientRect();
4643
4644                waveformContainer.style.left = (cellRect.left - 1) + 'px';
4645                waveformContainer.style.top = (cellRect.top) + 'px';
4646                waveformContainer.style.width = (cellRect.width) + 'px';
4647                waveformContainer.style.height = totalHeight + 'px';
4648
4649                // Draw the waveform on the canvas
4650                const ctx = canvas.getContext('2d');
4651                const width = canvas.width;
4652                const height = canvas.height;
4653
4654                // Clear and set background
4655                ctx.clearRect(0, 0, width, height);
4656                ctx.fillStyle = '#ffffff';
4657                ctx.fillRect(0, 0, width, height);
4658
4659                // Calculate scaling factors
4660                const totalDuration = audioBuffer.duration;
4661                const framesPerSecond = 24;
4662                const totalFrames = Math.ceil(totalDuration * framesPerSecond);
4663
4664                // Draw center line
4665                ctx.beginPath();
4666                ctx.strokeStyle = '#cccccc';
4667                ctx.moveTo(width / 2, 0);
4668                ctx.lineTo(width / 2, height);
4669                ctx.stroke();
4670
4671                // Draw the waveform
4672                ctx.beginPath();
4673                ctx.strokeStyle = '#000000';
4674                ctx.lineWidth = 1;
4675
4676                // Map waveform data to vertical height
4677                for (let i = 0; i < waveformData.length; i++) {
4678                    const y = (i / waveformData.length) * height;
4679                    const amplitude = waveformData[i] * (width * 0.4); // Scale amplitude to
    40% of width
4680                    const x = (width / 2) + amplitude;
4681
4682                    if (i === 0) {
4683                        ctx.moveTo(x, y);
4684                    } else {
4685                        ctx.lineTo(x, y);
```

```
                    }
                }

                // Draw mirrored waveform for visual effect
                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';

                // Add frame number
                marker.textContent = `Frame ${frame}`;
                marker.style.lineHeight = '0';
                marker.style.textAlign = 'right';
                marker.style.color = 'red';
                marker.style.fontWeight = 'bold';
                marker.style.fontSize = '8pt';

                waveformContainer.appendChild(marker);

                // Scroll to the marker if it's not visible and if we're playing
                if (isPlaying) {
                    const table = document.getElementById('xsheet-table');
                    const tableRect = table.getBoundingClientRect();
                    const markerY = tableRect.top + yPosition;

                    // Check if marker is outside visible area
                    if (markerY < window.scrollY || markerY > window.scrollY +
window.innerHeight) {
                        window.scrollTo({
                            top: markerY - (window.innerHeight / 2),
                            behavior: 'smooth'
                        });
                    }
                }

                // Update scrubber position
                if (audioBuffer) {
                    const scrubPercentage = (time / audioBuffer.duration) * 100;
                    audioScrubber.value = scrubPercentage;
                }

                // If playing, schedule the next update
                if (isPlaying) {
                    requestAnimationFrame(updateFrameMarker);
                }
            }

            function togglePlayAudio() {
                if (isPlaying) {
                    pauseAudio();
                } else {
                    playAudio();
                }
```

```
4834            }
4835
4836            function playAudio() {
4837                if (!audioBuffer) return;
4838
4839                try {
4840                    // Create a new source node
4841                    audioSource = audioContext.createBufferSource();
4842                    audioSource.buffer = audioBuffer;
4843                    audioSource.connect(audioContext.destination);
4844
4845                    // Calculate start position
4846                    startTime = audioContext.currentTime;
4847
4848                    // Start playback from the current offset
4849                    audioSource.start(0, startOffset);
4850                    isPlaying = true;
4851
4852                    // Set up animation loop
4853                    updateFrameMarker();
4854
4855                    // Update UI
4856                    playAudioButton.textContent = 'Pause';
4857
4858                    // Set up ended event
4859                    audioSource.onended = function () {
4860                        if (isPlaying) {
4861                            stopAudio();
4862                        }
4863                    };
4864                } catch (e) {
4865                    updateStatusMessage('Error playing audio: ' + e.message);
4866                }
4867            }
4868
4869            function pauseAudio() {
4870                if (!isPlaying || !audioSource) return;
4871
4872                // Stop the audio
4873                audioSource.stop();
4874
4875                // Calculate current position
4876                startOffset += audioContext.currentTime - startTime;
4877
4878                isPlaying = false;
4879
4880                // Update UI
4881                playAudioButton.textContent = 'Play';
4882            }
4883
```

```
4884            function stopAudio() {
4885                if (audioSource) {
4886                    try {
4887                        audioSource.stop();
4888                    } catch (e) {
4889                        // Ignore errors when stopping already stopped source
4890                    }
4891                }
4892
4893                isPlaying = false;
4894                startOffset = 0;
4895
4896                // Update UI
4897                playAudioButton.textContent = 'Play';
4898                audioScrubber.value = 0;
4899
4900                // Clear frame marker
4901                updateFrameMarker();
4902            }
4903
4904            function scrubAudio() {
4905                if (!audioBuffer) return;
4906
4907                // Calculate time from scrubber position
4908                const percentage = audioScrubber.value / 100;
4909                const newTime = percentage * audioBuffer.duration;
4910
4911                // If playing, restart from new position
4912                if (isPlaying) {
4913                    pauseAudio();
4914                    startOffset = newTime;
4915                    playAudio();
4916                } else {
4917                    startOffset = newTime;
4918                    updateFrameMarker();
4919                }
4920            }
4921
4922            function showPhoneticInput(time, initialText = '', editIndex = -1) {
4923                if (time === null) {
4924                    // If no time provided, use current position
4925                    time = isPlaying ?
4926                        (audioContext.currentTime - startTime + startOffset) :
4927                        startOffset;
4928                }
4929
4930                // Store position for later use
4931                phoneticEditPosition = {
4932                    time: time,
4933                    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;
```

```
5032            }
5033
5034            function restoreData(data) {
5035                if (!data) return;
5036
5037                // Update template and frame count
5038                currentTemplate = data.template || 'large';
5039                frameCount = data.frameCount || 96;
5040
5041                // Update UI to match
5042                templateSelector.value = currentTemplate;
5043                frameCountInput.value = frameCount;
5044
5045                // Restore metadata
5046                if (data.metadata) {
5047                    document.getElementById('project-number').value =
    data.metadata.projectNumber || '';
5048                    document.getElementById('project-date').value = data.metadata.date ||
    '';
5049                    document.getElementById('page-number').value = data.metadata.pageNumber
    || '';
5050                    document.getElementById('animator-name').value =
    data.metadata.animatorName || '';
5051                    document.getElementById('version-number').value =
    data.metadata.versionNumber || '';
5052                    document.getElementById('shot-number').value = data.metadata.shotNumber
    || '';
5053                }
5054
5055                // Generate the table with the right frame count
5056                generateTable(frameCount);
5057                updateTemplate();
5058
5059                // Restore audio data
5060                if (data.audio) {
5061                    audioFileName = data.audio.fileName || '';
5062                    phonetics = data.audio.phonetics || [];
5063
5064                    if (audioBuffer && phonetics.length > 0) {
5065                        renderWaveform();
5066                    }
5067                }
5068
5069                // Restore row data
5070                if (data.rows && data.rows.length > 0) {
5071                    const rows = tableBody.querySelectorAll('tr');
5072                    data.rows.forEach((rowData, index) => {
5073                        if (index < rows.length) {
5074                            rows[index].cells[0].innerText = rowData.action || '';
5075                            // Don't restore frame number cells as they're auto-generated
5076                            rows[index].cells[3].innerText = rowData.dialogue || '';
```

```
                            rows[index].cells[4].innerText = rowData.sound || '';
                            rows[index].cells[5].innerText = rowData.technical || '';
                            rows[index].cells[6].innerText = rowData.extra1 || '';
                            rows[index].cells[7].innerText = rowData.extra2 || '';
                            // 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);
                }

```

```
5126                // If no localStorage data, create file input for uploading JSON
5127                const fileInput = document.createElement('input');
5128                fileInput.type = 'file';
5129                fileInput.accept = '.json';
5130                fileInput.style.display = 'none';
5131                document.body.appendChild(fileInput);
5132
5133                fileInput.addEventListener('change', function (e) {
5134                    const file = e.target.files[0];
5135                    if (!file) return;
5136
5137                    const reader = new FileReader();
5138                    reader.onload = function (e) {
5139                        try {
5140                            const data = JSON.parse(e.target.result);
5141                            restoreData(data);
5142                        } catch (error) {
5143                            updateStatusMessage('Error parsing file: ' + error.message);
5144                        }
5145                    };
5146                    reader.readAsText(file);
5147                });
5148
5149                fileInput.click();
5150                fileInput.remove();
5151            }
5152
5153            function exportToPDF() {
5154                // Save the current state before PDF export
5155                const savedData = collectData();
5156
5157                updateStatusMessage('Preparing PDF. Please wait...');
5158
5159                // Remove controls temporarily for PDF generation
5160                const controls = document.querySelector('.controls');
5161                const audioControls = document.querySelector('#audio-controls');
5162                const statusMsg = document.querySelector('.status');
5163                const phoneticInputEl = document.querySelector('#phonetic-input');
5164
5165                controls.style.display = 'none';
5166                audioControls.style.display = 'none';
5167                statusMsg.style.display = 'none';
5168                phoneticInputEl.style.display = 'none';
5169
5170                // Clean up any previous waveform elements
5171                const previousContainers = document.querySelectorAll('.cell-waveform-
window');
5172                previousContainers.forEach(container => container.remove());
5173
5174                // Draw the waveform directly into the cells
```

```
5175                if (audioBuffer && waveformData.length > 0) {
5176                    drawWaveformInCells();
5177                }
5178
5179                // Wait a moment for DOM updates to complete
5180                setTimeout(() => {
5181                    try {
5182                        // Force a redraw of the drawing layer before capture
5183                        if (window.xsheetDrawing && window.xsheetDrawing.layerSystem) {
5184                            window.xsheetDrawing.layerSystem.redrawAll();
5185                        }
5186
5187                        // Use html2canvas to capture the printable area
5188                        html2canvas(document.getElementById('printable-area'), {
5189                            scale: 2, // Higher resolution
5190                            useCORS: true,
5191                            logging: false,
5192                            allowTaint: true,
5193                            onclone: function (clonedDoc) {
5194                                // Ensure drawing elements are visible in the cloned
    document
5195                                const clonedDrawingLayer =
    clonedDoc.querySelector('.drawing-layer-container');
5196                                if (clonedDrawingLayer) {
5197                                    clonedDrawingLayer.style.display = 'block';
5198                                    clonedDrawingLayer.style.visibility = 'visible';
5199                                    clonedDrawingLayer.style.opacity = '1';
5200                                }
5201                            }
5202                        }).then(canvas => {
5203                            const imgData = canvas.toDataURL('image/png');
5204
5205                            // Determine PDF size based on template
5206                            let pdfWidth, pdfHeight;
5207                            if (currentTemplate === 'large') {
5208                                // 11"x17"
5209                                pdfWidth = 279.4; // mm
5210                                pdfHeight = 431.8; // mm
5211                            } else {
5212                                // 8.5"x11"
5213                                pdfWidth = 215.9; // mm
5214                                pdfHeight = 279.4; // mm
5215                            }
5216
5217                            // Create PDF with jsPDF
5218                            const { jsPDF } = window.jspdf;
5219                            const pdf = new jsPDF({
5220                                orientation: 'portrait',
5221                                unit: 'mm',
5222                                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%';

                        const ctx = canvas.getContext('2d');

                        // Fill with white background
                        ctx.fillStyle = 'white';
                        ctx.fillRect(0, 0, canvas.width, canvas.height);

                        // Draw center line
                        ctx.beginPath();
                        ctx.strokeStyle = '#cccccc';
                        ctx.moveTo(canvas.width / 2, 0);
                        ctx.lineTo(canvas.width / 2, canvas.height);
                        ctx.stroke();

                        // Calculate which section of the waveform to draw
                        const startPoint = Math.floor((frameNum - 1) * pointsPerFrame);
                        const endPoint = Math.floor(frameNum * pointsPerFrame);

                        if (startPoint < waveformData.length) {
                            // Draw this portion of the waveform
                            ctx.beginPath();
                            ctx.strokeStyle = '#000000';

                            // Check if we have valid data to draw
                            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);

                    // 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);

                        updateStatusMessage('Print complete');
                    }, 500);
                }, 100);

                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%';

                        const ctx = canvas.getContext('2d');

                        // Fill with white background
                        ctx.fillStyle = 'white';
                        ctx.fillRect(0, 0, canvas.width, canvas.height);

                        // Draw center line
                        ctx.beginPath();
                        ctx.strokeStyle = '#cccccc';
                        ctx.moveTo(canvas.width / 2, 0);
                        ctx.lineTo(canvas.width / 2, canvas.height);
                        ctx.stroke();

                        // Calculate which section of the waveform to draw
                        const startPoint = Math.floor((frameNum - 1) * pointsPerFrame);
                        const endPoint = Math.floor(frameNum * pointsPerFrame);

                        if (startPoint < waveformData.length) {
```

```
                        // Draw this portion of the waveform
                        ctx.beginPath();
                        ctx.strokeStyle = '#000000';

                        // Check if we have valid data to draw
                        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';
```

```
5692
5693                    // Stop any playing audio
5694                    stopAudio();
5695
5696                    // Clear waveform visualization
5697                    waveformCanvases.forEach(canvas => {
5698                        const ctx = canvas.getContext('2d');
5699                        ctx.clearRect(0, 0, canvas.width, canvas.height);
5700                    });
5701
5702                    // Clear phonetic markers
5703                    const labels = document.querySelectorAll('.phonetic-label');
5704                    labels.forEach(label => label.remove());
5705
5706                    // Clear drawings
5707                    if (window.xsheetDrawing && window.xsheetDrawing.layerSystem) {
5708                        window.xsheetDrawing.layerSystem.clearAllLayers();
5709                    }
5710
5711                    modified = false;
5712                    updateStatusMessage('Sheet cleared');
5713                }
5714            }
5715
5716            function updateStatusMessage(message) {
5717                statusMessage.textContent = message;
5718                console.log(message);
5719
5720                // Clear status message after 3 seconds
5721                setTimeout(() => {
5722                    if (statusMessage.textContent === message) {
5723                        if (modified) {
5724                            statusMessage.textContent = 'Unsaved changes';
5725                        } else {
5726                            statusMessage.textContent = '';
5727                        }
5728                    }
5729                }, 3000);
5730            }
5731
5732            // Auto-save timer every 2 minutes
5733            setInterval(() => {
5734                if (modified) {
5735                    try {
5736                        const data = collectData();
5737                        localStorage.setItem('animationXSheet_autosave',
JSON.stringify(data));
5738                        updateStatusMessage('Auto-saved');
5739                    } catch (e) {
5740                        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>
```