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 {
```

```
543             constructor(xsheetTable) {
544                 this.xsheetTable = xsheetTable;
545                 this.layers = [];
546                 this.activeLayerIndex = 0;
547                 this.container = null;
548 
549                 this.init();
550             }
551 
552             init() {
553                 // Create container aligned with table
554                 const tableRect = this.xsheetTable.getBoundingClientRect();
555                 this.container = document.createElement('div');
556                 this.container.className = 'drawing-layer-container';
557                 this.container.style.position = 'absolute';
558                 this.container.style.left = `${tableRect.left}px`;
559                 this.container.style.top = `${tableRect.top}px`;
560                 this.container.style.width = `${tableRect.width}px`;
561                 this.container.style.height = `${tableRect.height}px`;
562                 this.container.style.pointerEvents = 'none'; // Initially pass events
    through
563                 this.container.style.zIndex = '5';
564                 this.container.style.overflow = 'hidden'; // Prevent drawings from
    overflowing
565                 document.body.appendChild(this.container);
566 
567                 // Create default background and foreground layers
568                 this.addLayer('background');
569                 this.addLayer('foreground');
570                 this.setActiveLayer(1); // Set foreground as active by default
571 
572                 // Handle window resize and table changes
573                 this.setupResizeHandling();
574             }
575 
576             addLayer(name) {
577                 const canvas = document.createElement('canvas');
578                 canvas.className = `drawing-layer-${name}`;
579                 canvas.width = this.container.clientWidth;
580                 canvas.height = this.container.clientHeight;
581                 canvas.style.position = 'absolute';
582                 canvas.style.left = '0';
583                 canvas.style.top = '0';
584                 canvas.style.pointerEvents = 'none';
585 
586                 this.container.appendChild(canvas);
587 
588                 const layer = {
589                     name: name,
590                     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) {
```

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

```
                    const obj = activeLayer.objects[i];
                    if (obj.containsPoint(x, y)) {
                        return { object: obj, layerIndex: this.activeLayerIndex };
                    }
                }

                // Check other layers from top to bottom
                for (let l = this.layers.length - 1; l >= 0; l--) {
                    if (l === this.activeLayerIndex) continue; // Skip active layer (already checked)

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

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

                return null;
            }

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

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

        // Base class for all drawing objects
        class DrawingObject {
            constructor(props = {}) {
                this.x = props.x || 0;
                this.y = props.y || 0;
                this.color = props.color || '#000000';
```

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

```
                    type: this.type,
                    x: this.x,
                    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);

```

```
887                 return { x: minX, y: minY, width, height };
888             }
889
890             containsPoint(x, y) {
891                 // Check if point is near the line
892                 const lineLength = Math.sqrt(
893                     Math.pow(this.x2 - this.x, 2) + Math.pow(this.y2 - this.y, 2)
894                 );
895
896                 // If line is too short, use a minimum distance
897                 if (lineLength < 1) {
898                     const dx = x - this.x;
899                     const dy = y - this.y;
900                     return Math.sqrt(dx * dx + dy * dy) <= 5;
901                 }
902
903                 // Calculate distance from point to line segment
904                 const t = ((x - this.x) * (this.x2 - this.x) + (y - this.y) * (this.y2 -
    this.y)) / (lineLength * lineLength);
905
906                 if (t < 0) {
907                     // Point is beyond start point
908                     const dx = x - this.x;
909                     const dy = y - this.y;
910                     return Math.sqrt(dx * dx + dy * dy) <= 5;
911                 }
912
913                 if (t > 1) {
914                     // Point is beyond end point
915                     const dx = x - this.x2;
916                     const dy = y - this.y2;
917                     return Math.sqrt(dx * dx + dy * dy) <= 5;
918                 }
919
920                 // Calculate perpendicular distance
921                 const px = this.x + t * (this.x2 - this.x);
922                 const py = this.y + t * (this.y2 - this.y);
923                 const dx = x - px;
924                 const dy = y - py;
925                 return Math.sqrt(dx * dx + dy * dy) <= 5;
926             }
927
928             move(dx, dy) {
929                 super.move(dx, dy);
930                 this.x2 += dx;
931                 this.y2 += dy;
932             }
933
934             toJSON() {
935                 const json = super.toJSON();
```

```
            return {
                ...json,
                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
                };
```

```
1184            }
1185
1186            containsPoint(x, y) {
1187                const bounds = this.getBounds();
1188                return (
1189                    x >= bounds.x && x <= bounds.x + bounds.width &&
1190                    y >= bounds.y && y <= bounds.y + bounds.height
1191                );
1192            }
1193
1194            toJSON() {
1195                const json = super.toJSON();
1196                return {
1197                    ...json,
1198                    text: this.text,
1199                    fontSize: this.fontSize,
1200                    fontFamily: this.fontFamily,
1201                    align: this.align
1202                };
1203            }
1204
1205            static fromJSON(data) {
1206                return new TextObject(data);
1207            }
1208        }
1209
1210        // Image object
1211        class ImageObject extends DrawingObject {
1212            constructor(props = {}) {
1213                super(props);
1214                this.width = props.width || 0;
1215                this.height = props.height || 0;
1216                this.imageUrl = props.imageUrl || '';
1217                this.image = null;
1218                this.loaded = false;
1219                this.type = 'image';
1220
1221                // Load the image if provided
1222                if (this.imageUrl) {
1223                    this.loadImage(this.imageUrl);
1224                }
1225            }
1226
1227            loadImage(url) {
1228                this.image = new Image();
1229                this.image.onload = () => {
1230                    this.loaded = true;
1231
1232                    // If dimensions not specified, use image dimensions
1233                    if (this.width === 0 || this.height === 0) {
```

```
                    this.width = this.image.width;
                    this.height = this.image.height;
                }

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

        draw(context) {
            if (this.loaded && this.image) {
                context.save();
                context.drawImage(this.image, this.x, this.y, this.width, this.height);
                context.restore();
            } else if (!this.loaded) {
                // Draw placeholder while loading
                context.save();
                context.strokeStyle = '#999999';
                context.lineWidth = 1;
                context.strokeRect(this.x, this.y, this.width, this.height);
                context.font = '10px Arial';
                context.fillStyle = '#999999';
                context.fillText('Loading Image...', this.x + 5, this.y + 15);
                context.restore();
            }

            super.draw(context);
        }

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

        containsPoint(x, y) {
            return (
                x >= this.x && x <= this.x + this.width &&
                y >= this.y && y <= this.y + this.height
            );
        }

        toJSON() {
            const json = super.toJSON();
            return {
                ...json,
```

```
                    width: this.width,
                    height: this.height,
                    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);
            }
```

```
1334
1335            drawAnticipation(context) {
1336                context.save();
1337                context.translate(this.x, this.y);
1338                context.scale(this.scale, this.scale);
1339
1340                // Draw curved arrow going back
1341                context.beginPath();
1342                context.moveTo(0, 0);
1343                context.bezierCurveTo(-20, -5, -25, 10, -10, 15);
1344                context.stroke();
1345
1346                // Draw arrowhead
1347                context.beginPath();
1348                context.moveTo(-10, 15);
1349                context.lineTo(-5, 10);
1350                context.lineTo(-15, 5);
1351                context.closePath();
1352                context.fill();
1353
1354                context.restore();
1355            }
1356
1357            drawImpact(context) {
1358                context.save();
1359                context.translate(this.x, this.y);
1360                context.scale(this.scale, this.scale);
1361
1362                // Draw impact star
1363                for (let i = 0; i < 8; i++) {
1364                    const angle = (i / 8) * Math.PI * 2;
1365                    const innerRadius = 5;
1366                    const outerRadius = 15;
1367
1368                    context.beginPath();
1369                    context.moveTo(
1370                        innerRadius * Math.cos(angle),
1371                        innerRadius * Math.sin(angle)
1372                    );
1373                    context.lineTo(
1374                        outerRadius * Math.cos(angle),
1375                        outerRadius * Math.sin(angle)
1376                    );
1377                    context.stroke();
1378                }
1379
1380                context.restore();
1381            }
1382
1383            drawKeyframe(context) {
```

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

            // Draw diamond
            context.beginPath();
            context.moveTo(0, -10);
            context.lineTo(10, 0);
            context.lineTo(0, 10);
            context.lineTo(-10, 0);
            context.closePath();
            context.stroke();
            context.fill();

            context.restore();
        }

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

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

            context.restore();
        }

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

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

            context.restore();
        }

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

```

```
1434                // Draw square
1435                context.beginPath();
1436                context.rect(-7, -7, 14, 14);
1437                context.stroke();
1438
1439                context.restore();
1440            }
1441
1442            getBounds() {
1443                // Approximate bounds based on symbol type
1444                const size = 20 * this.scale;
1445                return {
1446                    x: this.x - size / 2,
1447                    y: this.y - size / 2,
1448                    width: size,
1449                    height: size
1450                };
1451            }
1452
1453            containsPoint(x, y) {
1454                const bounds = this.getBounds();
1455                const dx = x - this.x;
1456                const dy = y - this.y;
1457                const distance = Math.sqrt(dx * dx + dy * dy);
1458
1459                // Use a radius-based check as most symbols are roughly circular
1460                return distance <= bounds.width / 2;
1461            }
1462
1463            toJSON() {
1464                const json = super.toJSON();
1465                return {
1466                    ...json,
1467                    symbolType: this.symbolType,
1468                    scale: this.scale
1469                };
1470            }
1471
1472            static fromJSON(data) {
1473                return new SymbolObject(data);
1474            }
1475        }
1476
1477        // Free-form path (for pen/brush tools)
1478        class PathObject extends DrawingObject {
1479            constructor(props = {}) {
1480                super(props);
1481                this.points = props.points || [];
1482                this.smoothing = props.smoothing !== undefined ? props.smoothing : true;
1483                this.closed = props.closed || false;
```

```
1484                this.fill = props.fill || false;
1485                this.fillColor = props.fillColor || this.color;
1486                this.type = 'path';
1487            }
1488
1489            addPoint(x, y) {
1490                this.points.push({ x, y });
1491            }
1492
1493            draw(context) {
1494                if (this.points.length < 2) return;
1495
1496                context.save();
1497                context.beginPath();
1498                context.strokeStyle = this.color;
1499                context.lineWidth = this.lineWidth;
1500                context.lineJoin = 'round';
1501                context.lineCap = 'round';
1502
1503                // Start from first point
1504                context.moveTo(this.points[0].x, this.points[0].y);
1505
1506                if (this.smoothing && this.points.length > 2) {
1507                    // Draw using bezier curves for smoothing
1508                    for (let i = 1; i < this.points.length - 1; i++) {
1509                        const p1 = this.points[i];
1510                        const p2 = this.points[i + 1];
1511
1512                        const xc = (p1.x + p2.x) / 2;
1513                        const yc = (p1.y + p2.y) / 2;
1514
1515                        context.quadraticCurveTo(p1.x, p1.y, xc, yc);
1516                    }
1517
1518                    // Connect to the last point
1519                    const last = this.points[this.points.length - 1];
1520                    context.lineTo(last.x, last.y);
1521                } else {
1522                    // Simple line segments
1523                    for (let i = 1; i < this.points.length; i++) {
1524                        context.lineTo(this.points[i].x, this.points[i].y);
1525                    }
1526                }
1527
1528                if (this.closed) {
1529                    context.closePath();
1530                }
1531
1532                if (this.fill) {
1533                    context.fillStyle = this.fillColor;
```

```
                    context.fill();
                }

                context.stroke();
                context.restore();

                super.draw(context);
            }

            getBounds() {
                if (this.points.length === 0) {
                    return { x: this.x, y: this.y, width: 0, height: 0 };
                }

                let minX = this.points[0].x;
                let maxX = this.points[0].x;
                let minY = this.points[0].y;
                let maxY = this.points[0].y;

                // Find min/max coordinates
                for (let i = 1; i < this.points.length; i++) {
                    const point = this.points[i];
                    minX = Math.min(minX, point.x);
                    maxX = Math.max(maxX, point.x);
                    minY = Math.min(minY, point.y);
                    maxY = Math.max(maxY, point.y);
                }

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

            containsPoint(x, y) {
                if (this.points.length < 2) return false;

                // Check if point is near any line segment
                for (let i = 0; i < this.points.length - 1; i++) {
                    const p1 = this.points[i];
                    const p2 = this.points[i + 1];

                    const lineLength = Math.sqrt(
                        Math.pow(p2.x - p1.x, 2) + Math.pow(p2.y - p1.y, 2)
                    );

                    // If line is too short, check distance to point
                    if (lineLength < 1) {
```

```
                        const dx = x - p1.x;
                        const dy = y - p1.y;
                        if (Math.sqrt(dx * dx + dy * dy) <= 5) {
                            return true;
                        }
                        continue;
                    }

                    // Calculate distance from point to line segment
                    const t = ((x - p1.x) * (p2.x - p1.x) + (y - p1.y) * (p2.y - p1.y)) / (lineLength * lineLength);

                    if (t < 0) {
                        // Point is beyond start point
                        const dx = x - p1.x;
                        const dy = y - p1.y;
                        if (Math.sqrt(dx * dx + dy * dy) <= 5) {
                            return true;
                        }
                    } else if (t > 1) {
                        // Point is beyond end point
                        const dx = x - p2.x;
                        const dy = y - p2.y;
                        if (Math.sqrt(dx * dx + dy * dy) <= 5) {
                            return true;
                        }
                    } else {
                        // Calculate perpendicular distance
                        const px = p1.x + t * (p2.x - p1.x);
                        const py = p1.y + t * (p2.y - p1.y);
                        const dx = x - px;
                        const dy = y - py;
                        if (Math.sqrt(dx * dx + dy * dy) <= 5) {
                            return true;
                        }
                    }
                }

                // If closed and filled, also check if point is inside
                if (this.closed && this.fill) {
                    // Use point-in-polygon algorithm
                    let inside = false;
                    for (let i = 0, j = this.points.length - 1; i < this.points.length; j = i++) {
                        const xi = this.points[i].x;
                        const yi = this.points[i].y;
                        const xj = this.points[j].x;
                        const yj = this.points[j].y;

                        const intersect = ((yi > y) !== (yj > y)) &&
```

```
                        (x < (xj - xi) * (y - yi) / (yj - yi) + xi);

                    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;
```

```
1682            }
1683
1684            draw(context) {
1685                // Calculate actual coordinates from frame and column
1686                const layerSystem = window.xsheetDrawing.layerSystem;
1687
1688                const startPos = layerSystem.gridToCanvas(this.startFrame,
    this.startColumn);
1689                const endPos = layerSystem.gridToCanvas(this.endFrame, this.endColumn);
1690
1691                if (!startPos || !endPos) return; // Skip if cells not found
1692
1693                context.save();
1694                context.beginPath();
1695                context.strokeStyle = this.color;
1696                context.lineWidth = this.lineWidth;
1697
1698                if (this.dashPattern.length > 0) {
1699                    context.setLineDash(this.dashPattern);
1700                }
1701
1702                context.moveTo(startPos.x, startPos.y);
1703                context.lineTo(endPos.x, endPos.y);
1704                context.stroke();
1705
1706                // Draw arrows if needed
1707                if (this.arrowStart) {
1708                    this.drawArrow(context, endPos.x, endPos.y, startPos.x, startPos.y);
1709                }
1710
1711                if (this.arrowEnd) {
1712                    this.drawArrow(context, startPos.x, startPos.y, endPos.x, endPos.y);
1713                }
1714
1715                context.restore();
1716
1717                // Store computed coordinates for selection/hit testing
1718                this.computedStart = startPos;
1719                this.computedEnd = endPos;
1720
1721                super.draw(context);
1722            }
1723
1724            drawArrow(context, fromX, fromY, toX, toY) {
1725                const angle = Math.atan2(toY - fromY, toX - fromX);
1726
1727                context.save();
1728                context.fillStyle = this.color;
1729                context.beginPath();
1730                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;
                }
```

```
1779
1780                if (t > 1) {
1781                    const dx = x - this.computedEnd.x;
1782                    const dy = y - this.computedEnd.y;
1783                    return Math.sqrt(dx * dx + dy * dy) <= 5;
1784                }
1785
1786                const px = this.computedStart.x + t * (this.computedEnd.x -
    this.computedStart.x);
1787                const py = this.computedStart.y + t * (this.computedEnd.y -
    this.computedStart.y);
1788                const dx = x - px;
1789                const dy = y - py;
1790                return Math.sqrt(dx * dx + dy * dy) <= 5;
1791            }
1792
1793            // This object type doesn't use the regular move method
1794            // Instead, it updates the frame/column values
1795
1796            toJSON() {
1797                const json = super.toJSON();
1798                return {
1799                    ...json,
1800                    startFrame: this.startFrame,
1801                    startColumn: this.startColumn,
1802                    endFrame: this.endFrame,
1803                    endColumn: this.endColumn,
1804                    dashPattern: this.dashPattern,
1805                    arrowStart: this.arrowStart,
1806                    arrowEnd: this.arrowEnd,
1807                    arrowSize: this.arrowSize
1808                };
1809            }
1810
1811            static fromJSON(data) {
1812                return new FrameSpanningLineObject(data);
1813            }
1814        }
1815
1816        // Register all object types in a factory
1817        const DrawingObjectFactory = {
1818            types: {
1819                'drawingObject': DrawingObject,
1820                'line': LineObject,
1821                'arrow': ArrowObject,
1822                'rectangle': RectangleObject,
1823                'ellipse': EllipseObject,
1824                'text': TextObject,
1825                'image': ImageObject,
1826                'symbol': SymbolObject,
```

```
                'path': PathObject,
                'frameSpanningLine': FrameSpanningLineObject
            },

            createFromJSON(data) {
                const Type = this.types[data.type];
                if (Type) {
                    return Type.fromJSON(data);
                }
                return null;
            }
        };

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

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

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

```

```
1877                // Set up toolbar UI
1878                this.createToolbar();
1879            }
1880
1881            setActiveTool(toolName) {
1882                if (this.activeTool) {
1883                    this.activeTool.deactivate();
1884                }
1885
1886                if (this.availableTools[toolName]) {
1887                    this.activeTool = this.availableTools[toolName];
1888                    this.activeTool.activate();
1889                    return true;
1890                }
1891
1892                return false;
1893            }
1894
1895            createToolbar() {
1896                // Create toolbar container
1897                const toolbar = document.createElement('div');
1898                toolbar.className = 'drawing-toolbar';
1899                toolbar.style.display = 'flex';
1900                toolbar.style.flexWrap = 'wrap';
1901                toolbar.style.gap = '5px';
1902                toolbar.style.padding = '10px';
1903                toolbar.style.backgroundColor = '#f5f5f5';
1904                toolbar.style.marginBottom = '10px';
1905                toolbar.style.borderRadius = '5px';
1906
1907                // Add tool buttons
1908                this.addToolButton(toolbar, 'select', 'Select', '👆');
1909                this.addToolButton(toolbar, 'pen', 'Freehand Drawing', '🖍️');
1910                this.addToolButton(toolbar, 'line', 'Line', '—');
1911                this.addToolButton(toolbar, 'arrow', 'Arrow', '→');
1912                this.addToolButton(toolbar, 'rectangle', 'Rectangle', '□');
1913                this.addToolButton(toolbar, 'ellipse', 'Circle/Ellipse', 'o');
1914                this.addToolButton(toolbar, 'text', 'Text', 'T');
1915                this.addToolButton(toolbar, 'image', 'Insert Image', '🖼️');
1916                this.addToolButton(toolbar, 'symbol', 'Animation Symbol', '⭐');
1917                this.addToolButton(toolbar, 'frameLine', 'Multi-Frame Line', '↕️');
1918
1919                // Add separator
1920                toolbar.appendChild(document.createElement('div')).style.borderLeft = '1px
    solid #ccc';
1921                toolbar.lastChild.style.height = '30px';
1922
1923                // Add color picker
1924                const colorContainer = document.createElement('div');
1925                colorContainer.style.display = 'flex';
```

```
                colorContainer.style.alignItems = 'center';
                colorContainer.style.gap = '5px';

                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);
```

```
1976
1977                // Add fill option
1978                const fillContainer = document.createElement('div');
1979                fillContainer.style.display = 'flex';
1980                fillContainer.style.alignItems = 'center';
1981                fillContainer.style.gap = '5px';
1982
1983                const fillCheck = document.createElement('input');
1984                fillCheck.type = 'checkbox';
1985                fillCheck.id = 'drawing-fill';
1986                fillCheck.checked = this.toolSettings.fill;
1987
1988                const fillLabel = document.createElement('label');
1989                fillLabel.textContent = 'Fill';
1990                fillLabel.htmlFor = 'drawing-fill';
1991
1992                fillCheck.addEventListener('change', (e) => {
1993                    this.toolSettings.fill = e.target.checked;
1994                    fillColorPicker.disabled = !e.target.checked;
1995                });
1996
1997                const fillColorPicker = document.createElement('input');
1998                fillColorPicker.type = 'color';
1999                fillColorPicker.id = 'drawing-fill-color';
2000                fillColorPicker.value = this.toolSettings.fillColor;
2001                fillColorPicker.disabled = !this.toolSettings.fill;
2002
2003                fillColorPicker.addEventListener('input', (e) => {
2004                    this.toolSettings.fillColor = e.target.value;
2005                });
2006
2007                fillContainer.appendChild(fillCheck);
2008                fillContainer.appendChild(fillLabel);
2009                fillContainer.appendChild(fillColorPicker);
2010                toolbar.appendChild(fillContainer);
2011
2012                // Add separator
2013                toolbar.appendChild(document.createElement('div')).style.borderLeft = '1px
    solid #ccc';
2014                toolbar.lastChild.style.height = '30px';
2015
2016                // Add layer selector
2017                const layerSelector = document.createElement('select');
2018                layerSelector.id = 'drawing-layer-selector';
2019
2020                // Add options for each layer
2021                this.layerSystem.layers.forEach((layer, index) => {
2022                    const option = document.createElement('option');
2023                    option.value = index;
2024                    option.textContent = layer.name;
```

```
                    if (index === this.layerSystem.activeLayerIndex) {
                        option.selected = true;
                    }
                    layerSelector.appendChild(option);
                });

                layerSelector.addEventListener('change', (e) => {
                    this.layerSystem.setActiveLayer(parseInt(e.target.value));
                });

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

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

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

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

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

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

                toolbar.appendChild(clearButton);

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

```
                }
            }

            addToolButton(toolbar, toolName, tooltip, icon) {
                const button = document.createElement('button');
                button.textContent = icon;
                button.title = tooltip;
                button.style.width = '36px';
                button.style.height = '36px';
                button.style.fontSize = '16px';
                button.style.margin = '0';
                button.style.padding = '5px';
                button.style.borderRadius = '4px';
                button.style.border = '1px solid #ccc';
                button.style.backgroundColor = 'white';
                button.style.cursor = 'pointer';

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

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

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

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

                toolbar.appendChild(button);
            }
        }

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

        // Base tool class
        class DrawingTool {
            constructor(toolSystem) {
                this.toolSystem = toolSystem;
                this.layerSystem = toolSystem.layerSystem;
```

```
2124                this.active = false;
2125                this.settings = toolSystem.toolSettings;
2126            }
2127
2128            activate() {
2129                this.active = true;
2130                this.layerSystem.enableDrawing();
2131                this.attachEvents();
2132            }
2133
2134            deactivate() {
2135                this.active = false;
2136                // turn pointer-events back off so clicks go to the table cells
2137                this.toolSystem.layerSystem.disableDrawing();
2138                this.detachEvents();
2139            }
2140
2141            attachEvents() {
2142                // Override in subclasses
2143            }
2144
2145            detachEvents() {
2146                // Override in subclasses
2147            }
2148        }
2149
2150        // Select tool for manipulating objects
2151        class SelectTool extends DrawingTool {
2152
2153            constructor(toolSystem) {
2154                super(toolSystem);
2155                this.selectedObject = null;
2156                this.selectedLayer = null;
2157                this.dragging = false;
2158                this.dragStart = { x: 0, y: 0 };
2159                this.objectStart = { x: 0, y: 0 };
2160            }
2161            activate() {
2162                this.active = true;
2163                // let pointer-events go to the table cells instead of the canvases
2164                this.layerSystem.disableDrawing();
2165                this.attachEvents();
2166            }
2167
2168            deactivate() {
2169                this.active = false;
2170                this.layerSystem.disableDrawing();
2171                this.detachEvents();
2172            }
2173            attachEvents() {
```

```
2174                const canvas = this.layerSystem.getActiveLayer().canvas;
2175
2176                canvas.addEventListener('mousedown', this.handleMouseDown);
2177                canvas.addEventListener('pointerdown', this.handleMouseDown);
2178                document.addEventListener('mousemove', this.handleMouseMove);
2179                document.addEventListener('pointermove', this.handleMouseMove);
2180                document.addEventListener('mouseup', this.handleMouseUp);
2181                document.addEventListener('pointerup',   this.handleMouseUp);
2182                document.addEventListener('keydown', this.handleKeyDown);
2183            }
2184
2185            detachEvents() {
2186                const canvas = this.layerSystem.getActiveLayer().canvas;
2187
2188                canvas.removeEventListener('mousedown', this.handleMouseDown);
2189                canvas.removeEventListener('pointerdown', this.handleMouseDown);
2190                document.removeEventListener('mousemove', this.handleMouseMove);
2191                document.removeEventListener('pointermove', this.handleMouseMove);
2192                document.removeEventListener('mouseup', this.handleMouseUp);
2193                document.removeEventListener('pointerup',   this.handleMouseUp);
2194                document.removeEventListener('keydown', this.handleKeyDown);
2195
2196                // Clear selection
2197                this.clearSelection();
2198            }
2199
2200            handleMouseDown = (e) => {
2201                // Convert to canvas coordinates
2202                const coords = this.layerSystem.screenToCanvas(e.clientX, e.clientY);
2203
2204                // Check if clicked on an object
2205                const hit = this.layerSystem.findObjectAt(coords.x, coords.y);
2206
2207                if (hit) {
2208                    // Select the object
2209                    this.selectObject(hit.object, hit.layerIndex);
2210
2211                    // Start drag
2212                    this.dragging = true;
2213                    this.dragStart = { x: coords.x, y: coords.y };
2214                    this.objectStart = { x: hit.object.x, y: hit.object.y };
2215                } else {
2216                    // Clear selection if clicked empty space
2217                    this.clearSelection();
2218                }
2219            }
2220
2221            handleMouseMove = (e) => {
2222                if (!this.dragging || !this.selectedObject) return;
2223
```

```
2224                const coords = this.layerSystem.screenToCanvas(e.clientX, e.clientY);
2225
2226                // Calculate move distance
2227                const dx = coords.x - this.dragStart.x;
2228                const dy = coords.y - this.dragStart.y;
2229
2230                // Move the selected object
2231                this.selectedObject.x = this.objectStart.x + dx;
2232                this.selectedObject.y = this.objectStart.y + dy;
2233
2234                // For objects with special move handling
2235                this.selectedObject.move(dx, dy);
2236                this.selectedObject.x = this.objectStart.x; // Reset x as move() already
    handled it
2237                this.selectedObject.y = this.objectStart.y; // Reset y as move() already
    handled it
2238
2239                // Redraw
2240                this.layerSystem.redrawAll();
2241            }
2242
2243            handleMouseUp = () => {
2244                this.dragging = false;
2245            }
2246
2247            handleKeyDown = (e) => {
2248                if (!this.selectedObject) return;
2249
2250                // Delete key
2251                if (e.key === 'Delete' || e.key === 'Backspace') {
2252                    this.layerSystem.removeObject(this.selectedObject, this.selectedLayer);
2253                    this.clearSelection();
2254                }
2255
2256                // Arrow keys for fine movement
2257                const moveDistance = e.shiftKey ? 10 : 1;
2258
2259                if (e.key === 'ArrowLeft') {
2260                    this.selectedObject.move(-moveDistance, 0);
2261                    this.layerSystem.redrawAll();
2262                } else if (e.key === 'ArrowRight') {
2263                    this.selectedObject.move(moveDistance, 0);
2264                    this.layerSystem.redrawAll();
2265                } else if (e.key === 'ArrowUp') {
2266                    this.selectedObject.move(0, -moveDistance);
2267                    this.layerSystem.redrawAll();
2268                } else if (e.key === 'ArrowDown') {
2269                    this.selectedObject.move(0, moveDistance);
2270                    this.layerSystem.redrawAll();
2271                }
```

```
2272            }
2273
2274            selectObject(object, layerIndex) {
2275                // Clear previous selection
2276                this.clearSelection();
2277
2278                // Set new selection
2279                this.selectedObject = object;
2280                this.selectedLayer = layerIndex;
2281                object.selected = true;
2282
2283                // Redraw with selection visual
2284                this.layerSystem.redrawAll();
2285            }
2286
2287            clearSelection() {
2288                if (this.selectedObject) {
2289                    this.selectedObject.selected = false;
2290                    this.selectedObject = null;
2291                    this.selectedLayer = null;
2292                    this.layerSystem.redrawAll();
2293                }
2294            }
2295        }
2296
2297        // Pen tool for free drawing
2298        class PenTool extends DrawingTool {
2299            constructor(toolSystem) {
2300                super(toolSystem);
2301                this.currentPath = null;
2302                this.drawing = false;
2303            }
2304
2305            attachEvents() {
2306                const canvas = this.layerSystem.getActiveLayer().canvas;
2307
2308                canvas.addEventListener('mousedown', this.handleMouseDown);
2309                canvas.addEventListener('pointerdown', this.handleMouseDown);
2310                document.addEventListener('mousemove', this.handleMouseMove);
2311                document.addEventListener('pointermove', this.handleMouseMove);
2312                document.addEventListener('mouseup', this.handleMouseUp);
2313                document.addEventListener('pointerup',   this.handleMouseUp);
2314            }
2315
2316            detachEvents() {
2317                const canvas = this.layerSystem.getActiveLayer().canvas;
2318
2319                canvas.removeEventListener('mousedown', this.handleMouseDown);
2320                canvas.removeEventListener('pointerdown', this.handleMouseDown);
2321                document.removeEventListener('mousemove', this.handleMouseMove);
```

```
2322                document.removeEventListener('pointermove', this.handleMouseMove);
2323                document.removeEventListener('mouseup', this.handleMouseUp);
2324                document.removeEventListener('pointerup',   this.handleMouseUp);
2325
2326                // Finish any in-progress drawing
2327                this.finishDrawing();
2328            }
2329
2330            handleMouseDown = (e) => {
2331                e.preventDefault();
2332                if (e.pointerId != null) {
2333                e.target.setPointerCapture(e.pointerId);
2334                }
2335
2336                const coords = this.layerSystem.screenToCanvas(e.clientX, e.clientY);
2337
2338                // Start a new path
2339                this.currentPath = new PathObject({
2340                    color: this.settings.color,
2341                    lineWidth: this.settings.lineWidth,
2342                    x: coords.x,
2343                    y: coords.y
2344                });
2345
2346                this.currentPath.addPoint(coords.x, coords.y);
2347                this.drawing = true;
2348
2349                // Add to layer
2350                this.layerSystem.addObject(this.currentPath);
2351            }
2352
2353            handleMouseMove = (e) => {
2354                if (!this.drawing || !this.currentPath) return;
2355
2356                const coords = this.layerSystem.screenToCanvas(e.clientX, e.clientY);
2357
2358                // Add point to path
2359                this.currentPath.addPoint(coords.x, coords.y);
2360
2361                // Redraw
2362                this.layerSystem.redrawLayer(this.layerSystem.activeLayerIndex);
2363            }
2364
2365            handleMouseUp = (e) => {
2366                if (e.pointerId != null) {
2367                  e.target.releasePointerCapture(e.pointerId);
2368                }
2369                this.finishDrawing();
2370            }
2371
```

```
2372            finishDrawing() {
2373                if (this.drawing && this.currentPath) {
2374                    // Finish the path
2375                    this.drawing = false;
2376                    this.currentPath = null;
2377                }
2378            }
2379        }
2380
2381        // Line tool
2382        class LineTool extends DrawingTool {
2383            constructor(toolSystem) {
2384                super(toolSystem);
2385                this.startPoint = null;
2386                this.currentLine = null;
2387                this.drawing = false;
2388            }
2389
2390            attachEvents() {
2391                const canvas = this.layerSystem.getActiveLayer().canvas;
2392
2393                canvas.addEventListener('mousedown', this.handleMouseDown);
2394                document.addEventListener('mousemove', this.handleMouseMove);
2395                document.addEventListener('mouseup', this.handleMouseUp);
2396            }
2397
2398            detachEvents() {
2399                const canvas = this.layerSystem.getActiveLayer().canvas;
2400
2401                canvas.removeEventListener('mousedown', this.handleMouseDown);
2402                document.removeEventListener('mousemove', this.handleMouseMove);
2403                document.removeEventListener('mouseup', this.handleMouseUp);
2404
2405                // Finish any in-progress drawing
2406                this.finishDrawing();
2407            }
2408
2409            handleMouseDown = (e) => {
2410                const coords = this.layerSystem.screenToCanvas(e.clientX, e.clientY);
2411
2412                this.startPoint = coords;
2413
2414                // Create temporary line
2415                this.currentLine = new LineObject({
2416                    x: coords.x,
2417                    y: coords.y,
2418                    x2: coords.x,
2419                    y2: coords.y,
2420                    color: this.settings.color,
2421                    lineWidth: this.settings.lineWidth
```

```
2422                 });
2423
2424                 this.drawing = true;
2425
2426                 // Add to layer
2427                 this.layerSystem.addObject(this.currentLine);
2428             }
2429
2430             handleMouseMove = (e) => {
2431                 if (!this.drawing || !this.currentLine) return;
2432
2433                 const coords = this.layerSystem.screenToCanvas(e.clientX, e.clientY);
2434
2435                 // Update end point
2436                 this.currentLine.x2 = coords.x;
2437                 this.currentLine.y2 = coords.y;
2438
2439                 // Redraw
2440                 this.layerSystem.redrawLayer(this.layerSystem.activeLayerIndex);
2441             }
2442
2443             handleMouseUp = () => {
2444                 this.finishDrawing();
2445             }
2446
2447             finishDrawing() {
2448                 if (this.drawing && this.currentLine) {
2449                     // Finish the line
2450                     this.drawing = false;
2451                     this.startPoint = null;
2452                     this.currentLine = null;
2453                 }
2454             }
2455         }
2456
2457         // Arrow tool (extends Line tool)
2458         class ArrowTool extends LineTool {
2459             handleMouseDown = (e) => {
2460                 const coords = this.layerSystem.screenToCanvas(e.clientX, e.clientY);
2461
2462                 this.startPoint = coords;
2463
2464                 // Create temporary arrow
2465                 this.currentLine = new ArrowObject({
2466                     x: coords.x,
2467                     y: coords.y,
2468                     x2: coords.x,
2469                     y2: coords.y,
2470                     color: this.settings.color,
2471                     lineWidth: this.settings.lineWidth
```

```
                });

                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
                });

                this.drawing = true;

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

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

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

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

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

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

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

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

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

```
            }
        }

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

```

```
2622                // Add to layer
2623                this.layerSystem.addObject(this.currentEllipse);
2624            }
2625
2626            handleMouseMove = (e) => {
2627                if (!this.drawing || !this.currentEllipse) return;
2628
2629                const coords = this.layerSystem.screenToCanvas(e.clientX, e.clientY);
2630
2631                // Update radii
2632                this.currentEllipse.radiusX = Math.abs(coords.x - this.center.x);
2633                this.currentEllipse.radiusY = Math.abs(coords.y - this.center.y);
2634
2635                // Redraw
2636                this.layerSystem.redrawLayer(this.layerSystem.activeLayerIndex);
2637            }
2638
2639            handleMouseUp = () => {
2640                this.finishDrawing();
2641            }
2642
2643            finishDrawing() {
2644                if (this.drawing && this.currentEllipse) {
2645                    // Finish the ellipse
2646                    this.drawing = false;
2647                    this.center = null;
2648                    this.currentEllipse = null;
2649                }
2650            }
2651        }
2652
2653        // Text tool
2654        class TextTool extends DrawingTool {
2655            constructor(toolSystem) {
2656                super(toolSystem);
2657                this.textInput = null;
2658            }
2659
2660            attachEvents() {
2661                const canvas = this.layerSystem.getActiveLayer().canvas;
2662
2663                canvas.addEventListener('click', this.handleClick);
2664            }
2665
2666            detachEvents() {
2667                const canvas = this.layerSystem.getActiveLayer().canvas;
2668
2669                canvas.removeEventListener('click', this.handleClick);
2670
2671                // Remove any active text input
```

```
2672                 this.removeTextInput();
2673             }
2674 
2675             handleClick = (e) => {
2676                 const coords = this.layerSystem.screenToCanvas(e.clientX, e.clientY);
2677 
2678                 // Show text input at click position
2679                 this.showTextInput(coords.x, coords.y);
2680             }
2681 
2682             showTextInput(x, y) {
2683                 // Remove any existing text input
2684                 this.removeTextInput();
2685 
2686                 // Create text input element
2687                 this.textInput = document.createElement('div');
2688                 this.textInput.style.position = 'absolute';
2689                 this.textInput.style.zIndex = '100';
2690 
2691                 // Position relative to canvas
2692                 const canvasRect = this.layerSystem.container.getBoundingClientRect();
2693                 this.textInput.style.left = (canvasRect.left + x) + 'px';
2694                 this.textInput.style.top = (canvasRect.top + y) + 'px';
2695 
2696                 // Style the input
2697                 this.textInput.style.backgroundColor = 'white';
2698                 this.textInput.style.border = '1px solid #ccc';
2699                 this.textInput.style.padding = '5px';
2700                 this.textInput.style.borderRadius = '3px';
2701                 this.textInput.style.boxShadow = '0 2px 5px rgba(0,0,0,0.1)';
2702 
2703                 // Create the actual input
2704                 const input = document.createElement('input');
2705                 input.type = 'text';
2706                 input.placeholder = 'Enter text...';
2707                 input.style.width = '200px';
2708                 input.style.padding = '5px';
2709                 input.style.border = '1px solid #ddd';
2710                 input.style.borderRadius = '3px';
2711 
2712                 // Create font size selector
2713                 const sizeSelect = document.createElement('select');
2714                 [8, 10, 12, 14, 16, 18, 20, 24, 36].forEach(size => {
2715                     const option = document.createElement('option');
2716                     option.value = size;
2717                     option.textContent = size + 'px';
2718                     if (size === this.settings.fontSize) {
2719                         option.selected = true;
2720                     }
2721                     sizeSelect.appendChild(option);
```

```
                });

                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';

```

```
2770                // Add button
2771                const addButton = document.createElement('button');
2772                addButton.textContent = 'Add Text';
2773                addButton.style.flex = '1';
2774                addButton.style.padding = '5px';
2775                addButton.style.backgroundColor = '#4CAF50';
2776                addButton.style.color = 'white';
2777                addButton.style.border = 'none';
2778                addButton.style.borderRadius = '3px';
2779                addButton.style.cursor = 'pointer';
2780
2781                addButton.addEventListener('click', () => {
2782                    this.addText(x, y, input.value);
2783                });
2784
2785                // Cancel button
2786                const cancelButton = document.createElement('button');
2787                cancelButton.textContent = 'Cancel';
2788                cancelButton.style.flex = '1';
2789                cancelButton.style.padding = '5px';
2790                cancelButton.style.backgroundColor = '#f44336';
2791                cancelButton.style.color = 'white';
2792                cancelButton.style.border = 'none';
2793                cancelButton.style.borderRadius = '3px';
2794                cancelButton.style.cursor = 'pointer';
2795
2796                cancelButton.addEventListener('click', () => {
2797                    this.removeTextInput();
2798                });
2799
2800                buttons.appendChild(addButton);
2801                buttons.appendChild(cancelButton);
2802                this.textInput.appendChild(buttons);
2803
2804                // Add to document
2805                document.body.appendChild(this.textInput);
2806
2807                // Focus the input
2808                input.focus();
2809
2810                // Handle enter key
2811                input.addEventListener('keydown', (e) => {
2812                    if (e.key === 'Enter') {
2813                        this.addText(x, y, input.value);
2814                    } else if (e.key === 'Escape') {
2815                        this.removeTextInput();
2816                    }
2817                });
2818            }
2819
```

```
2820            removeTextInput() {
2821                if (this.textInput && this.textInput.parentNode) {
2822                    this.textInput.parentNode.removeChild(this.textInput);
2823                    this.textInput = null;
2824                }
2825            }
2826
2827            addText(x, y, text) {
2828                if (!text.trim()) {
2829                    this.removeTextInput();
2830                    return;
2831                }
2832
2833                // Create text object
2834                const textObj = new TextObject({
2835                    x: x,
2836                    y: y,
2837                    text: text,
2838                    color: this.settings.color,
2839                    fontSize: this.settings.fontSize,
2840                    fontFamily: this.settings.fontFamily,
2841                    align: this.settings.textAlign
2842                });
2843
2844                // Add to layer
2845                this.layerSystem.addObject(textObj);
2846
2847                // Remove text input
2848                this.removeTextInput();
2849            }
2850        }
2851
2852        // Image tool
2853        class ImageTool extends DrawingTool {
2854            constructor(toolSystem) {
2855                super(toolSystem);
2856                this.fileInput = null;
2857            }
2858
2859            attachEvents() {
2860                const canvas = this.layerSystem.getActiveLayer().canvas;
2861
2862                canvas.addEventListener('click', this.handleClick);
2863            }
2864
2865            detachEvents() {
2866                const canvas = this.layerSystem.getActiveLayer().canvas;
2867
2868                canvas.removeEventListener('click', this.handleClick);
2869            }
```

```
2870
2871            handleClick = (e) => {
2872                // Show file upload dialog
2873                this.showFileDialog();
2874            }
2875
2876            showFileDialog() {
2877                // Create hidden file input if it doesn't exist
2878                if (!this.fileInput) {
2879                    this.fileInput = document.createElement('input');
2880                    this.fileInput.type = 'file';
2881                    this.fileInput.accept = 'image/*';
2882                    this.fileInput.style.display = 'none';
2883                    document.body.appendChild(this.fileInput);
2884
2885                    this.fileInput.addEventListener('change', (e) => {
2886                        if (e.target.files && e.target.files[0]) {
2887                            this.handleFileSelect(e.target.files[0]);
2888                        }
2889                    });
2890                }
2891
2892                // Trigger file dialog
2893                this.fileInput.click();
2894            }
2895
2896            handleFileSelect(file) {
2897                // Read the file and create a data URL
2898                const reader = new FileReader();
2899
2900                reader.onload = (e) => {
2901                    const imageUrl = e.target.result;
2902
2903                    // Show image placement UI
2904                    this.showImagePlacementUI(imageUrl);
2905                };
2906
2907                reader.readAsDataURL(file);
2908            }
2909
2910            showImagePlacementUI(imageUrl) {
2911                // Create a preview image to get dimensions
2912                const img = new Image();
2913
2914                img.onload = () => {
2915                    // Calculate dimensions (max size 300px width/height while maintaining
    aspect ratio)
2916                    let width = img.width;
2917                    let height = img.height;
2918
```

```
                    const maxSize = 300;
                    if (width > maxSize || height > maxSize) {
                        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) {
                super(toolSystem);
                this.symbolSelector = 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);

                // Remove any active symbol selector
                this.removeSymbolSelector();
            }

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

                // Show symbol selector
                this.showSymbolSelector(coords.x, coords.y);
            }

            showSymbolSelector(x, y) {
                // Remove existing selector if any
                this.removeSymbolSelector();

                // Create symbol selector
                this.symbolSelector = document.createElement('div');
                this.symbolSelector.style.position = 'absolute';
                this.symbolSelector.style.zIndex = '100';

                // Position relative to canvas
                const canvasRect = this.layerSystem.container.getBoundingClientRect();
                this.symbolSelector.style.left = (canvasRect.left + x + 10) + 'px';
                this.symbolSelector.style.top = (canvasRect.top + y + 10) + 'px';

                // Style
                this.symbolSelector.style.backgroundColor = 'white';
                this.symbolSelector.style.border = '1px solid #ccc';
                this.symbolSelector.style.padding = '10px';
                this.symbolSelector.style.borderRadius = '5px';
                this.symbolSelector.style.boxShadow = '0 2px 10px rgba(0,0,0,0.1)';

```

```
3118                // Add title
3119                const title = document.createElement('h4');
3120                title.textContent = 'Select Animation Symbol';
3121                title.style.margin = '0 0 10px 0';
3122                this.symbolSelector.appendChild(title);
3123
3124                // Symbol grid
3125                const symbolGrid = document.createElement('div');
3126                symbolGrid.style.display = 'grid';
3127                symbolGrid.style.gridTemplateColumns = 'repeat(3, 1fr)';
3128                symbolGrid.style.gap = '10px';
3129                symbolGrid.style.marginBottom = '10px';
3130
3131                // Available symbols
3132                const symbols = [
3133                    { type: 'anticipation', name: 'Anticipation' },
3134                    { type: 'impact', name: 'Impact' },
3135                    { type: 'keyframe', name: 'Keyframe' },
3136                    { type: 'inbetween', name: 'Inbetween' },
3137                    { type: 'hold', name: 'Hold' },
3138                    { type: 'default', name: 'Default' }
3139                ];
3140
3141                // Create symbol preview for each type
3142                symbols.forEach(symbol => {
3143                    const symbolItem = document.createElement('div');
3144                    symbolItem.style.display = 'flex';
3145                    symbolItem.style.flexDirection = 'column';
3146                    symbolItem.style.alignItems = 'center';
3147                    symbolItem.style.cursor = 'pointer';
3148                    symbolItem.style.padding = '5px';
3149                    symbolItem.style.border = '1px solid #ddd';
3150                    symbolItem.style.borderRadius = '3px';
3151
3152                    // Create canvas for symbol preview
3153                    const canvas = document.createElement('canvas');
3154                    canvas.width = 50;
3155                    canvas.height = 50;
3156                    const ctx = canvas.getContext('2d');
3157
3158                    // Draw symbol preview
3159                    ctx.save();
3160                    ctx.strokeStyle = this.settings.color;
3161                    ctx.fillStyle = this.settings.color;
3162                    ctx.lineWidth = 2;
3163
3164                    // Draw centered preview
3165                    const symbolObj = new SymbolObject({
3166                        x: 25,
3167                        y: 25,
```

```
                        symbolType: symbol.type,
                        color: this.settings.color,
                        lineWidth: 2,
                        scale: 1.5
                    });

                    symbolObj.draw(ctx);
                    ctx.restore();

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

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

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

                    symbolGrid.appendChild(symbolItem);
                });

                this.symbolSelector.appendChild(symbolGrid);

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

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

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

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

```
3218
3219                scaleInput.addEventListener('input', () => {
3220                    this.settings.symbolScale = parseFloat(scaleInput.value);
3221                    scaleValue.textContent = this.settings.symbolScale + 'x';
3222                });
3223
3224                scaleContainer.appendChild(scaleLabel);
3225                scaleContainer.appendChild(scaleInput);
3226                scaleContainer.appendChild(scaleValue);
3227
3228                this.symbolSelector.appendChild(scaleContainer);
3229
3230                // Add cancel button
3231                const cancelButton = document.createElement('button');
3232                cancelButton.textContent = 'Cancel';
3233                cancelButton.style.width = '100%';
3234                cancelButton.style.padding = '5px';
3235                cancelButton.style.backgroundColor = '#f44336';
3236                cancelButton.style.color = 'white';
3237                cancelButton.style.border = 'none';
3238                cancelButton.style.borderRadius = '3px';
3239                cancelButton.style.cursor = 'pointer';
3240
3241                cancelButton.addEventListener('click', () => {
3242                    this.removeSymbolSelector();
3243                });
3244
3245                this.symbolSelector.appendChild(cancelButton);
3246
3247                // Add to document
3248                document.body.appendChild(this.symbolSelector);
3249            }
3250
3251            removeSymbolSelector() {
3252                if (this.symbolSelector && this.symbolSelector.parentNode) {
3253                    this.symbolSelector.parentNode.removeChild(this.symbolSelector);
3254                    this.symbolSelector = null;
3255                }
3256            }
3257
3258            addSymbol(x, y, symbolType) {
3259                // Create symbol object
3260                const symbolObj = new SymbolObject({
3261                    x: x,
3262                    y: y,
3263                    symbolType: symbolType,
3264                    color: this.settings.color,
3265                    lineWidth: this.settings.lineWidth,
3266                    scale: this.settings.symbolScale
3267                });
```

```
3268
3269                // Add to layer
3270                this.layerSystem.addObject(symbolObj);
3271            }
3272        }
3273
3274        // Frame-spanning line tool (for connecting cells across frames)
3275        class FrameSpanningLineTool extends DrawingTool {
3276            constructor(toolSystem) {
3277                super(toolSystem);
3278                this.startCell = null;
3279                this.currentLine = null;
3280                this.drawing = false;
3281            }
3282
3283            attachEvents() {
3284                // This tool works with the table cells, not the canvas
3285                this.setupCellEvents();
3286            }
3287
3288            detachEvents() {
3289                this.removeCellEvents();
3290
3291                // Finish any in-progress drawing
3292                this.finishDrawing();
3293            }
3294
3295            setupCellEvents() {
3296                // Find all table cells and add mousedown handler
3297                const cells = document.querySelectorAll('#xsheet-table td');
3298                cells.forEach(cell => {
3299                    cell.addEventListener('mousedown', this.handleCellMouseDown);
3300                    cell.addEventListener('mouseup', this.handleCellMouseUp);
3301
3302                    // Add hover effect
3303                    cell.addEventListener('mouseover', this.handleCellHover);
3304                    cell.addEventListener('mouseout', this.handleCellOut);
3305
3306                    // Store original background for hover effect
3307                    if (!cell.dataset.originalBg) {
3308                        cell.dataset.originalBg = cell.style.backgroundColor || '';
3309                    }
3310                });
3311            }
3312
3313            removeCellEvents() {
3314                const cells = document.querySelectorAll('#xsheet-table td');
3315                cells.forEach(cell => {
3316                    cell.removeEventListener('mousedown', this.handleCellMouseDown);
3317                    cell.removeEventListener('mouseup', this.handleCellMouseUp);
```

```
                    cell.removeEventListener('mouseover', this.handleCellHover);
                    cell.removeEventListener('mouseout', this.handleCellOut);

                    // Restore original background
                    if (cell.dataset.originalBg) {
                        cell.style.backgroundColor = cell.dataset.originalBg;
                    }
                });
            }

            handleCellHover = (e) => {
                const cell = e.currentTarget;
                cell.style.backgroundColor = 'rgba(0, 123, 255, 0.2)';
            }

            handleCellOut = (e) => {
                const cell = e.currentTarget;
                if (!this.drawing || cell !== this.startCell) {
                    cell.style.backgroundColor = cell.dataset.originalBg || '';
                }
            }

            handleCellMouseDown = (e) => {
                const cell = e.currentTarget;

                // Find the row and column index
                const row = cell.parentElement;
                const frameNumber = parseInt(row.getAttribute('data-frame') || row.className.replace('frame-', ''));
                const columnIndex = Array.from(row.children).indexOf(cell) + 1;

                if (isNaN(frameNumber) || frameNumber <= 0) return;

                // Store as start cell
                this.startCell = cell;
                this.drawing = true;

                // Highlight the cell
                cell.style.backgroundColor = 'rgba(0, 123, 255, 0.4)';

                // Create temporary frame-spanning line object
                this.currentLine = new FrameSpanningLineObject({
                    startFrame: frameNumber,
                    startColumn: columnIndex,
                    endFrame: frameNumber,
                    endColumn: columnIndex,
                    color: this.settings.color,
                    lineWidth: this.settings.lineWidth,
                    arrowEnd: true
                });
```

```
3367
3368                // Add to layer
3369                this.layerSystem.addObject(this.currentLine);
3370            }
3371
3372            handleCellMouseUp = (e) => {
3373                if (!this.drawing || !this.currentLine) return;
3374
3375                const cell = e.currentTarget;
3376
3377                // Skip if this is the same cell
3378                if (cell === this.startCell) {
3379                    this.finishDrawing();
3380                    return;
3381                }
3382
3383                // Find the row and column index
3384                const row = cell.parentElement;
3385                const frameNumber = parseInt(row.getAttribute('data-frame') ||
    row.className.replace('frame-', ''));
3386                const columnIndex = Array.from(row.children).indexOf(cell) + 1;
3387
3388                if (isNaN(frameNumber) || frameNumber <= 0) {
3389                    this.finishDrawing();
3390                    return;
3391                }
3392
3393                // Update the end point of the line
3394                this.currentLine.endFrame = frameNumber;
3395                this.currentLine.endColumn = columnIndex;
3396
3397                // Redraw
3398                this.layerSystem.redrawLayer(this.layerSystem.activeLayerIndex);
3399
3400                // Finish drawing
3401                this.finishDrawing();
3402            }
3403
3404            finishDrawing() {
3405                if (this.drawing) {
3406                    // Reset flags
3407                    this.drawing = false;
3408
3409                    // If the line is too short (same start and end), remove it
3410                    if (this.currentLine &&
3411                        this.currentLine.startFrame === this.currentLine.endFrame &&
3412                        this.currentLine.startColumn === this.currentLine.endColumn) {
3413                        this.layerSystem.removeObject(this.currentLine);
3414                    }
3415
```

```
3416                    // Clear current line reference
3417                    this.currentLine = null;
3418
3419                    // Reset cell highlights
3420                    if (this.startCell) {
3421                        this.startCell.style.backgroundColor =
this.startCell.dataset.originalBg || '';
3422                        this.startCell = null;
3423                    }
3424                }
3425            }
3426        }
3427
3428        // Eraser tool
3429        class EraserTool extends DrawingTool {
3430            constructor(toolSystem) {
3431                super(toolSystem);
3432            }
3433
3434            attachEvents() {
3435                const canvas = this.layerSystem.getActiveLayer().canvas;
3436
3437                canvas.addEventListener('mousedown', this.handleMouseDown);
3438            }
3439
3440            detachEvents() {
3441                const canvas = this.layerSystem.getActiveLayer().canvas;
3442
3443                canvas.removeEventListener('mousedown', this.handleMouseDown);
3444            }
3445
3446            handleMouseDown = (e) => {
3447                const coords = this.layerSystem.screenToCanvas(e.clientX, e.clientY);
3448
3449                // Find object at click position
3450                const hit = this.layerSystem.findObjectAt(coords.x, coords.y);
3451
3452                if (hit) {
3453                    // Remove the object
3454                    this.layerSystem.removeObject(hit.object, hit.layerIndex);
3455                }
3456            }
3457        }
3458
3459        /**
3460         * INTEGRATION WITH X-SHEET APPLICATION
3461         * Initialize the drawing system, save/load integration, and PDF/print handling
3462         */
3463
3464        // Initialize drawing system when document is loaded
```

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

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

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

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

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

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

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

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

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

            // Override collectData to include drawings
            window.collectData = function() {
                // Call original function to get base data
```

```
3515                const data = originalCollectData ? originalCollectData() : {};
3516
3517                // Add drawing data if drawing system is initialized
3518                if (window.xsheetDrawing && window.xsheetDrawing.layerSystem) {
3519                    data.drawingLayers = [];
3520
3521                    // Collect objects from each layer
3522                    window.xsheetDrawing.layerSystem.layers.forEach(layer => {
3523                        const layerData = {
3524                            name: layer.name,
3525                            visible: layer.visible,
3526                            objects: layer.objects.map(obj => obj.toJSON())
3527                        };
3528
3529                        data.drawingLayers.push(layerData);
3530                    });
3531                }
3532
3533                return data;
3534            };
3535
3536            // Override restoreData to handle drawings
3537            window.restoreData = function(data) {
3538                // Call original function to restore base data
3539                if (originalRestoreData) {
3540                    originalRestoreData(data);
3541                }
3542
3543                // Restore drawing data if available
3544                if (data.drawingLayers && window.xsheetDrawing &&
    window.xsheetDrawing.layerSystem) {
3545                    const layerSystem = window.xsheetDrawing.layerSystem;
3546
3547                    // Clear existing layers
3548                    layerSystem.clearAllLayers();
3549
3550                    // Restore each layer
3551                    data.drawingLayers.forEach((layerData, index) => {
3552                        // Create layer if needed
3553                        if (index >= layerSystem.layers.length) {
3554                            layerSystem.addLayer(layerData.name);
3555                        } else {
3556                            layerSystem.layers[index].name = layerData.name;
3557                            layerSystem.layers[index].visible = layerData.visible;
3558                        }
3559
3560                        // Restore objects
3561                        layerData.objects.forEach(objData => {
3562                            const newObj = DrawingObjectFactory.createFromJSON(objData);
3563                            if (newObj) {
```

```
3564                                layerSystem.layers[index].objects.push(newObj);
3565                            }
3566                        });
3567                    });
3568
3569                    // Redraw
3570                    layerSystem.redrawAll();
3571                }
3572            };
3573        }
3574
3575       // Integrate drawing layers with PDF export and printing
3576        function integrateWithXSheetExport() {
3577            // Store original functions
3578            const originalExportToPDF = window.exportToPDF;
3579            const originalPrintSheet = window.printSheet;
3580
3581            // Override PDF export to include drawings
3582            window.exportToPDF = function () {
3583                if (!window.xsheetDrawing || !window.xsheetDrawing.layerSystem) {
3584                    return originalExportToPDF ? originalExportToPDF() : null;
3585                }
3586
3587                // Save current state
3588                const layerSystem = window.xsheetDrawing.layerSystem;
3589                const layerContainer = layerSystem.container;
3590
3591                // 1) Save original CSS & DOM properties
3592                const originalDisplay = layerContainer.style.display;
3593                const originalPos = layerContainer.style.position;
3594                const originalTop = layerContainer.style.top;
3595                const originalLeft = layerContainer.style.left;
3596                const originalZIndex = layerContainer.style.zIndex;
3597                const originalWidth = layerContainer.style.width;
3598                const originalHeight = layerContainer.style.height;
3599                const originalParent = layerContainer.parentNode;
3600
3601                // 2) Make it visible for capture
3602                layerContainer.style.display = 'block';
3603
3604                // 3) Compute offsets - FIXED POSITIONING LOGIC
3605                const printableArea = document.getElementById('printable-area');
3606                const printableRect = printableArea.getBoundingClientRect();
3607                const tableRect = document.getElementById('xsheet-
table').getBoundingClientRect();
3608
3609                // 4) Reparent into the printable area
3610                printableArea.appendChild(layerContainer);
3611
3612                // 5) Force absolute positioning & explicit size - FIXED POSITIONING LOGIC
```

```
3613                layerContainer.style.position = 'absolute';
3614                layerContainer.style.top = `${tableRect.top - printableRect.top}px`;
3615                layerContainer.style.left = `${tableRect.left - printableRect.left}px`;
3616                layerContainer.style.width = `${tableRect.width}px`;
3617                layerContainer.style.height = `${tableRect.height}px`;
3618                layerContainer.style.zIndex = '1000';
3619
3620                // Force a redraw of all drawing objects
3621                layerSystem.redrawAll();
3622
3623                // 6) Wait a brief moment to ensure drawings are rendered
3624                setTimeout(() => {
3625                    // 7) Generate the PDF
3626                    const result = originalExportToPDF ? originalExportToPDF() : null;
3627
3628                    // 8) Restore everything
3629                    layerContainer.style.display = originalDisplay;
3630                    layerContainer.style.position = originalPos;
3631                    layerContainer.style.top = originalTop;
3632                    layerContainer.style.left = originalLeft;
3633                    layerContainer.style.zIndex = originalZIndex;
3634                    layerContainer.style.width = originalWidth;
3635                    layerContainer.style.height = originalHeight;
3636                    originalParent.appendChild(layerContainer);
3637                }, 50);
3638
3639                return true; // Will be async due to setTimeout
3640            };
3641
3642            // Override print to include drawings
3643            window.printSheet = function () {
3644                // 1) Fallback if no drawing system
3645                if (!window.xsheetDrawing || !window.xsheetDrawing.layerSystem) {
3646                    return originalPrintSheet ? originalPrintSheet() : null;
3647                }
3648
3649                // 2) Grab the container and save its original state
3650                const layerSystem = window.xsheetDrawing.layerSystem;
3651                const layerContainer = layerSystem.container;
3652                const originalDisplay = layerContainer.style.display;
3653                const originalPos = layerContainer.style.position;
3654                const originalTop = layerContainer.style.top;
3655                const originalLeft = layerContainer.style.left;
3656                const originalZIndex = layerContainer.style.zIndex;
3657                const originalWidth = layerContainer.style.width;
3658                const originalHeight = layerContainer.style.height;
3659                const originalParent = layerContainer.parentNode;
3660
3661                // 3) Make it visible for print capture
3662                layerContainer.style.display = 'block';
```

```
3663
3664                // 4) Compute offsets relative to the sheet - FIXED POSITIONING LOGIC
3665                const printableArea = document.getElementById('printable-area');
3666                const printableRect = printableArea.getBoundingClientRect();
3667                const tableRect = document.getElementById('xsheet-
    table').getBoundingClientRect();
3668
3669                // 5) Reparent into the printable-area element
3670                printableArea.appendChild(layerContainer);
3671
3672                // 6) Force absolute positioning & lock size - FIXED POSITIONING LOGIC
3673                layerContainer.style.position = 'absolute';
3674                layerContainer.style.top = `${tableRect.top - printableRect.top}px`;
3675                layerContainer.style.left = `${tableRect.left - printableRect.left}px`;
3676                layerContainer.style.width = `${tableRect.width}px`;
3677                layerContainer.style.height = `${tableRect.height}px`;
3678                layerContainer.style.zIndex = '1000';
3679
3680                // 7) Force a redraw of all drawing objects
3681                layerSystem.redrawAll();
3682
3683                // 8) Add a print class to the container for print-specific CSS
3684                layerContainer.classList.add('printing');
3685
3686                // 9) Call the original print with a slight delay to ensure drawing render
3687                setTimeout(() => {
3688                    const result = originalPrintSheet ? originalPrintSheet() :
    window.print();
3689
3690                    // 10) Restore everything back
3691                    layerContainer.style.display = originalDisplay;
3692                    layerContainer.style.position = originalPos;
3693                    layerContainer.style.top = originalTop;
3694                    layerContainer.style.left = originalLeft;
3695                    layerContainer.style.zIndex = originalZIndex;
3696                    layerContainer.style.width = originalWidth;
3697                    layerContainer.style.height = originalHeight;
3698                    layerContainer.classList.remove('printing');
3699                    originalParent.appendChild(layerContainer);
3700                }, 50);
3701
3702                return true; // Will be async due to setTimeout
3703            };
3704        }
3705
3706
3707        // Setup event listeners for table updates
3708        function setupXSheetUpdateHandling() {
3709            // Hook into existing update functions
3710            if (typeof window.generateTable === 'function') {
```

```
3711                const originalGenerateTable = window.generateTable;
3712                window.generateTable = function() {
3713                    // Call original function
3714                    const result = originalGenerateTable.apply(this, arguments);
3715
3716                    // Fire update event
3717                    document.dispatchEvent(new Event('xsheet-updated'));
3718
3719                    return result;
3720                };
3721            }
3722
3723            if (typeof window.addEightRows === 'function') {
3724                const originalAddEightRows = window.addEightRows;
3725                window.addEightRows = function() {
3726                    // Call original function
3727                    const result = originalAddEightRows.apply(this, arguments);
3728
3729                    // Fire update event
3730                    document.dispatchEvent(new Event('xsheet-updated'));
3731
3732                    return result;
3733                };
3734            }
3735
3736            if (typeof window.updateTemplate === 'function') {
3737                const originalUpdateTemplate = window.updateTemplate;
3738                window.updateTemplate = function() {
3739                    // Call original function
3740                    const result = originalUpdateTemplate.apply(this, arguments);
3741
3742                    // Fire update event
3743                    document.dispatchEvent(new Event('xsheet-updated'));
3744
3745                    return result;
3746                };
3747            }
3748        }
3749
3750        // Original X-Sheet code
3751        // Wait for page load and libraries to initialize
3752        document.addEventListener('DOMContentLoaded', function () {
3753            // Global variables
3754            let frameCount = 96;
3755            let projectName = 'Animation_XSheet_' + new Date().toISOString().split('T')[0];
3756            let modified = false;
3757            let currentTemplate = 'large';
3758
3759            // Audio variables
3760            let audioContext = null;
```

```
3761            let audioBuffer = null;
3762            let audioSource = null;
3763            let audioData = null;
3764            let isPlaying = false;
3765            let startTime = 0;
3766            let startOffset = 0;
3767            let audioFileName = '';
3768            let waveformData = [];
3769            let waveformCanvases = [];
3770            let phonetics = [];
3771            let frameDuration = 1 / 24; // 24fps
3772            let currentFrame = 0;
3773            let phoneticEditPosition = null;
3774
3775            // Initialize UI elements
3776            const templateSelector = document.getElementById('template-selector');
3777            const frameCountInput = document.getElementById('frame-count');
3778            const saveButton = document.getElementById('save-button');
3779            const loadButton = document.getElementById('load-button');
3780            const pdfButton = document.getElementById('pdf-button');
3781            const printButton = document.getElementById('print-button');
3782            const addRowsButton = document.getElementById('add-rows-button');
3783            const clearButton = document.getElementById('clear-button');
3784            const tableBody = document.getElementById('xsheet-body');
3785            const statusMessage = document.getElementById('status-message');
3786            const audioButton = document.getElementById('audio-button');
3787            const audioUpload = document.getElementById('audio-upload');
3788            const playAudioButton = document.getElementById('play-audio');
3789            const stopAudioButton = document.getElementById('stop-audio');
3790            const audioScrubber = document.getElementById('audio-scrubber');
3791            const audioInfo = document.getElementById('audio-info');
3792            const addPhoneticButton = document.getElementById('add-phonetic');
3793            const phoneticInput = document.getElementById('phonetic-input');
3794            const phoneticText = document.getElementById('phonetic-text');
3795            const savePhoneticButton = document.getElementById('save-phonetic');
3796            const cancelPhoneticButton = document.getElementById('cancel-phonetic');
3797
3798            // Date field - set today by default
3799            const dateField = document.getElementById('project-date');
3800            dateField.valueAsDate = new Date();
3801
3802            // Initialize the table
3803            generateTable(frameCount);
3804
3805            // Initialize the drawing system
3806            initDrawingSystem();
3807            setupXSheetUpdateHandling();
3808
3809            // Event Listeners
3810            templateSelector.addEventListener('change', function () {
```

```
3811                currentTemplate = this.value;
3812                updateTemplate();
3813            });
3814
3815            frameCountInput.addEventListener('change', function () {
3816                frameCount = parseInt(this.value);
3817                if (frameCount < 8) frameCount = 8;
3818                generateTable(frameCount);
3819                updateStatusMessage('Frame count updated to ' + frameCount);
3820
3821                // Re-render waveform if audio is loaded
3822                if (audioBuffer) {
3823                    renderWaveform();
3824                }
3825            });
3826
3827            saveButton.addEventListener('click', saveProject);
3828            loadButton.addEventListener('click', loadProject);
3829            pdfButton.addEventListener('click', exportToPDF);
3830            printButton.addEventListener('click', printSheet);
3831            addRowsButton.addEventListener('click', addEightRows);
3832            clearButton.addEventListener('click', clearSheet);
3833
3834            // Audio related event listeners
3835            audioButton.addEventListener('click', function () {
3836                audioUpload.click();
3837            });
3838
3839            audioUpload.addEventListener('change', function (e) {
3840                if (e.target.files.length > 0) {
3841                    const file = e.target.files[0];
3842                    audioFileName = file.name;
3843                    loadAudioFile(file);
3844                }
3845            });
3846
3847            playAudioButton.addEventListener('click', togglePlayAudio);
3848            stopAudioButton.addEventListener('click', stopAudio);
3849            audioScrubber.addEventListener('input', scrubAudio);
3850
3851            addPhoneticButton.addEventListener('click', function () {
3852                if (audioBuffer) {
3853                    showPhoneticInput(null);
3854                } else {
3855                    updateStatusMessage('Please load audio first');
3856                }
3857            });
3858
3859            savePhoneticButton.addEventListener('click', savePhoneticMarker);
3860            cancelPhoneticButton.addEventListener('click', function () {
```

```
3861                phoneticInput.style.display = 'none';
3862            });
3863
3864            // Monitor for changes to set modified flag
3865            document.addEventListener('input', function (e) {
3866                if (!e.target.matches('input, [contenteditable="true"]')) return;
3867                modified = true;
3868
3869                if (e.target.matches('[contenteditable="true"]')) {
3870                    // Add the modified class when content is added
3871                    e.target.classList.add('modified');
3872
3873                    // Remove the modified class when the cell is empty
3874                    if (e.target.textContent.trim() === '') {
3875                        e.target.classList.remove('modified');
3876                    }
3877                }
3878
3879                updateStatusMessage('Changes detected - not saved');
3880            });
3881
3882            // Variables for cell selection
3883            let selectedCells = [];
3884            let isSelecting = false;
3885            let selectionStart = null;
3886            let hasMovedDuringSelection = false;
3887
3888            // Set up multi-cell selection functionality
3889            function setupCellSelection() {
3890                const editableCells = document.querySelectorAll('[contenteditable="true"]');
3891
3892                // Clear selected cells when clicking outside
3893                document.addEventListener('click', function (e) {
3894                    // If this was part of a drag operation, skip the click handler
3895                    if (hasMovedDuringSelection) {
3896                        // We removed this line to fix the selection issue
3897                        // hasMovedDuringSelection = false;
3898                        return;
3899                    }
3900
3901                    // Get the clicked element and check if it's inside an editable cell or
    a selected cell
3902                    const clickedCell = e.target.closest('[contenteditable="true"]');
3903                    const clickedInSelection = e.target.closest('.selected-cell');
3904
3905                    // If not clicking in a cell or selection and not using modifier keys
3906                    if (!clickedCell && !clickedInSelection && !e.ctrlKey && !e.metaKey) {
3907                        clearCellSelection();
3908                    }
3909                });
```

```
3910
3911                // Handle clicks on the table for better cell focusing
3912                document.getElementById('xsheet-table').addEventListener('click', function
    (e) {
3913                    // Find the closest editable cell from the click point
3914                    const targetCell = e.target.closest('[contenteditable="true"]');
3915
3916                    // If we found a cell and we're not in multi-select mode
3917                    if (targetCell && !hasMovedDuringSelection && !e.ctrlKey && !e.metaKey)
    {
3918                        // Focus the cell and position cursor at the end
3919                        if (selectedCells.length === 1 && selectedCells[0] === targetCell) {
3920                            // If already selected, just ensure focus
3921                            targetCell.focus();
3922                            placeCaretAtEnd(targetCell);
3923                        }
3924                    }
3925                });
3926
3927                // Helper function to place caret at the end of content
3928                function placeCaretAtEnd(el) {
3929                    if (document.createRange) {
3930                        const range = document.createRange();
3931                        range.selectNodeContents(el);
3932                        range.collapse(false); // false means collapse to end
3933                        const selection = window.getSelection();
3934                        selection.removeAllRanges();
3935                        selection.addRange(range);
3936                    }
3937                }
3938
3939                // Handle mousedown for selection start
3940                editableCells.forEach(cell => {
3941                    // Mousedown - start potential selection
3942                    cell.addEventListener('mousedown', function (e) {
3943                        // Only handle left mouse button
3944                        if (e.button !== 0) return;
3945
3946                        // If using modifier keys for multi-select
3947                        if (e.ctrlKey || e.metaKey) {
3948                            toggleCellSelection(cell);
3949                            e.preventDefault(); // Prevent text cursor
3950                            return;
3951                        }
3952
3953                        // Clear previous selection unless clicking on already selected cell
3954                        if (!cell.classList.contains('selected-cell')) {
3955                            clearCellSelection();
3956                        }
3957
```

```
3958                        // Store starting cell and reset movement flag
3959                        selectionStart = cell;
3960                        hasMovedDuringSelection = false;
3961                        isSelecting = true;
3962
3963                        // Initially add this cell to selection
3964                        if (!cell.classList.contains('selected-cell')) {
3965                            toggleCellSelection(cell, true);
3966                        }
3967
3968                        // If it's a single click (not start of drag), focus the cell
3969                        cell.focus();
3970
3971                        // Don't prevent default here to allow normal focus behavior
3972                    });
3973
3974                    // Additional keyboard events for selected cells
3975                    cell.addEventListener('keydown', function (e) {
3976                        // If Delete or Backspace key and we have selected cells
3977                        if ((e.key === 'Delete' || e.key === 'Backspace') &&
    selectedCells.length > 1) {
3978                            e.preventDefault();
3979                            clearSelectedCellsContent();
3980                        }
3981
3982                        // Copy with Ctrl+C or Cmd+C
3983                        if ((e.ctrlKey || e.metaKey) && e.key === 'c' &&
    selectedCells.length > 0) {
3984                            e.preventDefault();
3985                            copySelectedCells();
3986                        }
3987
3988                        // Select all cells in row with Ctrl+A or Cmd+A
3989                        if ((e.ctrlKey || e.metaKey) && e.key === 'a') {
3990                            e.preventDefault();
3991                            selectRowCells(cell.parentElement);
3992                        }
3993                    });
3994                });
3995
3996                // Global mousemove handler for extending selection
3997                document.addEventListener('mousemove', function (e) {
3998                    if (!isSelecting) return;
3999
4000                    // Set the flag that we've moved during this selection
4001                    hasMovedDuringSelection = true;
4002
4003                    // Find the element under the cursor
4004                    const elementUnderCursor = document.elementFromPoint(e.clientX,
    e.clientY);
4005                    if (!elementUnderCursor) return;
```

```
4006
4007                    // Find the closest editable cell
4008                    const targetCell =
    elementUnderCursor.closest('[contenteditable="true"]');
4009                    if (targetCell) {
4010                        // Add to selection
4011                        toggleCellSelection(targetCell, true);
4012
4013                        // Prevent text selection during drag
4014                        e.preventDefault();
4015                        if (window.getSelection) {
4016                            window.getSelection().removeAllRanges();
4017                        }
4018                    }
4019                });
4020
4021                // Global mouseup to end selection - THIS IS THE FIXED VERSION
4022                document.addEventListener('mouseup', function (e) {
4023                    // End selection mode but keep selected cells
4024                    if (isSelecting) {
4025                        isSelecting = false;
4026
4027                        // FIX: Use setTimeout to delay resetting the hasMovedDuringSelect-
    ion flag
4028                        // This gives the click handler a chance to see the flag first
4029                        if (hasMovedDuringSelection) {
4030                            setTimeout(function () {
4031                                hasMovedDuringSelection = false;
4032                            }, 10);
4033                        }
4034
4035                        // If this wasn't a drag and there's only one cell selected, focus
    it
4036                        if (!hasMovedDuringSelection && selectedCells.length === 1) {
4037                            const cell = selectedCells[0];
4038                            cell.focus();
4039                            placeCaretAtEnd(cell);
4040                        }
4041                    }
4042
4043                    selectionStart = null;
4044                });
4045
4046                // Listen for paste events
4047                document.addEventListener('paste', function (e) {
4048                    if (selectedCells.length > 0) {
4049                        handlePaste(e);
4050                    }
4051                });
4052            }
4053
```

```
4054            function toggleCellSelection(cell, addOnly = false) {
4055                const index = selectedCells.indexOf(cell);
4056
4057                if (index === -1) {
4058                    // Add to selection
4059                    selectedCells.push(cell);
4060                    cell.classList.add('selected-cell');
4061                } else if (!addOnly) {
4062                    // Remove from selection if not add-only mode
4063                    selectedCells.splice(index, 1);
4064                    cell.classList.remove('selected-cell');
4065                }
4066            }
4067
4068            function clearCellSelection() {
4069                selectedCells.forEach(cell => {
4070                    cell.classList.remove('selected-cell');
4071                });
4072                selectedCells = [];
4073            }
4074
4075            function clearSelectedCellsContent() {
4076                selectedCells.forEach(cell => {
4077                    cell.textContent = '';
4078                    cell.classList.remove('modified');
4079                });
4080                modified = true;
4081                updateStatusMessage('Cleared selected cells');
4082            }
4083
4084            function copySelectedCells() {
4085                if (selectedCells.length === 0) return;
4086
4087                // Create a text representation of selected cells
4088                const cellData = selectedCells.map(cell => cell.textContent ||
    '').join('\t');
4089
4090                // Copy to clipboard using Clipboard API
4091                navigator.clipboard.writeText(cellData)
4092                    .then(() => {
4093                        updateStatusMessage('Copied selected cells to clipboard');
4094                    })
4095                    .catch(err => {
4096                        updateStatusMessage('Failed to copy: ' + err);
4097                    });
4098            }
4099
4100            function handlePaste(e) {
4101                // Prevent default paste behavior
4102                e.preventDefault();
```

```
4103
4104                // Get clipboard data
4105                const clipboardData = e.clipboardData || window.clipboardData;
4106                const pastedText = clipboardData.getData('text');
4107
4108                // If we have a focused element within selection, paste there
4109                const activeElement = document.activeElement;
4110                if (activeElement && selectedCells.includes(activeElement)) {
4111                    activeElement.textContent = pastedText;
4112                    activeElement.classList.add('modified');
4113                    modified = true;
4114                } else if (selectedCells.length > 0) {
4115                    // Otherwise paste into first selected cell
4116                    selectedCells[0].textContent = pastedText;
4117                    selectedCells[0].classList.add('modified');
4118                    modified = true;
4119                }
4120
4121                if (pastedText.trim() === '') {
4122                    selectedCells.forEach(cell => {
4123                        cell.classList.remove('modified');
4124                    });
4125                }
4126
4127                updateStatusMessage('Pasted content into selected cell');
4128            }
4129
4130            function selectRowCells(row) {
4131                // Clear previous selection
4132                clearCellSelection();
4133
4134                // Select all editable cells in the row
4135                const cells = Array.from(row.cells).filter(cell => cell.contentEditable ===
    'true');
4136                cells.forEach(cell => {
4137                    toggleCellSelection(cell, true);
4138                });
4139            }
4140
4141            // Functions
4142            function generateTable(frames) {
4143                tableBody.innerHTML = '';
4144                waveformCanvases = [];
4145
4146                // Create a column container for the waveform
4147                const waveformColContainer = document.createElement('div');
4148                waveformColContainer.className = 'waveform-col-container';
4149
4150                for (let i = 1; i <= frames; i++) {
4151                    const row = document.createElement('tr');
```

```
                    row.className = `frame-${i}`;
                    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();
                }
            }

```

```
4250            function setupCellNavigation() {
4251                const editableCells = document.querySelectorAll('[contenteditable="true"]');
4252
4253                editableCells.forEach(cell => {
4254                    cell.addEventListener('keydown', function (e) {
4255                        // Tab navigation
4256                        if (e.key === 'Tab') {
4257                            e.preventDefault();
4258                            const currentRow = this.parentElement;
4259                            const currentIndex = Array.from(currentRow.cells).indexOf(this);
4260
4261                            if (e.shiftKey) {
4262                                // Shift+Tab moves backward
4263                                let prevCell = null;
4264                                if (currentIndex > 0) {
4265                                    // Find previous editable cell in same row
4266                                    for (let i = currentIndex - 1; i >= 0; i--) {
4267                                        if (currentRow.cells[i].contentEditable === 'true')
    {
4268                                            prevCell = currentRow.cells[i];
4269                                            break;
4270                                        }
4271                                    }
4272                                }
4273
4274                                if (!prevCell) {
4275                                    // Move to previous row, last cell
4276                                    const prevRow = currentRow.previousElementSibling;
4277                                    if (prevRow) {
4278                                        const cells = Array.from(prevRow.cells).filter(c =>
    c.contentEditable === 'true');
4279                                        prevCell = cells[cells.length - 1];
4280                                    }
4281                                }
4282
4283                                if (prevCell) {
4284                                    prevCell.focus();
4285                                    // Place cursor at end of content
4286                                    const range = document.createRange();
4287                                    const sel = window.getSelection();
4288                                    range.selectNodeContents(prevCell);
4289                                    range.collapse(false);
4290                                    sel.removeAllRanges();
4291                                    sel.addRange(range);
4292                                }
4293                            } else {
4294                                // Tab moves forward
4295                                let nextCell = null;
4296                                if (currentIndex < currentRow.cells.length - 1) {
4297                                    // Find next editable cell in same row
```

```
4298                                    for (let i = currentIndex + 1; i < currentRow.cells.length; i++) {
4299                                        if (currentRow.cells[i].contentEditable === 'true') {
4300                                            nextCell = currentRow.cells[i];
4301                                            break;
4302                                        }
4303                                    }
4304                                }
4305
4306                                if (!nextCell) {
4307                                    // Move to next row, first cell
4308                                    const nextRow = currentRow.nextElementSibling;
4309                                    if (nextRow) {
4310                                        nextCell = Array.from(nextRow.cells).find(c => c.contentEditable === 'true');
4311                                    }
4312                                }
4313
4314                                if (nextCell) {
4315                                    nextCell.focus();
4316                                }
4317                            }
4318                        }
4319
4320                        // Enter key to move down
4321                        if (e.key === 'Enter' && !e.shiftKey) {
4322                            e.preventDefault();
4323                            const currentRow = this.parentElement;
4324                            const nextRow = currentRow.nextElementSibling;
4325                            const currentIndex = Array.from(currentRow.cells).indexOf(this);
4326
4327                            if (nextRow) {
4328                                const cells = Array.from(nextRow.cells);
4329                                const samePositionCell = cells[currentIndex];
4330                                if (samePositionCell && samePositionCell.contentEditable === 'true') {
4331                                    samePositionCell.focus();
4332                                }
4333                            }
4334                        }
4335                    });
4336
4337                    // Add listener to remove the modified class when cell is emptied
4338                    cell.addEventListener('keyup', function (e) {
4339                        if ((e.key === 'Delete' || e.key === 'Backspace') && this.textContent.trim() === '') {
4340                            this.classList.remove('modified');
4341                        }
4342                    });
4343                });
```

```
4344
4345                // Setup cell selection functionality
4346                setupCellSelection();
4347            }
4348
4349            // Audio functions
4350            function loadAudioFile(file) {
4351                if (!audioContext) {
4352                    try {
4353                        window.AudioContext = window.AudioContext ||
    window.webkitAudioContext;
4354                        audioContext = new AudioContext();
4355                    } catch (e) {
4356                        updateStatusMessage('Web Audio API is not supported in this
    browser');
4357                        return;
4358                    }
4359                }
4360
4361                const reader = new FileReader();
4362
4363                reader.onload = function (e) {
4364                    const audioData = e.target.result;
4365
4366                    // Update status
4367                    updateStatusMessage('Decoding audio...');
4368
4369                    // Decode the audio data
4370                    audioContext.decodeAudioData(audioData, function (buffer) {
4371                        audioBuffer = buffer;
4372
4373                        // Update audio info
4374                        const duration = buffer.duration;
4375                        const minutes = Math.floor(duration / 60);
4376                        const seconds = Math.floor(duration % 60);
4377                        const frameCount = Math.ceil(duration * 24); // Assuming 24fps
4378
4379                        audioInfo.textContent = `${audioFileName}
    (${minutes}:${seconds.toString().padStart(2, '0')}, ${frameCount} frames @ 24fps)`;
4380
4381                        // Enable audio controls
4382                        playAudioButton.disabled = false;
4383                        stopAudioButton.disabled = false;
4384                        audioScrubber.disabled = false;
4385
4386                        // Generate waveform visualization
4387                        generateWaveformData(buffer);
4388
4389                        // Update status
4390                        updateStatusMessage('Audio loaded: ' + audioFileName);
4391
```

```
4392                        // If the x-sheet has fewer frames than the audio, suggest
    increasing
4393                        if (frameCount > frameCountInput.value) {
4394                            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?`)) {
4395                                frameCountInput.value = frameCount;
4396                                frameCount = frameCount;
4397                                generateTable(frameCount);
4398                                renderWaveform();
4399                            } else {
4400                                renderWaveform();
4401                            }
4402                        } else {
4403                            renderWaveform();
4404                        }
4405                    }, function (e) {
4406                        updateStatusMessage('Error decoding audio: ' + e.message);
4407                    });
4408                };
4409
4410                reader.onerror = function () {
4411                    updateStatusMessage('Error reading audio file');
4412                };
4413
4414                reader.readAsArrayBuffer(file);
4415            }
4416
4417            function generateWaveformData(buffer) {
4418                // Get the raw audio data from the buffer
4419                const rawData = buffer.getChannelData(0); // Use first channel
4420
4421                // Calculate how many samples we need for our visualization
4422                const totalSamples = rawData.length;
4423
4424                // Process for visualization - we need to reduce the resolution
4425                // to make it efficient to display
4426                waveformData = [];
4427
4428                // Calculate desired number of points for the visualization
4429                // For vertical waveform, we want more detail
4430                const pointsPerSecond = 100; // Increase detail for vertical display
4431                const totalPoints = Math.ceil(buffer.duration * pointsPerSecond);
4432
4433                // Calculate step size
4434                const step = Math.floor(totalSamples / totalPoints);
4435
4436                // Build the waveform data array
4437                for (let i = 0; i < totalPoints; i++) {
4438                    const index = Math.floor(i * step);
4439                    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;
```

```
4490
4491                    const rect = overlay.getBoundingClientRect();
4492                    const y = e.clientY - rect.top;
4493                    const percentage = y / rect.height;
4494
4495                    // Calculate time point in the audio
4496                    const timePoint = percentage * audioBuffer.duration;
4497
4498                    // Update audio position
4499                    audioScrubber.value = (timePoint / audioBuffer.duration) * 100;
4500                    if (isPlaying) {
4501                        stopAudio();
4502                        startOffset = timePoint;
4503                        playAudio();
4504                    } else {
4505                        startOffset = timePoint;
4506                    }
4507
4508                    // Update the UI to show frame position
4509                    updateFrameMarker();
4510                });
4511
4512                // Variables for scrubbing
4513                let isScrubbing = false;
4514
4515                // Add scrubbing functionality (dragging while holding mouse button)
4516                overlay.addEventListener('mousedown', function (e) {
4517                    if (!audioBuffer) return;
4518                    if (e.button !== 0) return; // Only respond to left mouse button
4519
4520                    isScrubbing = true;
4521
4522                    // Pause any playing audio
4523                    if (isPlaying) {
4524                        pauseAudio();
4525                    }
4526                    // — NEW: allow pen (pointerType="pen") to scrub as well —
4527                    overlay.addEventListener('pointerdown', function (e) {
4528                        if (!audioBuffer || e.pointerType !== 'pen') return;
4529                        isScrubbing = true;
4530                        if (isPlaying) pauseAudio();          // pause continuous playback
4531                        updateScrubPosition(e);               // scrub to this location
4532                        e.preventDefault();                   // prevent default pen
    behavior
4533                    });
4534                    overlay.addEventListener('pointermove', function (e) {
4535                        // — FIX: only scrub while the pen is physically pressing
    (pressure>0)
4536                        if (isScrubbing && e.pointerType === 'pen' && e.pressure > 0) {
4537                            updateScrubPosition(e);
```

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

```
                    updateFrameMarker();
                }

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

                overlay.addEventListener('mouseleave', function () {
                    isScrubbing = false;
                });

                // Function to play a short snippet at the scrub position
                let scrubSource = null;
                function playScrubAudio(timePoint) {
                    if (scrubSource) {
                        try {
                            scrubSource.stop();
                        } catch (e) {
                            // Ignore errors when stopping already stopped source
                        }
                    }

                    // Play a very short snippet at the current position
                    scrubSource = audioContext.createBufferSource();
                    scrubSource.buffer = audioBuffer;
                    scrubSource.connect(audioContext.destination);

                    // Play just a short snippet (equivalent to 1-2 frames at 24fps)
                    const snippetDuration = 1 / 12; // 1/12 of a second (2 frames at 24fps)
                    scrubSource.start(0, timePoint, snippetDuration);
                }

                // Right-click to add phonetic marker
                overlay.addEventListener('contextmenu', function (e) {
                    e.preventDefault();
                    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;

                    showPhoneticInput(timePoint);
                    return false;
                });

                waveformContainer.appendChild(overlay);
```

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

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

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

```
                    editIndex: editIndex
                };

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

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

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

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

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

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

```
                    }
                }

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

                // Re-render markers
                renderWaveform();

                // Mark as modified
                modified = true;
            }

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

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

```

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

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

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

```
                                format: [pdfWidth, pdfHeight]
                            });

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

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

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

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

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

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

                    updateStatusMessage('PDF exported successfully');
                }

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

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

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

                        // Create a canvas for this cell
```

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

                        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);
                                    }
                                }

```

```
5320                                // Draw left side (mirror)
5321                                for (let i = endPoint; i >= startPoint && i < waveformData.length; i--) {
5322                                    const relativePos = (i - startPoint) / (endPoint - startPoint);
5323                                    const y = relativePos * canvas.height;
5324
5325                                    const amplitude = waveformData[i] * (canvas.width * 0.4);
5326                                    const x = (canvas.width / 2) - amplitude;
5327
5328                                    ctx.lineTo(x, y);
5329                                }
5330
5331                                ctx.stroke();
5332                            }
5333
5334                            // Look for phonetic markers that might be in this frame
5335                            if (phonetics && phonetics.length > 0) {
5336                                phonetics.forEach(phonetic => {
5337                                    // Calculate which frame this phonetic marker belongs to
5338                                    const frameOfMarker = Math.floor(phonetic.time / frameDuration) + 1;
5339
5340                                    // If it's in this frame, draw it
5341                                    if (frameOfMarker === frameNum) {
5342                                        // Calculate position within cell
5343                                        const posInFrameRatio = (phonetic.time - ((frameNum - 1) * frameDuration)) / frameDuration;
5344                                        const yPos = posInFrameRatio * canvas.height;
5345
5346                                        // Draw marker line
5347                                        ctx.beginPath();
5348                                        ctx.strokeStyle = '#ff0000';
5349                                        ctx.lineWidth = 1;
5350                                        ctx.moveTo(0, yPos);
5351                                        ctx.lineTo(canvas.width, yPos);
5352                                        ctx.stroke();
5353
5354                                        // Add label if it fits
5355                                        if (canvas.width > 30) {
5356                                            ctx.fillStyle = 'rgba(255, 255, 255, 0.7)';
5357                                            const textWidth = ctx.measureText(phonetic.text).width;
5358                                            if (textWidth < canvas.width - 4) {
5359                                                ctx.fillRect(2, yPos - 8, textWidth + 4, 14);
5360                                                ctx.fillStyle = '#ff0000';
5361                                                ctx.font = '8px Arial';
5362                                                ctx.fillText(phonetic.text, 4, yPos + 4);
5363                                            }
```

```
                                    }
                                }
                            });
                        }
                    }

                    // 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';
```

```
5693
5694                    // Stop any playing audio
5695                    stopAudio();
5696
5697                    // Clear waveform visualization
5698                    waveformCanvases.forEach(canvas => {
5699                        const ctx = canvas.getContext('2d');
5700                        ctx.clearRect(0, 0, canvas.width, canvas.height);
5701                    });
5702
5703                    // Clear phonetic markers
5704                    const labels = document.querySelectorAll('.phonetic-label');
5705                    labels.forEach(label => label.remove());
5706
5707                    // Clear drawings
5708                    if (window.xsheetDrawing && window.xsheetDrawing.layerSystem) {
5709                        window.xsheetDrawing.layerSystem.clearAllLayers();
5710                    }
5711
5712                    modified = false;
5713                    updateStatusMessage('Sheet cleared');
5714                }
5715            }
5716
5717            function updateStatusMessage(message) {
5718                statusMessage.textContent = message;
5719                console.log(message);
5720
5721                // Clear status message after 3 seconds
5722                setTimeout(() => {
5723                    if (statusMessage.textContent === message) {
5724                        if (modified) {
5725                            statusMessage.textContent = 'Unsaved changes';
5726                        } else {
5727                            statusMessage.textContent = '';
5728                        }
5729                    }
5730                }, 3000);
5731            }
5732
5733            // Auto-save timer every 2 minutes
5734            setInterval(() => {
5735                if (modified) {
5736                    try {
5737                        const data = collectData();
5738                        localStorage.setItem('animationXSheet_autosave',
JSON.stringify(data));
5739                        updateStatusMessage('Auto-saved');
5740                    } catch (e) {
5741                        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>
```