```
import "../styles/ContouringWorkspace.css";
import React, { useState, useEffect, useRef } from "react";

import polygonClipping from "polygon-clipping";
import {
  Point,
  Rect,
  translateDisplayToWorld,
  convertRectWorldToDisplay,
} from "../utils/Transformations";

import {
  translateWorldToDisplay,
  convertRectDisplayToWorld,
} from "../utils/Transformations";
import { ZOOM_STEP } from "../utils/constants";

import { eventBus } from "../utils/eventBus";

const TransversalView = ({
  // images are in [Y X Z] form
  images,
  HUValues,
  pixelSpacing,
  zList,
  minX,
  minY,
  isBrushEnabled,
  isEraserEnabled,
  eraserType,
```

```
  contourStyle, // ContourStyle is moved up to Contouring Workspace component as it is shared by all views
  brushSizeInMM,
  selectedStructureSetUID,
  selectedStructureUID,
  displayStructuresUIDs,
  structureSets,
  setStructureSets,
  transversalCanvasDimensions,
  isCrosshairEnabled,
  isRulerEnabled,
  isProtractorEnabled,
  isMeasureCircleEnabled,
  isPixelProbeEnabled,
  isRectangleProbeEnabled,
  isEllipseProbeEnabled,
  isTextAnnotationEnabled,
  isDeleteAnnotationEnabled,
  currentZSlice, // this is the index of Z (0,1,...)
  setCurrentZSlice,
  openAutoSegDialog,
  ROI3dCT,
  setROI3dCT,
  setCurrentXSlice,
  setCurrentYSlice,
  currentXSlice,
  currentYSlice,
  pointsToDisplayAuto,
  onActiveViewClick,
  activeView,
}) => {
```

```
  const originalHeight = images.length;
  const originalWidth = images[0].length;
  const zSlices = images[0][0].length;

  const [canvasWidth, setCanvasWidth] = useState(
    transversalCanvasDimensions.width
  );
  const [canvasHeight, setCanvasHeight] = useState(
    transversalCanvasDimensions.height
  );

  const [ZCoordinate, setZCoordinate] = useState(
    zList[Math.floor(zSlices / 2)]
  );
  const [CTCoordinate, setCTCoordinate] = useState(null);
  const [currentHU, setCurrentHU] = useState(null);

  // For ruler
  const [rulerPointsCT, setRulerPointsCT] = useState({}); // array of objects of type: {start:Point, end:Point, distance:number }
  const [isRulerDrag, setIsRulerDrag] = useState(false);
  const [rulerStartPoint, setRulerStartPoint] = useState(null);
  const [currentMouse, setCurrentMouse] = useState(null); // This stores the current point for both the ruler and the protractor
  const [hoveredRulerIndex, setHoveredRulerIndex] = useState(null);
  const [resizeRulerIndex, setResizeRulerIndex] = useState(null); // Type of object: {index: 1, point: 'start'/'end'}
  const [movedRulerIndex, setMovedRulerIndex] = useState(null);
  const [isRulerMoving, setIsRulerMoving] = useState(null);
  const [rulerStartMovePoint, setRulerStartMovePoint] = useState(null);
```

```
  const [pendingMoveAnnotation, setPendingMoveAnnotation] = useState(null);
// shape:
// {
//   index,
//   originalHead,
//   originalTail
// }

  // For protractor
  const [anglePointsCT, setAnglePointsCT] = useState({});
  const [currentAnglePoints, setCurrentAnglePoints] = useState([]);
  const [hoveredProtractorIndex, setHoveredProtractorIndex] = useState(null);
  const [resizeProtractorInfo, setResizeProtractorInfo] = useState(null);
  const [moveProtractorArmInfo, setMoveProtractorArmInfo] = useState(null);
  const [lastPhysicalPoint, setLastPhysicalPoint] = useState(null);
  const [draggingType, setDraggingType] = useState(null);

  // For measure circle feature
  const [circlePointsCT, setCirclePointsCT] = useState({});
  const [currentCircleStart, setCurrentCircleStart] = useState(null);
  const [hoveredCircleIndex, setHoveredCircleIndex] = useState(null);
  const [movedCircleIndex, setMovedCircleIndex] = useState(null);
  const [resizeCircleIndex, setResizeCircleIndex] = useState(null);
  const [circleStartMovePoint, setCircleStartMovePoint] = useState(null);
  const [isCircleMoving, setIsCircleMoving] = useState(false);

  // For rectangle probe tool
  const [rectangleProbesCT, setRectangleProbesCT] = useState({}); // This is the array of objects: {p1:
Point, p2: Point, area:number, mean: number, stdDev: number}
  const [currentRectangleStart, setCurrentRectangleStart] = useState(null);
  const [hoveredRectangleIndex, setHoveredRectangleIndex] = useState(null);
```

```
  const [movedRectangleIndex, setMovedRectangleIndex] = useState(null);
  const [resizeRectangleIndex, setResizeRectangleIndex] = useState(null); // { index: number, point: "p1" | "p2" }
  const [rectangleStartMovePoint, setRectangleStartMovePoint] = useState(null);
  const [isRectangleMoving, setIsRectangleMoving] = useState(false);

  // For ellipse probe tool
  const [ellipseProbesCT, setEllipseProbesCT] = useState({});
  const [currentEllipseStart, setCurrentEllipseStart] = useState(null);
  const [hoveredEllipseIndex, setHoveredEllipseIndex] = useState(null);
  const [movedEllipseIndex, setMovedEllipseIndex] = useState(null);
  const [resizeEllipseIndex, setResizeEllipseIndex] = useState(null); // { index: number, point: "p1" | "p2" }
  const [ellipseStartMovePoint, setEllipseStartMovePoint] = useState(null);
  const [isEllipseMoving, setIsEllipseMoving] = useState(false);

  // For pixel probe tool
  const [pixelProbesCT, setPixelProbesCT] = useState({});
  const [hoveredPixelIndex, setHoveredPixelIndex] = useState(null);
  const [movedPixelProbeIndex, setMovedPixelProbeIndex] = useState(null);
  const [isMovingPixelProbe, setIsMovingPixelProbe] = useState(false);
  const [previewPixelPoint, setPreviewPixelPoint] = useState(null);

  // For text annotations
  const [textAnnotations, setTextAnnotations] = useState({}); // Shruti
  const [currentTextTail, setCurrentTextTail] = useState(null);
  const [hoveredTextAnnotationIndex, setHoveredTextAnnotationIndex] =
    useState(null);
  const [resizedTextAnnotationsIndex, setResizedTextAnnotationsIndex] =
    useState(null); // { index, point: "head" | "tail" | "body" }
  const [movedTextAnnotationIndex, setMovedTextAnnotationIndex] =
    useState(null);
```

```
  const [textStartMovePoint, setTextStartMovePoint] = useState(null);
  const [isTextMoving, setIsTextMoving] = useState(false);
  const [resizingTextPointType, setResizingTextPointType] = useState(null);
  const [activeTextValue, setActiveTextValue] = useState(null);
const [pendingTextAnnotation, setPendingTextAnnotation] = useState(null);//added
const [isConfirmingText, setIsConfirmingText] = useState(false);//added
const [pendingEditAnnotation, setPendingEditAnnotation] = useState(null);
// shape: { index, oldText, newText }

  const [isMouseDown, setIsMouseDown] = useState(null);
  const canvasRef = useRef(null);
  const [translateX, setTranslateX] = useState(0); // Accounts for panning
  const [translateY, setTranslateY] = useState(0); // Accounts for panning
  const [startPanX, setStartPanX] = useState(0);
  const [startPanY, setStartPanY] = useState(0);
  const [isPanning, setIsPanning] = useState(false);

  const bodyHeight = (originalHeight - 1) * pixelSpacing[1];
  const bodyWidth = (originalWidth - 1) * pixelSpacing[0];
  const [brushSize, setBrushSize] = useState(
    Math.round((transversalCanvasDimensions.width / bodyWidth) * brushSizeInMM)
  );

  let positionRect = new Rect(minX, minY, bodyWidth, bodyHeight);
  const [activeBrush, setActiveBrush] = useState(null);

  const [points, setPoints] = useState({});
  const [pointBrushPolygons, setPointBrushPolygons] = useState([]);
  const [splinePoints, setSplinePoints] = useState([]);
  const [freePointBrushPolygons, setFreePointBrushPolygons] = useState({});
```

```
// For autosegmentation ROI
const [dragging, setDragging] = useState(false);
const [resizing, setResizing] = useState(null);
const [offset, setOffset] = useState({ x: 0, y: 0 });
// const [ROI2dDisplay, setROI2dDisplay] = useState({
//   x: 0,
//   y: 0,
//   width: canvasWidth,
//   height: canvasHeight,
// }); // Just a default case in the beginning - this will be changed whenever CT ic changed

// States to handle circular brush
const [isBrushing, setIsBrushing] = useState(false);
const [prevPoint, setPrevPoint] = useState(null);
const [brushPolygons, setBrushPolygons] = useState([]);
const unionRef = useRef(null);
const scale = 1; //adjust if sub pixel accuracy is needed

const [isPointInside, setIsPointInside] = useState({
  status: false,
  polygonIndices: [],
});
const [isShiftPressed, setIsShiftPressed] = useState(false);

// States to handle eraser
const eraserRef = useRef(null);
const [isErasing, setIsErasing] = useState(false);
const [eraserPolygons, setEraserPolygons] = useState([]);

const [middleDisplayPoint, setMiddleDisplayPoint] = useState({
  x: 0,
```

```
  y: 0,
});
const [draggedRlAp, setDraggedRlAp] = useState(null); // "RL" or "AP"
const [hoveredRlAp, setHoveredRlAp] = useState(null); // "RL" or "AP"

function isPointInPolygon(point, polygon) {
  if (!polygon || polygon.length === 0) return false;

  // Step 1: Calculate bounding box (optimization)
  let minX = polygon[0].x;
  let maxX = polygon[0].x;
  let minY = polygon[0].y;
  let maxY = polygon[0].y;

  for (let i = 1; i < polygon.length; i++) {
    const q = polygon[i];
    minX = Math.min(q.x, minX);
    maxX = Math.max(q.x, maxX);
    minY = Math.min(q.y, minY);
    maxY = Math.max(q.y, maxY);
  }

  // If point is outside bounding box, it's definitely not inside polygon
  if (point.x < minX || point.x > maxX || point.y < minY || point.y > maxY) {
    return false;
  }

  // Step 2: Ray-casting algorithm — pnpoly
  let inside = false;
  for (let i = 0, j = polygon.length - 1; i < polygon.length; j = i++) {
    const xi = polygon[i].x;
```

```
    const yi = polygon[i].y;
    const xj = polygon[j].x;
    const yj = polygon[j].y;

    const intersect =
      yi > point.y !== yj > point.y &&
      point.x < ((xj - xi) * (point.y - yi)) / (yj - yi) + xi;

    if (intersect) inside = !inside;
  }

  return inside;
}

// Eraser functions
const unionEraser = (circles) => {
  try {
    eraserRef.current = eraserRef.current
      ? polygonClipping.union(eraserRef.current, ...circles)
      : polygonClipping.union(...circles);
  } catch (err) {
    console.warn("Eraser-union failed", err);
  }
  return eraserRef.current || [];
};

useEffect(() => {
  setBrushSize(Math.round(brushSizeInMM * (canvasWidth / bodyWidth)));
}, [canvasWidth, bodyWidth, brushSizeInMM]);

useEffect(() => {
```

```
  const textAnn = textAnnotations;
  console.log("Text Annotations: ", textAnnotations);
}, [textAnnotations]);

// Circular brush functions
const drawPolygons = (ctx, polygons, isFilled = false) => {
  if (!polygons || !(polygons.length > 0)) return;
  //  ctx.clearRect(0, 0, canvasWidth, canvasHeight);
  //ctx.fillStyle = "rgba(255, 165, 0, 0.5)";

  if (isFilled) {
    ctx.fillStyle = "rgba(224, 255, 255, 0.4)"; // light cyan with transparency
    ctx.lineWidth = 1;
  } else {
    ctx.lineWidth = 2;
  }

  ctx.strokeStyle = "aqua";

  for (const poly of polygons) {
    for (const ring of poly) {
      ctx.beginPath();
      ring.forEach(([x, y], i) => {
        if (i === 0) ctx.moveTo(x, y);
        else ctx.lineTo(x, y);
      });
      ctx.closePath();
      if (isFilled) {
        ctx.fill();
      }
      ctx.stroke();
```

```
    }
  }
};

const createCirclePolygon = ({ x, y, radius }) => {
  const points = [];
  let segments = 36; // default: circle
  if (eraserType === "Square") {
    segments = 4; // for square
  }

  for (let i = 0; i < segments; i++) {
    const angle = (i * 2 * Math.PI) / segments;
    const px = x + radius * Math.cos(angle);
    const py = y + radius * Math.sin(angle);
    points.push([px * scale, py * scale]);
  }
  points.push(points[0]);
  return [[points]];
};

const unionPolygons = (polys) => {
  try {
    if (unionRef.current) {
      unionRef.current = polygonClipping.union(unionRef.current, ...polys);
    } else {
      unionRef.current = polygonClipping.union(...polys);
    }
    return unionRef.current;
  } catch (err) {
    console.warn("Union failed:", err);
```

```
    return unionRef.current || [];
  }
};

const getCanvasPoint = (event) => {
  if (!canvasRef.current) return null;
  const rect = canvasRef.current.getBoundingClientRect();
  return {
    x: event.clientX - rect.left,
    y: event.clientY - rect.top,
  };
};

const findDistance = (p1, p2) => {
  return Math.sqrt((p1.x - p2.x) ** 2 + (p1.y - p2.y) ** 2);
};

const lerp = (x, y, a) => x * (1 - a) + y * a;

const convertSliceToDisplay = (
  pSlice,
  qSlice,
  minP,
  minQ,
  pixelSpacingP,
  pixelSpacingQ
) => {
  //First convert slice to CT
  const PCT = pSlice * pixelSpacingP + minP;
  const QCT = qSlice * pixelSpacingQ + minQ;
  //Then convert CT to display
```

```
  const PQDisplay = translateWorldToDisplay(
    new Point(PCT, QCT),
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth
  );
  return PQDisplay;
};

useEffect(() => {
  const currentMidPoint = convertSliceToDisplay(
    currentXSlice,
    currentYSlice,
    minX,
    minY,
    pixelSpacing[0],
    pixelSpacing[1]
  );
  setMiddleDisplayPoint(currentMidPoint);
}, [
  canvasWidth,
  canvasHeight,
  translateX,
  translateY,
  currentXSlice,
  currentYSlice,
]);
```

```
//#region Autosegmentation of body functions
const drawROI = (ctx) => {
  const ROIRectCT = new Rect(
    ROI3dCT.x,
    ROI3dCT.y,
    ROI3dCT.width,
    ROI3dCT.height
  );
  const ROIRectDisplay = convertRectWorldToDisplay(
    ROIRectCT,
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth
  );

  ctx.fillStyle = "rgba(0,255,255,0.2)";
  ctx.fillRect(
    ROIRectDisplay.x,
    ROIRectDisplay.y,
    ROIRectDisplay.width,
    ROIRectDisplay.height
  );
  ctx.strokeStyle = "orange";
  ctx.strokeRect(
    ROIRectDisplay.x,
    ROIRectDisplay.y,
```

```
    ROIRectDisplay.width,
    ROIRectDisplay.height
  );
  drawHandle(ctx, ROIRectDisplay.x, ROIRectDisplay.y);
  drawHandle(ctx, ROIRectDisplay.x + ROIRectDisplay.width, ROIRectDisplay.y);
  drawHandle(ctx, ROIRectDisplay.x, ROIRectDisplay.y + ROIRectDisplay.height);
  drawHandle(
    ctx,
    ROIRectDisplay.x + ROIRectDisplay.width,
    ROIRectDisplay.y + ROIRectDisplay.height
  );
};

const drawHandle = (ctx, x, y) => {
  ctx.fillStyle = "blue";
  ctx.fillRect(x - 5, y - 5, 10, 10);
};

const getHandle = (mouseX, mouseY) => {
  const CTRect2D = new Rect(
    ROI3dCT.x,
    ROI3dCT.y,
    ROI3dCT.width,
    ROI3dCT.height
  );

  const ROIRectDisplay = convertRectWorldToDisplay(
    CTRect2D,
    canvasWidth,
    canvasHeight,
    translateX,
```

```
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth
  );

  const handles = {
    "top-left": { x: ROIRectDisplay.x, y: ROIRectDisplay.y },
    "top-right": {
      x: ROIRectDisplay.x + ROIRectDisplay.width,
      y: ROIRectDisplay.y,
    },
    "bottom-left": {
      x: ROIRectDisplay.x,
      y: ROIRectDisplay.y + ROIRectDisplay.height,
    },
    "bottom-right": {
      x: ROIRectDisplay.x + ROIRectDisplay.width,
      y: ROIRectDisplay.y + ROIRectDisplay.height,
    },
  };
  for (let key in handles) {
    const { x, y } = handles[key];
    if (Math.abs(mouseX - x) < 10 && Math.abs(mouseY - y) < 10) {
      return key;
    }
  }
  return null;
};

//#endregion
```

```
//#region Measure circle tool functions
function getPhysicalDistance(p1, p2) {
  const dx = p2.x - p1.x;
  const dy = p2.y - p1.y;
  return Math.sqrt(dx * dx + dy * dy); // Assuming units are in mm
}

const drawMeasureCircle = (
  ctx,
  p1,
  p2,
  dashed = false,
  isHovered = false,
  isResized = false
) => {
  const dp1 = translateWorldToDisplay(
    p1,
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth
  );
  const dp2 = translateWorldToDisplay(
    p2,
    canvasWidth,
    canvasHeight,
    translateX,
```

```
  translateY,
  positionRect,
  bodyHeight,
  bodyWidth
);

const centerX = (dp1.x + dp2.x) / 2;
const centerY = (dp1.y + dp2.y) / 2;
const dx = dp2.x - dp1.x;
const dy = dp2.y - dp1.y;
const radius = Math.sqrt(dx * dx + dy * dy) / 2;

const diameterMM = getPhysicalDistance(p1, p2);

const areaMM2 = Math.PI * Math.pow(diameterMM / 2, 2);

if (isHovered || isResized) {
  ctx.strokeStyle = `rgb(101,185,145)`;
} else {
  ctx.strokeStyle = `rgb(64,130,172)`;
}

if (isHovered) {
  ctx.font = "bold 14px Arial";
  ctx.fillStyle = "red";
  ctx.textAlign = "center";
  ctx.textBaseline = "middle";
  ctx.fillText("✕", centerX, centerY);
}

ctx.strokeRect(dp1.x - 3, dp1.y - 3, 6, 6);
```

```
ctx.strokeRect(dp2.x - 3, dp2.y - 3, 6, 6);

if (dashed) {
  ctx.setLineDash([4, 4]);
} else {
  ctx.setLineDash([]);
}

ctx.beginPath();
ctx.arc(centerX, centerY, radius, 0, 2 * Math.PI);

ctx.lineWidth = 2;
ctx.stroke();
// Draw the diameter
ctx.beginPath();
ctx.moveTo(dp1.x, dp1.y);
ctx.lineTo(dp2.x, dp2.y);
ctx.stroke();

ctx.setLineDash([]);

ctx.font = "14px Arial";
//ctx.fillStyle = `rgb(40,114,79)`;
ctx.fillStyle = "yellow";
ctx.textAlign = "center";
ctx.textBaseline = "bottom";
ctx.fillText(
  `ϕ ${diameterMM.toFixed(2)} mm, ${areaMM2.toFixed(2)} mm²`,
  centerX,
  centerY - radius - 10
);
```

```
};

//#endregion

//#region Measure length tool functions
function pointToSegmentDistance(p, p1, p2) {
  const A = p.x - p1.x;
  const B = p.y - p1.y;
  const C = p2.x - p1.x;
  const D = p2.y - p1.y;

  const dot = A * C + B * D;
  const lenSq = C * C + D * D;
  const param = lenSq !== 0 ? dot / lenSq : -1;

  let xx, yy;
  if (param < 0) {
    xx = p1.x;
    yy = p1.y;
  } else if (param > 1) {
    xx = p2.x;
    yy = p2.y;
  } else {
    xx = p1.x + param * C;
    yy = p1.y + param * D;
  }

  const dx = p.x - xx;
  const dy = p.y - yy;
  return Math.sqrt(dx * dx + dy * dy);
}
```

```
const drawRuler = (
  ctx,
  p1CT,
  p2CT,
  distance,
  dashed = false,
  isHovered = false,
  isResized = false
) => {
  const p1 = translateWorldToDisplay(
    p1CT,
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth
  );

  const p2 = translateWorldToDisplay(
    p2CT,
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth
  );
```

```
//ctx.strokeStyle = "cyan";

if (isHovered || isResized) {
  ctx.strokeStyle = `rgb(101,185,145)`;
} else {
  ctx.strokeStyle = `rgb(64,130,172)`;
}
// ctx.fillStyle = `rgb(101,185,145)`;
ctx.fillStyle = "yellow";
ctx.lineWidth = 2;

// Line
if (dashed) {
  ctx.setLineDash([4, 4]);
} else {
  ctx.setLineDash([]);
}
ctx.beginPath();
ctx.moveTo(p1.x, p1.y);
ctx.lineTo(p2.x, p2.y);
ctx.stroke();

// Squares
ctx.strokeRect(p1.x - 3, p1.y - 3, 6, 6);
ctx.strokeRect(p2.x - 3, p2.y - 3, 6, 6);

// Write the text on the right of the rightmost point // like in Stone webviewer
let textX, textY;
if (p1.x > p2.x) {
  textX = p1.x + 40;
```

```
    textY = p1.y;
  } else {
    textX = p2.x + 40;
    textY = p2.y;
  }

  // Text

  ctx.font = "14px Arial";
  ctx.textAlign = "center";
  ctx.textBaseline = "middle";
  ctx.fillText(`${distance.toFixed(2)} mm`, textX, textY);

  // If hovered, draw a small delete box at midpoint
  if (isHovered) {
    const midX = (p1.x + p2.x) / 2;
    const midY = (p1.y + p2.y) / 2;

    ctx.font = "bold 14px Arial";
    ctx.fillStyle = "red";
    ctx.textAlign = "center";
    ctx.textBaseline = "middle";
    ctx.fillText("✕", midX, midY);
  }
};

//#endregion

//#region Measure angle tool functions
function calculateAngle(p1, p2, p3) {
  const a = Math.atan2(p1.y - p2.y, p1.x - p2.x);
```

```
  const b = Math.atan2(p3.y - p2.y, p3.x - p2.x);
  let angle = ((b - a) * 180) / Math.PI;
  if (angle < 0) angle += 360;
  if (angle > 180) angle = 360 - angle; // Acute angle
  return angle;
}

const drawProtractor = (
  ctx,
  p1CT,
  p2CT,
  p3CT,
  showAngle = true,
  isHovered,
  isResized = false
) => {
  if (isHovered || isResized) {
    ctx.strokeStyle = `rgb(101,185,145)`;
  } else {
    ctx.strokeStyle = `rgb(64,130,172)`;
  }
  const p1 = translateWorldToDisplay(
    p1CT,
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth
  );
```

```
const p2 = translateWorldToDisplay(
  p2CT,
  canvasWidth,
  canvasHeight,
  translateX,
  translateY,
  positionRect,
  bodyHeight,
  bodyWidth
);
const p3 = translateWorldToDisplay(
  p3CT,
  canvasWidth,
  canvasHeight,
  translateX,
  translateY,
  positionRect,
  bodyHeight,
  bodyWidth
);

ctx.lineWidth = 2;
ctx.setLineDash([]);

// Lines
ctx.beginPath();
ctx.moveTo(p2.x, p2.y);
ctx.lineTo(p1.x, p1.y);
ctx.moveTo(p2.x, p2.y);
ctx.lineTo(p3.x, p3.y);
ctx.stroke();
```

```
// Points
ctx.fillStyle = "yellow";
[p1, p2, p3].forEach((p) => {
  ctx.beginPath();
  ctx.strokeRect(p.x - 3, p.y - 3, 6, 6);
  ctx.stroke();
});

// Angle text
if (showAngle) {
  const angle = calculateAngle(p1CT, p2CT, p3CT);
  ctx.font = "14px Arial";
  ctx.textAlign = "center";
  ctx.textBaseline = "middle";
  ctx.fillText(`${angle.toFixed(2)}°`, p2.x - 10, p2.y - 10);
  const angle1 = Math.atan2(p1.y - p2.y, p1.x - p2.x);
  const angle2 = Math.atan2(p3.y - p2.y, p3.x - p2.x);
  let startAngle = angle1;
  let endAngle = angle2;
  let anticlockwise = false;

  let delta = endAngle - startAngle;
  if (delta < 0) delta += 2 * Math.PI;
  if (delta > Math.PI) {
    anticlockwise = true;
  }
  ctx.beginPath();
  ctx.arc(p2.x, p2.y, 20, startAngle, endAngle, anticlockwise);
  ctx.stroke();
}
```

```
  if (isHovered) {
    const p2Disp = translateWorldToDisplay(
      p2CT,
      canvasWidth,
      canvasHeight,
      translateX,
      translateY,
      positionRect,
      bodyHeight,
      bodyWidth
    );
    ctx.font = "bold 14px Arial";
    ctx.fillStyle = "red";
    ctx.textAlign = "center";
    ctx.textBaseline = "middle";
    ctx.fillText("✕", p2Disp.x, p2Disp.y);
  }
};

const drawProtractors = (ctx, anglePointsCT) => {
  anglePointsCT.forEach(([p1, p2, p3], i) => {
    drawProtractor(
      ctx,
      p1,
      p2,
      p3,
      true,
      i === hoveredProtractorIndex,
      i === moveProtractorArmInfo?.index || i === resizeProtractorInfo?.index
    );
```

```
  });
};

const drawProtractorInProgress = (ctx, currentAnglePoints, currentMouse) => {
  if (currentAnglePoints.length === 1 && currentMouse) {
    // Draw line from p1 to mouse
    const p1 = translateWorldToDisplay(
      currentAnglePoints[0],
      canvasWidth,
      canvasHeight,
      translateX,
      translateY,
      positionRect,
      bodyHeight,
      bodyWidth
    );
    const p2 = translateWorldToDisplay(
      currentMouse,
      canvasWidth,
      canvasHeight,
      translateX,
      translateY,
      positionRect,
      bodyHeight,
      bodyWidth
    );

    ctx.strokeStyle = `rgb(64,130,172)`;
    ctx.lineWidth = 1;
    ctx.setLineDash([4, 2]);
    ctx.beginPath();
```

```
    ctx.moveTo(p1.x, p1.y);
    ctx.lineTo(p2.x, p2.y);
    ctx.stroke();
    ctx.setLineDash([]);

    [p1, p2].forEach((p) => {
      ctx.beginPath();
      ctx.arc(p.x, p.y, 3, 0, 2 * Math.PI);
      ctx.stroke();
    });
  }

  if (currentAnglePoints.length === 2 && currentMouse) {
    drawProtractor(
      ctx,
      currentAnglePoints[0],
      currentAnglePoints[1],
      currentMouse,
      true
    );
  }
};

//#endregion

// #region Rectangle Probe tool functions

function toVoxelIndex(coord, origin, spacing) {
  return Math.floor((coord - origin) / spacing);
}
```

```
function computeRectangleStats(p1, p2) {
  // Finding the index of x and Y
  const p1x = toVoxelIndex(p1.x, minX, pixelSpacing[0]);
  const p1y = toVoxelIndex(p1.y, minY, pixelSpacing[1]);
  const p2x = toVoxelIndex(p2.x, minX, pixelSpacing[0]);
  const p2y = toVoxelIndex(p2.y, minY, pixelSpacing[1]);
  const x1 = Math.min(p1x, p2x);
  const x2 = Math.max(p1x, p2x);
  const y1 = Math.min(p1y, p2y);
  const y2 = Math.max(p1y, p2y);
  let values = [];
  for (let y = y1; y <= y2; y++) {
    for (let x = x1; x <= x2; x++) {
      const hu = HUValues[y]?.[x]?.[currentZSlice];
      if (hu !== undefined) {
        values.push(hu);
      }
    }
  }
  const n = values.length;
  const mean = values.reduce((a, b) => a + b, 0) / n || 0;
  const stdDev =
    Math.sqrt(values.reduce((sum, val) => sum + (val - mean) ** 2, 0) / n) ||
    0;
  const area = Math.abs(p2.x - p1.x) * Math.abs(p2.y - p1.y); // In mm2 CT coordinates
  return { mean, stdDev, area };
}

const drawRectangle = (
  ctx,
  p1,
```

```
  p2,
  area,
  mean,
  stdDev,
  isDashed,
  isHovered,
  isResized
) => {
  const dp1 = translateWorldToDisplay(
    p1,
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth
  );

  const dp2 = translateWorldToDisplay(
    p2,
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth
  );
  if (isHovered || isResized) {
    ctx.strokeStyle = `rgb(101,185,145)`;
```

```
} else {
  ctx.strokeStyle = `rgb(64,130,172)`;
}
// Squares

ctx.strokeRect(dp1.x - 3, dp1.y - 3, 6, 6);
ctx.strokeRect(dp2.x - 3, dp2.y - 3, 6, 6);

const x = Math.min(dp1.x, dp2.x);
const y = Math.min(dp1.y, dp2.y);
const width = Math.abs(dp1.x - dp2.x);
const height = Math.abs(dp1.y - dp2.y);
if (isDashed) {
  ctx.setLineDash([4, 4]);
} else {
  ctx.setLineDash([]);
}
ctx.lineWidth = 2;

ctx.strokeRect(x, y, width, height);
// Text display
ctx.font = "14px Arial";
ctx.fillStyle = "yellow";
ctx.textAlign = "left";
const statsLines = [
  `Area: ${area.toFixed(2)} mm²`,
  `Mean: ${mean.toFixed(2)}`,
  `StdDev: ${stdDev.toFixed(2)}`,
];

statsLines.forEach((line, i) => {
```

```
    ctx.fillText(line, x + width + 10, y + height / 2 + i * 16); // 16px line spacing
  });
  if (isHovered) {
    const midX = (dp1.x + dp2.x) / 2;
    const midY = (dp1.y + dp2.y) / 2;

    ctx.font = "bold 14px Arial";
    ctx.fillStyle = "red";
    ctx.textAlign = "center";
    ctx.textBaseline = "middle";
    ctx.fillText("⨯", midX, midY);
  }
};

//#endregion

//#region Ellipse probe tool functions

function checkIfPointNearLine(p1, p2, x, y, threshold = 3) {
  const A = x - p1.x;
  const B = y - p1.y;
  const C = p2.x - p1.x;
  const D = p2.y - p1.y;

  const dot = A * C + B * D;
  const lenSq = C * C + D * D;
  let param = -1;
  if (lenSq !== 0) param = dot / lenSq;

  let xx, yy;
  if (param < 0) {
```

```
    xx = p1.x;
    yy = p1.y;
  } else if (param > 1) {
    xx = p2.x;
    yy = p2.y;
  } else {
    xx = p1.x + param * C;
    yy = p1.y + param * D;
  }

  const dx = x - xx;
  const dy = y - yy;
  return Math.sqrt(dx * dx + dy * dy) < threshold;
}

const drawEllipse = (
  ctx,
  p1,
  p2,
  area,
  mean,
  stdDev,
  isDashed,
  isHovered,
  isResized
) => {
  const dp1 = translateWorldToDisplay(
    p1,
    canvasWidth,
    canvasHeight,
    translateX,
```

```
  translateY,
  positionRect,
  bodyHeight,
  bodyWidth
);
const dp2 = translateWorldToDisplay(
  p2,
  canvasWidth,
  canvasHeight,
  translateX,
  translateY,
  positionRect,
  bodyHeight,
  bodyWidth
);

const centerX = (dp1.x + dp2.x) / 2;
const centerY = (dp1.y + dp2.y) / 2;
const radiusX = Math.abs(dp2.x - dp1.x) / 2;
const radiusY = Math.abs(dp2.y - dp1.y) / 2;

ctx.beginPath();
ctx.setLineDash(isDashed ? [4, 4] : []);
ctx.lineWidth = 2;
if (isHovered || isResized) {
  ctx.strokeStyle = `rgb(101,185,145)`;
} else {
  ctx.strokeStyle = `rgb(64,130,172)`;
}
ctx.ellipse(centerX, centerY, radiusX, radiusY, 0, 0, 2 * Math.PI);
ctx.stroke();
```

```
  // Draw handles
  ctx.strokeRect(dp1.x - 3, dp1.y - 3, 6, 6);
  ctx.strokeRect(dp2.x - 3, dp2.y - 3, 6, 6);

  // Display stats
  ctx.font = "14px Arial";
  ctx.fillStyle = "yellow";
  ctx.textAlign = "left";
  const statsLines = [
    `Area: ${area.toFixed(2)} mm²`,
    `Mean: ${mean.toFixed(2)}`,
    `StdDev: ${stdDev.toFixed(2)}`,
  ];
  statsLines.forEach((line, i) => {
    ctx.fillText(line, centerX + radiusX + 10, centerY + i * 16);
  });
  if (isHovered) {
    const midX = (dp1.x + dp2.x) / 2;
    const midY = (dp1.y + dp2.y) / 2;

    ctx.font = "bold 14px Arial";
    ctx.fillStyle = "red";
    ctx.textAlign = "center";
    ctx.textBaseline = "middle";
    ctx.fillText("✕", midX, midY);
  }
};

function computeEllipseStats(p1, p2) {
  const p1x = toVoxelIndex(p1.x, minX, pixelSpacing[0]);
```

```
const p1y = toVoxelIndex(p1.y, minY, pixelSpacing[1]);
const p2x = toVoxelIndex(p2.x, minX, pixelSpacing[0]);
const p2y = toVoxelIndex(p2.y, minY, pixelSpacing[1]);

const x1 = Math.min(p1x, p2x);
const x2 = Math.max(p1x, p2x);
const y1 = Math.min(p1y, p2y);
const y2 = Math.max(p1y, p2y);

const cx = (x1 + x2) / 2;
const cy = (y1 + y2) / 2;
const rx = (x2 - x1) / 2;
const ry = (y2 - y1) / 2;

let values = [];
for (let y = y1; y <= y2; y++) {
  for (let x = x1; x <= x2; x++) {
    const dx = x - cx;
    const dy = y - cy;
    if ((dx * dx) / (rx * rx) + (dy * dy) / (ry * ry) <= 1) {
      const hu = HUValues[y]?.[x]?.[currentZSlice];
      if (hu !== undefined) values.push(hu);
    }
  }
}

const n = values.length;
const mean = values.reduce((a, b) => a + b, 0) / n || 0;
const stdDev =
  Math.sqrt(values.reduce((sum, val) => sum + (val - mean) ** 2, 0) / n) ||
  0;
```

```
  const area =
    (((Math.PI * Math.abs(p2.x - p1.x)) / 2) * Math.abs(p2.y - p1.y)) / 2;
  return { mean, stdDev, area };
}

//#endregion

//#region Pixel probe tool functions
const drawPixel = (ctx, p1CT, hu, isHovered, isMoved) => {
  const p1 = translateWorldToDisplay(
    p1CT,
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth
  );

  if (isHovered || isMoved) {
    ctx.strokeStyle = `rgb(101,185,145)`;
  } else {
    ctx.strokeStyle = `rgb(64,130,172)`;
  }

  ctx.fillStyle = "yellow";
  ctx.lineWidth = 2;

  // Squares
  ctx.strokeRect(p1.x - 5, p1.y - 5, 5, 5);
```

```
  ctx.strokeRect(p1.x, p1.y - 5, 5, 5);
  ctx.strokeRect(p1.x - 5, p1.y, 5, 5);
  ctx.strokeRect(p1.x, p1.y, 5, 5);

  // Text

  ctx.font = "14px Arial";
  ctx.textAlign = "left";
  ctx.textBaseline = "middle";
  ctx.fillText(
    `(${p1CT.x.toFixed(2)}, ${p1CT.y.toFixed(2)}, ${ZCoordinate.toFixed(
      2
    )}) ${hu}`,
    p1.x + 10,
    p1.y
  );

  // If hovered, draw a small delete box at midpoint
  if (isHovered) {
    ctx.font = "bold 14px Arial";
    ctx.fillStyle = "red";
    ctx.textAlign = "center";
    ctx.textBaseline = "middle";
    ctx.fillText("✕", p1.x, p1.y);
  }
};

const drawTextAnnotation = (
  ctx,
  headCT,
  tailCT,
```

```
  text,
  isDashed,
  isHovered,
  isResized
) => {
  const tail = translateWorldToDisplay(
    tailCT,
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth
  ); // from annotation.tail
  const head = translateWorldToDisplay(
    headCT,
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth
  ); // from annotation.head

  if (isDashed) {
    // Draw dashed line
    ctx.setLineDash([5, 5]);
  }
  ctx.font = "bold 14px Arial";
```

```
// ⨯ Icon at midpoint if hovered

if (isResized || isHovered) {
  ctx.strokeStyle = `rgb(101,185,145)`;
  ctx.fillStyle = `rgb(101,185,145)`;
} else {
  ctx.strokeStyle = `rgb(64,130,172)`;
  ctx.fillStyle = `rgb(64,130,172)`;
}
// Draw line with arrowhead
ctx.beginPath();
ctx.moveTo(tail.x, tail.y);
ctx.lineTo(head.x, head.y);
ctx.stroke();
ctx.strokeRect(tail.x - 3, tail.y - 3, 6, 6);

drawArrowhead(ctx, tail, head);
if (isHovered) {
  const midX = (tail.x + head.x) / 2;
  const midY = (tail.y + head.y) / 2;

  ctx.fillStyle = "red";
  // ctx.textAlign = "center";
  // ctx.textBaseline = "middle";
  ctx.fillText("⨯", midX, midY);
}

// Draw text
if (text != null) {
  ctx.fillStyle = "yellow";
  ctx.font = "14px Arial";
```

```
    ctx.fillText(text, tail.x + 15, tail.y);
  }
  ctx.setLineDash([]);
};

const findHU = (ctPoint) => {
  return HUValues[toVoxelIndex(ctPoint.y, minY, pixelSpacing[1])][
    toVoxelIndex(ctPoint.x, minX, pixelSpacing[0])
  ][currentZSlice];
};
//#endregion

//#region Text Annotations tool functions
function drawArrowhead(ctx, from, to) {
  const angle = Math.atan2(to.y - from.y, to.x - from.x);
  const size = 8;
  ctx.beginPath();
  ctx.moveTo(to.x, to.y);
  ctx.lineTo(
    to.x - size * Math.cos(angle - Math.PI / 6),
    to.y - size * Math.sin(angle - Math.PI / 6)
  );
  ctx.lineTo(
    to.x - size * Math.cos(angle + Math.PI / 6),
    to.y - size * Math.sin(angle + Math.PI / 6)
  );
  ctx.closePath();
  ctx.fill();
}
//#endregion
```

```
useEffect(() => {
  setCanvasHeight(transversalCanvasDimensions.height);
  setCanvasWidth(transversalCanvasDimensions.width);
}, [transversalCanvasDimensions.height, transversalCanvasDimensions.width]);

//Change current Z based on current Index
useEffect(() => {
  setZCoordinate((prevZ) => zList[currentZSlice]);
}, [currentZSlice]);

const transversalImgDataRef = useRef(null);
const renderCanvas = () => {
  const canvas = canvasRef.current;
  const ctx = canvas.getContext("2d");
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  const transversalImgData = ctx.createImageData(
    originalWidth,
    originalHeight
  );

  //Create image Data takes parameters as (width, height)
  for (let i = 0; i < originalWidth; i++) {
    // Y
    for (let j = 0; j < originalHeight; j++) {
      //X
      const value8bit = images[j][i][currentZSlice];
      const index = j * originalWidth + i;
      // Assign the value to both red and green channels (since it's grayscale)
      transversalImgData.data[index * 4] = value8bit; // Red channel
      transversalImgData.data[index * 4 + 1] = value8bit; // Green channel
      transversalImgData.data[index * 4 + 2] = value8bit; // Blue channel
```

```
    transversalImgData.data[index * 4 + 3] = 255; // Alpha channel (fully opaque)
   }
 }

transversalImgDataRef.current = transversalImgData;

const newWidth = canvasWidth; //391;
const newHeight = canvasHeight; //233;
const stretchedImgData = new ImageData(newWidth, newHeight);
// Interpolation code
for (let y = 0; y < newHeight; y++) {
  for (let x = 0; x < newWidth; x++) {
    // Calculate corresponding position in original image
    const originalX = (x / newWidth) * (originalWidth - 1);
    const originalY = (y / newHeight) * (originalHeight - 1);

    // Get the integer coordinates of the four nearest pixels
    const x0 = Math.floor(originalX);
    const x1 = Math.ceil(originalX);
    const y0 = Math.floor(originalY);
    const y1 = Math.ceil(originalY);

    // Get the fractional parts for interpolation
    const xf = originalX - x0;
    const yf = originalY - y0;

    // Get the color values of the four nearest pixels
    const topLeftIndex = (y0 * originalWidth + x0) * 4;
    const topRightIndex = (y0 * originalWidth + x1) * 4;
    const bottomLeftIndex = (y1 * originalWidth + x0) * 4;
    const bottomRightIndex = (y1 * originalWidth + x1) * 4;
```

```
    // Perform bilinear interpolation for each color channel
    for (let i = 0; i < 4; i++) {
      const topInterpolated =
        transversalImgData.data[topLeftIndex + i] * (1 - xf) +
        transversalImgData.data[topRightIndex + i] * xf;
      const bottomInterpolated =
        transversalImgData.data[bottomLeftIndex + i] * (1 - xf) +
        transversalImgData.data[bottomRightIndex + i] * xf;
      const interpolatedValue =
        topInterpolated * (1 - yf) + bottomInterpolated * yf;
      stretchedImgData.data[(y * newWidth + x) * 4 + i] = interpolatedValue;
    }
  }
}
ctx.putImageData(stretchedImgData, translateX, translateY);

if (
  isRulerEnabled ||
  isProtractorEnabled ||
  isMeasureCircleEnabled ||
  isRectangleProbeEnabled ||
  isEllipseProbeEnabled ||
  isPixelProbeEnabled ||
  isTextAnnotationEnabled ||
  isDeleteAnnotationEnabled
) {
  // Render all the annotations

  // Draw all completed rulers
  if (rulerPointsCT[ZCoordinate]) {
```

```
  rulerPointsCT[ZCoordinate].forEach((ruler, index) => {
    drawRuler(
      ctx,
      ruler.start,
      ruler.end,
      ruler.distance,
      false,
      index === hoveredRulerIndex,
      index === resizeRulerIndex?.index || index === movedRulerIndex
    );
  });
}

// Draw the ruler that is in progress
if (isRulerDrag && rulerStartPoint && currentMouse) {
  const distance = Math.sqrt(
    (currentMouse.x - rulerStartPoint.x) ** 2 +
      (currentMouse.y - rulerStartPoint.y)
  );
  drawRuler(ctx, rulerStartPoint, currentMouse, distance, true);
}

// Draw the completed angles

if (anglePointsCT[ZCoordinate]) {
  drawProtractors(ctx, anglePointsCT[ZCoordinate]);
}
drawProtractorInProgress(ctx, currentAnglePoints, currentMouse);

// Draw the finalised circles
if (circlePointsCT[ZCoordinate]) {
```

```
  circlePointsCT[ZCoordinate].forEach(({ p1, p2 }, i) => {
    drawMeasureCircle(
      ctx,
      p1,
      p2,
      false,
      i === hoveredCircleIndex,
      i === movedCircleIndex || i === resizeCircleIndex?.index
    ); // We can delete the circle only if it is completed
  });
}

// Draw the circles that are in progress
if (currentCircleStart && currentMouse) {
  drawMeasureCircle(ctx, currentCircleStart, currentMouse, true);
}

if (rectangleProbesCT[ZCoordinate]) {
  rectangleProbesCT[ZCoordinate].forEach(
    ({ p1, p2, area, mean, stdDev }, i) => {
      drawRectangle(
        ctx,
        p1,
        p2,
        area,
        mean,
        stdDev,
        false,
        i === hoveredRectangleIndex,
        i === movedRectangleIndex || i === resizeRectangleIndex?.index
      );
```

```
    }
  );
}

if (currentRectangleStart && currentMouse) {
  const stat = computeRectangleStats(currentRectangleStart, currentMouse);
  drawRectangle(
    ctx,
    currentRectangleStart,
    currentMouse,
    stat.area,
    stat.mean,
    stat.stdDev,
    true
  );
}
if (ellipseProbesCT[ZCoordinate]) {
  ellipseProbesCT[ZCoordinate].forEach(
    ({ p1, p2, area, mean, stdDev }, i) => {
      drawEllipse(
        ctx,
        p1,
        p2,
        area,
        mean,
        stdDev,
        false,
        i === hoveredEllipseIndex,
        i === movedEllipseIndex || i === resizeEllipseIndex?.index
      );
    }
```

```
  );
}
if (currentEllipseStart && currentMouse) {
  const stat = computeEllipseStats(currentEllipseStart, currentMouse);
  drawEllipse(
    ctx,
    currentEllipseStart,
    currentMouse,
    stat.area,
    stat.mean,
    stat.stdDev,
    true
  );
}
if (pixelProbesCT[ZCoordinate]) {
  pixelProbesCT[ZCoordinate].forEach(({ point, hu }, i) => {
    drawPixel(
      ctx,
      point,
      hu,
      i === hoveredPixelIndex,
      !isMovingPixelProbe && i === movedPixelProbeIndex // make the colur green while hovering
    );
  });
}

if (isMovingPixelProbe && previewPixelPoint) {
  drawPixel(
    ctx,
    previewPixelPoint?.point,
    previewPixelPoint?.hu,
```

```
        false,
        true // make the colour green while moving the pixel
      );
    }

    if (textAnnotations[ZCoordinate]) {
      textAnnotations[ZCoordinate].forEach((annotation, index) => {
        drawTextAnnotation(
          ctx,
          annotation.head,
          annotation.tail,
          annotation.text,
          false,
          index === hoveredTextAnnotationIndex,
          index === movedTextAnnotationIndex ||
            index === resizedTextAnnotationsIndex?.index
        );
      });
    }
    // Draw confirmation buttons for pending annotation
if (isConfirmingText && pendingTextAnnotation) {
  // Draw the annotation preview first
  drawTextAnnotation(
    ctx,
    pendingTextAnnotation.head,
    pendingTextAnnotation.tail,
    pendingTextAnnotation.text,
    false,
    false,
    false
  );
```

```
  const tail = translateWorldToDisplay(
    pendingTextAnnotation.tail,
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth
  );

  ctx.font = "bold 18px Arial";
  ctx.textAlign = "center";
  ctx.textBaseline = "middle";

  // ✔ confirm
  ctx.fillStyle = "lime";
  ctx.fillText("✔", tail.x + 20, tail.y - 20);

  // ✕ cancel
  // ctx.fillStyle = "red";
  // ctx.fillText("✕", tail.x + 40, tail.y - 20);
}
if (isConfirmingText && pendingEditAnnotation) {

  const ann = textAnnotations[ZCoordinate][pendingEditAnnotation.index];

  const tail = translateWorldToDisplay(
    ann.tail,
    canvasWidth,
```

```
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth
   );

   ctx.font = "bold 18px Arial";
   ctx.textAlign = "center";
   ctx.textBaseline = "middle";

   // ✔ confirm
   ctx.fillStyle = "lime";
   ctx.fillText("✔", tail.x + 20, tail.y - 20);

   // ✕ cancel
   ctx.fillStyle = "red";
   ctx.fillText("✕", tail.x + 40, tail.y - 20);
  }
  if (isConfirmingText && pendingMoveAnnotation) {
   const ann = textAnnotations[ZCoordinate][pendingMoveAnnotation.index];

   const tail = translateWorldToDisplay(
    ann.tail,
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
```

```
    bodyWidth
  );

  ctx.font = "bold 18px Arial";
  ctx.textAlign = "center";
  ctx.textBaseline = "middle";

  ctx.fillStyle = "lime";
  ctx.fillText("✔", tail.x + 20, tail.y - 20);

  ctx.fillStyle = "red";
  ctx.fillText("✕", tail.x + 40, tail.y - 20);
}



    let previewHead = null;
    let previewTail = null;

    if (resizingTextPointType === "head") {
      previewHead = currentMouse;
      previewTail = currentTextTail;
    } else {
      previewHead = currentTextTail;
      previewTail = currentMouse;
    }

    if (isTextAnnotationEnabled && currentTextTail && currentMouse) {
      drawTextAnnotation(ctx, previewHead, previewTail, null, true, false);
    }
  }
```

```
// Drawing the IEC 61217 annotations

// Draw horizontal line (e.g., A–P or H–F)
ctx.beginPath();
ctx.strokeStyle = "#FFFF00";
ctx.setLineDash([1, 1]);
ctx.lineWidth = 1;
ctx.moveTo(0, middleDisplayPoint.y);
ctx.lineTo(canvasWidth, middleDisplayPoint.y);
ctx.stroke();
ctx.closePath();

// Draw vertical line (e.g., R–L)
ctx.beginPath();
ctx.strokeStyle = "green";
ctx.setLineDash([1, 1]);
ctx.lineWidth = 1;
ctx.moveTo(middleDisplayPoint.x, 0);
ctx.lineTo(middleDisplayPoint.x, canvasHeight);
ctx.stroke();
ctx.closePath();

ctx.font = "14px Arial";
ctx.fillStyle = "#FFFF00"; // bright yellow for horizontal labels

// Horizontal line: R (left), L (right)
ctx.fillText("R", 5, middleDisplayPoint.y - 5); // small padding from the intersection
ctx.fillText("L", canvasWidth - 25, middleDisplayPoint.y - 5);

// Vertical line: A (top), P (bottom)
```

```
ctx.fillStyle = "green"; // bright green for vertical labels
ctx.fillText("A", middleDisplayPoint.x + 5, 15); // small offset right and down from top
ctx.fillText("P", middleDisplayPoint.x + 5, canvasHeight - 5);

ctx.setLineDash([]);

// Draw the complete polygons only of the selected structure for current index
const selectedStructureSet = structureSets.find(
  (set) => set.structureSetUID === selectedStructureSetUID
);

if (selectedStructureSet) {
  const selectedStructures = selectedStructureSet.structuresList.filter(
    (structure) => displayStructuresUIDs.includes(structure.structureID)
  );
  selectedStructures.map((selectedStructure) => {
    // Draw all displayed structures (checkboxed) one by one
    const allPolygons = [
      ...(selectedStructure.polygonsList[ZCoordinate] || []),
    ];

    if (allPolygons.length > 0) {
      ctx.strokeStyle = `rgb(${selectedStructure.Color.split("\\")
        .map((v) => parseInt(v, 10))
        .join(",")})`;

      ctx.lineWidth = 2;
      allPolygons.forEach((polygon) => {
        ctx.beginPath();
        polygon.forEach((point, index) => {
          const displayPoint = translateWorldToDisplay(
```

```
          point,
          canvasWidth,
          canvasHeight,
          translateX,
          translateY,
          positionRect,
          bodyHeight,
          bodyWidth
        );

        if (index === 0) {
          ctx.moveTo(displayPoint.x, displayPoint.y);
        } else {
          ctx.lineTo(displayPoint.x, displayPoint.y);
        }
      });
      ctx.closePath();
      ctx.stroke();
    });
  }
});
// Draw points in open polygon

if (points[ZCoordinate] && points[ZCoordinate].length > 0) {
  points[ZCoordinate].forEach((point, index) => {
    const displayPoint = translateWorldToDisplay(
      point,
      canvasWidth,
      canvasHeight,
      translateX,
      translateY,
```

```
    positionRect,
    bodyHeight,
    bodyWidth
  );

  ctx.fillStyle = "red"; // Color for points
  ctx.beginPath();

  ctx.arc(displayPoint.x, displayPoint.y, 3, 0, Math.PI * 2); // Draw point as a circle

  ctx.fill();
});

//Draw lines in open polygon using spline - in case of Point

if (splinePoints.length > 0) {
  ctx.strokeStyle = "aqua";

  ctx.lineWidth = 2;
  ctx.beginPath();

  splinePoints.forEach((point, index) => {
    const displayPoint = translateWorldToDisplay(
      point,
      canvasWidth,
      canvasHeight,
      translateX,
      translateY,
      positionRect,
      bodyHeight,
      bodyWidth
```

```
    );

    if (index === 0) {
      ctx.moveTo(displayPoint.x, displayPoint.y);
    } else {
      ctx.lineTo(displayPoint.x, displayPoint.y);
    }
  });

  ctx.stroke();
 } else if (points[ZCoordinate] && points[ZCoordinate].length > 0) {
  // In case of freepoint, directly draw lines between the selected points
  ctx.strokeStyle = "aqua";
  ctx.lineWidth = 2;

  ctx.beginPath();

  points[ZCoordinate].forEach((point, index) => {
    const displayPoint = translateWorldToDisplay(
      point,
      canvasWidth,
      canvasHeight,
      translateX,
      translateY,
      positionRect,
      bodyHeight,
      bodyWidth
    );

    if (index === 0) {
      ctx.moveTo(displayPoint.x, displayPoint.y);
```

```
      } else {
        ctx.lineTo(displayPoint.x, displayPoint.y);
      }
    });

    ctx.stroke();
  }
 }

 if (openAutoSegDialog && ROI3dCT) {
  drawROI(ctx);
 }
 if (isBrushing) {
  drawPolygons(ctx, brushPolygons, false);
 }

 if (isErasing) {
  drawPolygons(ctx, eraserPolygons, true);
 }
}
// Draw the crosshair horizontal and vertical lines
if (CTCoordinate && isCrosshairEnabled) {
 const mousePointCrosshair = translateWorldToDisplay(
  CTCoordinate,
  canvasWidth,
  canvasHeight,
  translateX,
  translateY,
  positionRect,
  bodyHeight,
  bodyWidth
```

```
  );
  ctx.beginPath();
  ctx.strokeStyle = "gray"; // Crosshair color

  ctx.setLineDash([5, 5]); // Dashed line: [5px dash, 5px gap]
  ctx.lineWidth = 1;

  // Horizontal line
  ctx.moveTo(0, mousePointCrosshair.y); // Start at the left edge
  ctx.lineTo(canvasWidth, mousePointCrosshair.y); // End at the right edge

  // Vertical line
  ctx.moveTo(mousePointCrosshair.x, 0); // Start at the top edge
  ctx.lineTo(mousePointCrosshair.x, canvasHeight); // End at the bottom edge

  ctx.stroke();
  ctx.closePath();
}
ctx.setLineDash([]);

// Display the points of autosegmentation
// if (pointsToDisplayAuto) {
//   for (let i = 0; i < pointsToDisplayAuto.length; i++) {
//     const point = pointsToDisplayAuto[i];
//     const pointDisplay = translateWorldToDisplay(
//       point,
//       canvasWidth,
//       canvasHeight,
//       translateX,
//       translateY,
//       positionRect,
```

```
    //       bodyHeight,
    //       bodyWidth
    //     );
    //     ctx.beginPath();
    //     ctx.arc(pointDisplay.x, pointDisplay.y, 5, 0, 2 * Math.PI); // 5px radius dot
    //     ctx.fillStyle = "rgba(255, 0, 0, 0.1)"; // Red with 40% opacity
    //     ctx.fill();
    //   }
    // }
  };
useEffect(() => {
  const fetchAnnotations = async () => {
    try {
      const response = await fetch(
        `http://localhost:5000/api/annotations/${ZCoordinate}`
      );

      const data = await response.json();

      const formatted = data.map(row => ({
        id: row.id,
        head: new Point(row.head_x, row.head_y),
        tail: new Point(row.tail_x, row.tail_y),
        text: row.comment
      }));

      setTextAnnotations(prev => ({
        ...prev,
        [ZCoordinate]: formatted
      }));
```

```
    } catch (err) {
      console.error("Failed to load annotations:", err);
    }
  };

  if (ZCoordinate !== undefined) {
    fetchAnnotations();
  }

}, [ZCoordinate]);

  useEffect(() => {
    renderCanvas();
  }, [
    pendingTextAnnotation,
isConfirmingText,

    points,
    ZCoordinate,
    contourStyle,
    splinePoints,
    translateX,
    translateY,
    selectedStructureSetUID,
    selectedStructureUID,
    displayStructuresUIDs,
    structureSets,
    canvasHeight,
    canvasWidth,
    isCrosshairEnabled,
    isRulerEnabled,
```

isProtractorEnabled,

anglePointsCT,

currentAnglePoints,

isMeasureCircleEnabled,

isRectangleProbeEnabled,

isEllipseProbeEnabled,

isPixelProbeEnabled,

isTextAnnotationEnabled,

isDeleteAnnotationEnabled,

textAnnotations,

currentTextTail,

circlePointsCT,

currentCircleStart,

currentRectangleStart,

rectangleProbesCT,

ellipseProbesCT,

pixelProbesCT,

currentEllipseStart,

CTCoordinate?.x,

CTCoordinate?.y,

ROI3dCT?.x,

ROI3dCT?.y,

ROI3dCT?.width,

ROI3dCT?.height,

openAutoSegDialog,

rulerPointsCT,

rulerStartPoint,

currentMouse,

isRulerDrag,

hoveredRulerIndex,

resizeRulerIndex,

movedRulerIndex,

isRulerMoving,

rulerStartMovePoint,

hoveredProtractorIndex,

resizeProtractorInfo,

moveProtractorArmInfo,

lastPhysicalPoint,

draggingType,

hoveredCircleIndex,

movedCircleIndex,

resizeCircleIndex,

circleStartMovePoint,

isCircleMoving,

hoveredRectangleIndex,

movedRectangleIndex,

resizeRectangleIndex,

rectangleStartMovePoint,

isRectangleMoving,

hoveredEllipseIndex,

movedEllipseIndex,

resizeEllipseIndex,

ellipseStartMovePoint,

isEllipseMoving,

hoveredPixelIndex,

movedPixelProbeIndex,

isMovingPixelProbe,

previewPixelPoint,

hoveredTextAnnotationIndex,

resizedTextAnnotationsIndex,

movedTextAnnotationIndex,

```
    textStartMovePoint,
    isTextMoving,
    resizingTextPointType,
    middleDisplayPoint,
    hoveredRlAp,
    draggedRlAp,
    brushPolygons,
    isBrushing,
    isErasing,
    eraserPolygons,
    pointsToDisplayAuto,
  ]);
  /*
If the contour style is changed or the slice Index is changed without completing the polygon,
 then all the incomplete open polygons will be cleared
 */
  // If another structure/structure set is selected before closing the contour, then also the open
points will be cleared


  useEffect(() => {
    const canvas = canvasRef.current;
    const ctx = canvas.getContext("2d");


    const handleWheel = (event) => {
      event.preventDefault();
      const delta = event.deltaY;
      if (event.ctrlKey) {
        //Zoom functionality
        if (delta < 0) {
          // Zoom in
```

```
        setCanvasHeight((canvasHeight) => canvasHeight + ZOOM_STEP);
        setCanvasWidth((canvasWidth) => canvasWidth + ZOOM_STEP);
      } else if (delta > 0) {
        // Zoom out
        setCanvasHeight((canvasHeight) => canvasHeight - ZOOM_STEP);
        setCanvasWidth((canvasWidth) => canvasWidth - ZOOM_STEP);
      }
    } else {
      //Wheel to another slice
      setCurrentZSlice((prevIndex) => {
        let newIndex = prevIndex;
        if (delta > 0 && prevIndex < zSlices - 1) {
          newIndex = prevIndex + 1;
        } else if (delta < 0 && prevIndex > 0) {
          newIndex = prevIndex - 1;
        }
        return Math.min(Math.max(newIndex, 0), zSlices - 1); // Clamp within valid range
      });
    }
  };

  const handleMouseDown = async (event) => {
    onActiveViewClick("transversal");
    const rectBounds = canvas.getBoundingClientRect();
    const mouseX = event.clientX - rectBounds.left;
    const mouseY = event.clientY - rectBounds.top;
    // Handle confirmation click
// ================== TEXT CONFIRMATION MODE ==================
if (isConfirmingText) {
  // ===== EDIT CONFIRM MODE =====
if (pendingEditAnnotation) {
```

```
const ann =
  textAnnotations[ZCoordinate][pendingEditAnnotation.index];

const tail = translateWorldToDisplay(
  ann.tail,
  canvasWidth,
  canvasHeight,
  translateX,
  translateY,
  positionRect,
  bodyHeight,
  bodyWidth
);

const confirmX = tail.x + 20;
const confirmY = tail.y - 20;
const cancelX = tail.x + 40;
const cancelY = tail.y - 20;

if (Math.hypot(mouseX - confirmX, mouseY - confirmY) < 12) {
  // ✔ CONFIRM → SAVE TO DB
  try {
    await fetch(
      `http://localhost:5000/api/annotations/${ann.id}`,
      {
        method: "PUT",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({ comment: ann.text }),
      }
    );
```

```
    } catch (err) {
      console.error("Edit save failed:", err);
    }

    setPendingEditAnnotation(null);
    setIsConfirmingText(false);
    return;
  }

  if (Math.hypot(mouseX - cancelX, mouseY - cancelY) < 12) {
    // ✕ CANCEL → REVERT TEXT
    setTextAnnotations(prev => {
      const updated = [...(prev[ZCoordinate] || [])];
      updated[pendingEditAnnotation.index] = {
        ...updated[pendingEditAnnotation.index],
        text: pendingEditAnnotation.oldText
      };
      return { ...prev, [ZCoordinate]: updated };
    });

    setPendingEditAnnotation(null);
    setIsConfirmingText(false);
    return;
  }

  return; // block everything else
}
if (pendingMoveAnnotation) {
  const ann =
    textAnnotations[ZCoordinate][pendingMoveAnnotation.index];
```

```
const tail = translateWorldToDisplay(
  ann.tail,
  canvasWidth,
  canvasHeight,
  translateX,
  translateY,
  positionRect,
  bodyHeight,
  bodyWidth
);

const confirmX = tail.x + 20;
const confirmY = tail.y - 20;
const cancelX = tail.x + 40;
const cancelY = tail.y - 20;

// ✔ CONFIRM MOVE
if (Math.hypot(mouseX - confirmX, mouseY - confirmY) < 12) {
  try {
    await fetch(
      `http://localhost:5000/api/annotations/${ann.id}/position`,
      {
        method: "PUT",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({
          head_x: ann.head.x,
          head_y: ann.head.y,
          tail_x: ann.tail.x,
          tail_y: ann.tail.y,
        }),
      }
```

```
      );
    } catch (err) {
      console.error("Move save failed", err);
    }

    setPendingMoveAnnotation(null);
    setIsConfirmingText(false);
    setMovedTextAnnotationIndex(null);
    return;
  }

  // ✕ CANCEL MOVE
  if (Math.hypot(mouseX - cancelX, mouseY - cancelY) < 12) {
    setTextAnnotations(prev => {
      const updated = [...prev[ZCoordinate]];
      updated[pendingMoveAnnotation.index] = {
        ...updated[pendingMoveAnnotation.index],
        head: pendingMoveAnnotation.originalHead,
        tail: pendingMoveAnnotation.originalTail,
      };
      return { ...prev, [ZCoordinate]: updated };
    });

    setPendingMoveAnnotation(null);
    setIsConfirmingText(false);
    setMovedTextAnnotationIndex(null);
    return;
  }

  return; // HARD STOP
}
```

```
console.log("CONFIRM MODE CLICKED");


if (!pendingTextAnnotation) return;

const tail = translateWorldToDisplay(
  pendingTextAnnotation.tail,
  canvasWidth,
  canvasHeight,
  translateX,
  translateY,
  positionRect,
  bodyHeight,
  bodyWidth
);

const confirmX = tail.x + 20;
const confirmY = tail.y - 20;

const cancelX = tail.x + 40;
const cancelY = tail.y - 20;

const distanceToConfirm = Math.hypot(mouseX - confirmX, mouseY - confirmY);
const distanceToCancel = Math.hypot(mouseX - cancelX, mouseY - cancelY);

if (distanceToConfirm < 12) {
try {
  const response = await fetch(
    "http://localhost:5000/api/annotations",
    {
```

```
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({
        z_coordinate: ZCoordinate,
        head_x: pendingTextAnnotation.head.x,
        head_y: pendingTextAnnotation.head.y,
        tail_x: pendingTextAnnotation.tail.x,
        tail_y: pendingTextAnnotation.tail.y,
        comment: pendingTextAnnotation.text,
      }),
    }
  );

  const saved = await response.json();

  // Store locally WITH DB id
  setTextAnnotations(prev => {
    const updated = { ...prev };
    if (!updated[ZCoordinate]) updated[ZCoordinate] = [];

    updated[ZCoordinate].push({
      ...pendingTextAnnotation,
      id: saved.id   // important for editing later
    });

    return updated;
  });

} catch (error) {
  console.error("Failed to save annotation:", error);
}
```

```
  setPendingTextAnnotation(null);
  setIsConfirmingText(false);
  setCurrentTextTail(null);
  setActiveTextValue(null);
  return;
}


  if (distanceToCancel < 12) {
    setPendingTextAnnotation(null);
    setIsConfirmingText(false);
    setCurrentTextTail(null);
    setActiveTextValue(null);
    return;  // HARD STOP
  }

  // IMPORTANT: block all other clicks while confirming
  return;
}




      if (hoveredRlAp) {
        setDraggedRlAp(hoveredRlAp);
        canvas.style.cursor = "grabbing";
      }

      if (openAutoSegDialog) {
        const handle = getHandle(mouseX, mouseY);
        if (handle) {
```

```
  setResizing(handle);
  return;
}

const CTRect2D = new Rect(
  ROI3dCT.x,
  ROI3dCT.y,
  ROI3dCT.width,
  ROI3dCT.height
);

const ROIRectDisplay = convertRectWorldToDisplay(
  CTRect2D,
  canvasWidth,
  canvasHeight,
  translateX,
  translateY,
  positionRect,
  bodyHeight,
  bodyWidth
);

// Check if mouse is inside the rectangle for dragging
if (
  mouseX > ROIRectDisplay.x &&
  mouseX < ROIRectDisplay.x + ROIRectDisplay.width &&
  mouseY > ROIRectDisplay.y &&
  mouseY < ROIRectDisplay.y + ROIRectDisplay.height
) {
  setDragging(true);
  setOffset({
```

```
      x: mouseX - ROIRectDisplay.x,
      y: mouseY - ROIRectDisplay.y,
    });
  }
} else if (isRulerEnabled) {
  const ctPoint = translateDisplayToWorld(
    new Point(mouseX, mouseY),
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth
  );

  // Don't start a new ruler if we're finishing a move
  if (isRulerMoving) {
    setIsRulerMoving(false);
    setMovedRulerIndex(null);
    setRulerStartMovePoint(null);
    return;
  }

  if (!isRulerDrag) {
    // Start ruler move if near a line
    if (movedRulerIndex != null) {
      setIsRulerMoving(true);
      setRulerStartMovePoint(ctPoint);
      return;
    }
```

```
  // 👇 only create a new ruler if not resizing/moving
  if (resizeRulerIndex != null) {
    const updatedRulers = [...rulerPointsCT[ZCoordinate]];
    if (resizeRulerIndex.point === "start") {
      setRulerStartPoint(updatedRulers[resizeRulerIndex.index].end);
      setCurrentMouse(ctPoint);
      setIsRulerDrag(true);
    } else if (resizeRulerIndex.point === "end") {
      setRulerStartPoint(updatedRulers[resizeRulerIndex.index].start);
      setCurrentMouse(ctPoint);
      setIsRulerDrag(true);
    }

    updatedRulers.splice(resizeRulerIndex.index, 1);
    setRulerPointsCT((prev) => ({
      ...prev,
      [ZCoordinate]: updatedRulers,
    }));

    setResizeRulerIndex(null);
  } else {
    // Start a new ruler (only when not dragging/moving)
    setRulerStartPoint(ctPoint);
    setCurrentMouse(ctPoint);
    setIsRulerDrag(true);
  }
} else {
  // Finish drawing a ruler
  const distance = Math.sqrt(
    (ctPoint.x - rulerStartPoint.x) ** 2 +
```

```
      (ctPoint.y - rulerStartPoint.y) ** 2
    );
    setRulerPointsCT((prev) => {
     const currentSliceRulers = prev[ZCoordinate] || [];
     return {
      ...prev,
      [ZCoordinate]: [
       ...currentSliceRulers,
       {
        start: rulerStartPoint,
        end: new Point(ctPoint.x, ctPoint.y),
        distance,
       },
      ],
     };
    });

    setRulerStartPoint(null);
    setIsRulerDrag(false);
   }
  } else if (isProtractorEnabled) {
   // Convert canvas pixel (x,y) to CT coordinate system

   const physicalPoint = translateDisplayToWorld(
    new Point(mouseX, mouseY),
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
```

```
  bodyWidth
);

if (draggingType === "resize" || draggingType === "arm") {
  setDraggingType(null);
  setResizeProtractorInfo(null);
  setMoveProtractorArmInfo(null);
  setLastPhysicalPoint(null);
  return;
}

if (resizeProtractorInfo) {
  setDraggingType("resize");
  setLastPhysicalPoint(physicalPoint);
} else if (moveProtractorArmInfo) {
  setDraggingType("arm");
  setLastPhysicalPoint(physicalPoint);
} else {
  if (currentAnglePoints.length === 0) {
    setCurrentAnglePoints([physicalPoint]);
  } else if (currentAnglePoints.length === 1) {
    setCurrentAnglePoints((prev) => [...prev, physicalPoint]);
  } else if (currentAnglePoints.length === 2) {
    const newProtractor = [...currentAnglePoints, physicalPoint];
    setAnglePointsCT((prev) => {
      const currentSlice = prev[ZCoordinate] || [];
      return {
        ...prev,
        [ZCoordinate]: [...currentSlice, newProtractor],
      };
    });
```

```
      setCurrentAnglePoints([]);
      setCurrentMouse(null);
    }
  }
} else if (isMeasureCircleEnabled) {
  const ctPoint = translateDisplayToWorld(
    new Point(mouseX, mouseY),
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth
  );

  if (isCircleMoving) {
    setIsCircleMoving(false);
    setMovedCircleIndex(null);
    setCircleStartMovePoint(null);
    return;
  }

  if (!currentCircleStart) {
    if (movedCircleIndex != null) {
      setIsCircleMoving(true);
      setCircleStartMovePoint(ctPoint);
      return;
    }

    if (resizeCircleIndex != null) {
```

```
      const updatedCircles = [...(circlePointsCT[ZCoordinate] || [])];
      const otherPoint =
        resizeCircleIndex.point === "p1"
          ? updatedCircles[resizeCircleIndex.index].p2
          : updatedCircles[resizeCircleIndex.index].p1;
      setCurrentCircleStart(otherPoint);
      updatedCircles.splice(resizeCircleIndex.index, 1);
      setCirclePointsCT((prev) => ({
        ...prev,
        [ZCoordinate]: updatedCircles,
      }));
      setResizeCircleIndex(null);
    } else {
      setCurrentCircleStart(ctPoint);
    }
  } else {
    setCirclePointsCT((prev) => {
      const currentSlice = prev[ZCoordinate] || [];
      return {
        ...prev,
        [ZCoordinate]: [
          ...currentSlice,
          { p1: currentCircleStart, p2: ctPoint },
        ],
      };
    });
    setCurrentCircleStart(null);
    setCurrentMouse(null);
  }
} else if (isRectangleProbeEnabled) {
  const ctPoint = translateDisplayToWorld(
```

```
  new Point(mouseX, mouseY),
  canvasWidth,
  canvasHeight,
  translateX,
  translateY,
  positionRect,
  bodyHeight,
  bodyWidth
);

if (isRectangleMoving) {
  setIsRectangleMoving(false);
  setMovedRectangleIndex(null);
  setRectangleStartMovePoint(null);
  return;
}

if (!currentRectangleStart) {
  if (movedRectangleIndex != null) {
    setIsRectangleMoving(true);
    setRectangleStartMovePoint(ctPoint);
    return;
  }

  if (resizeRectangleIndex != null) {
    const updatedRectangles = [
      ...(rectangleProbesCT[ZCoordinate] || []),
    ];
    const otherPoint =
      resizeRectangleIndex.point === "p1"
        ? updatedRectangles[resizeRectangleIndex.index].p2
```

```
        : updatedRectangles[resizeRectangleIndex.index].p1;

    setCurrentRectangleStart(otherPoint);
    updatedRectangles.splice(resizeRectangleIndex.index, 1);
    setRectangleProbesCT((prev) => ({
      ...prev,
      [ZCoordinate]: updatedRectangles,
    }));
    setResizeRectangleIndex(null);
  } else {
    setCurrentRectangleStart(ctPoint);
  }
} else {
  const newStats = computeRectangleStats(
    currentRectangleStart,
    ctPoint
  );
  const newRect = {
    p1: currentRectangleStart,
    p2: ctPoint,
    ...newStats,
  };
  setRectangleProbesCT((prev) => {
    const currentSlice = prev[ZCoordinate] || [];
    return {
      ...prev,
      [ZCoordinate]: [...currentSlice, newRect],
    };
  });

  setCurrentRectangleStart(null);
```

```
    setCurrentMouse(null);
   }
  } else if (isEllipseProbeEnabled) {
   const ctPoint = translateDisplayToWorld(
    new Point(mouseX, mouseY),
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth
   );

   if (isEllipseMoving) {
    setIsEllipseMoving(false);
    setMovedEllipseIndex(null);
    setEllipseStartMovePoint(null);
    return;
   }

   if (!currentEllipseStart) {
    if (movedEllipseIndex != null) {
     setIsEllipseMoving(true);
     setEllipseStartMovePoint(ctPoint);
     return;
    }

    if (resizeEllipseIndex != null) {
     const updatedEllipses = [...(ellipseProbesCT[ZCoordinate] || [])];
     const otherPoint =
```

```
      resizeEllipseIndex.point === "p1"
        ? updatedEllipses[resizeEllipseIndex.index].p2
        : updatedEllipses[resizeEllipseIndex.index].p1;
    setCurrentEllipseStart(otherPoint);
    updatedEllipses.splice(resizeEllipseIndex.index, 1);
    setEllipseProbesCT((prev) => ({
      ...prev,
      [ZCoordinate]: updatedEllipses,
    }));
    setResizeEllipseIndex(null);
  } else {
    setCurrentEllipseStart(ctPoint);
  }
} else {
  const newStats = computeEllipseStats(currentEllipseStart, ctPoint);
  const newEllipse = {
    p1: currentEllipseStart,
    p2: ctPoint,
    ...newStats,
  };
  setEllipseProbesCT((prev) => {
    const currentSlice = prev[ZCoordinate] || [];
    return {
      ...prev,
      [ZCoordinate]: [...currentSlice, newEllipse],
    };
  });

  setCurrentEllipseStart(null);
  setCurrentMouse(null);
}
```

```
} else if (isPixelProbeEnabled) {
  const ctPoint = translateDisplayToWorld(
    new Point(mouseX, mouseY),
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth
  );

  if (isMovingPixelProbe && movedPixelProbeIndex !== null) {
    // Drop the probe at new location
    setPixelProbesCT((prev) => {
      const existing = prev[ZCoordinate] || [];
      const updated = [...existing];
      updated.splice(movedPixelProbeIndex, 0, {
        point: ctPoint,
        hu: findHU(ctPoint),
      });
      return {
        ...prev,
        [ZCoordinate]: updated,
      };
    });

    setIsMovingPixelProbe(false);
    setMovedPixelProbeIndex(null);
    setPreviewPixelPoint(null);
    return;
```

```
      }

      if (movedPixelProbeIndex !== null) {
        // Start moving: remove from array and show preview
        setPixelProbesCT((prev) => {
          const existing = prev[ZCoordinate] || [];
          const updated = [...existing];
          updated.splice(movedPixelProbeIndex, 1);
          return {
            ...prev,
            [ZCoordinate]: updated,
          };
        });

        setIsMovingPixelProbe(true);
        setPreviewPixelPoint({ point: ctPoint, hu: findHU(ctPoint) });
        return;
      }

      // Add new probe
      setPixelProbesCT((prev) => {
        const existingPixels = prev[ZCoordinate] || [];
        const newProbe = { point: ctPoint, hu: findHU(ctPoint) };
        return {
          ...prev,
          [ZCoordinate]: [...existingPixels, newProbe],
        };
      });
    } else if (isTextAnnotationEnabled) {
if (event.detail === 2) {
 const currentSlice = textAnnotations[ZCoordinate] || [];
```

```
for (let i = 0; i < currentSlice.length; i++) {
  const ann = currentSlice[i];

  const tail = translateWorldToDisplay(
    ann.tail,
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth
  );

  const head = translateWorldToDisplay(
    ann.head,
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth
  );

  const distance = pointToSegmentDistance(
    { x: mouseX, y: mouseY },
    tail,
    head
  );
```

```
    if (distance < 5) {
const newText = prompt("Edit annotation text:", ann.text);
if (newText !== null && newText !== ann.text) {

  // Show preview edit ONLY
  setTextAnnotations(prev => {
    const updatedSlice = [...(prev[ZCoordinate] || [])];
    updatedSlice[i] = {
      ...ann,
      text: newText
    };
    return {
      ...prev,
      [ZCoordinate]: updatedSlice
    };
  });

  // Store edit context (NO DB yet)
  setPendingEditAnnotation({
    index: i,
    oldText: ann.text,
    newText: newText
  });

  setIsConfirmingText(true);
}



      // 🔥 CRITICAL: clear all interaction state
      setResizedTextAnnotationsIndex(null);
```

```
    setMovedTextAnnotationIndex(null);
    setResizingTextPointType(null);
    setIsTextMoving(false);
    setCurrentTextTail(null);
    setActiveTextValue(null);

    return; // HARD STOP
   }
  }

 return; // <-- also stop if double click but no hit
}



      const ctPoint = translateDisplayToWorld(
        new Point(mouseX, mouseY),
        canvasWidth,
        canvasHeight,
        translateX,
        translateY,
        positionRect,
        bodyHeight,
        bodyWidth
      );

      // Finalize movement on second click
 if (isTextMoving && movedTextAnnotationIndex !== null) {
  setIsTextMoving(false);
  setIsConfirmingText(true); // 👈 same flag you already use
  return;
}
```

```
      // First click
      if (!currentTextTail) {
       if (movedTextAnnotationIndex != null) {
  const annotation =
    textAnnotations[ZCoordinate][movedTextAnnotationIndex];

  // 🔒 store original position
  setPendingMoveAnnotation({
    index: movedTextAnnotationIndex,
    originalHead: annotation.head,
    originalTail: annotation.tail,
  });

  setActiveTextValue(annotation.text);
  setIsTextMoving(true);
  setTextStartMovePoint(ctPoint);

  return;
}


        if (resizedTextAnnotationsIndex != null) {
          const { index, point } = resizedTextAnnotationsIndex;
          const annotation = textAnnotations[ZCoordinate][index];
          // Save old text so that it is used during modification
          setActiveTextValue(annotation.text);
          const otherPoint =
            point === "head" ? annotation.tail : annotation.head;
```

```
    // Start from the fixed point (opposite of the one being resized)
    setCurrentTextTail(otherPoint);
    setResizingTextPointType(point); // Track what we're resizing

    // Remove the existing annotation
    const updated = [...textAnnotations[ZCoordinate]];
    updated.splice(index, 1);
    setTextAnnotations((prev) => ({
      ...prev,
      [ZCoordinate]: updated,
    }));
    setResizedTextAnnotationsIndex(null);
    return;
  }

  // If not moving or resizing, we're starting a new annotation: set head
  setCurrentTextTail(ctPoint);
  setResizingTextPointType(null); // Not resizing
  return;
}

// Second click — complete the annotation
let head, tail;

if (resizingTextPointType === "head") {
  head = ctPoint;
  tail = currentTextTail;
} else if (resizingTextPointType === "tail") {
  head = currentTextTail;
  tail = ctPoint;
```

```
    } else {
      // For new annotations
      head = currentTextTail;
      tail = ctPoint;
    }

    // Only ask for new user text prompt if the annotation is brand new - nto while resizing or moving
    const isNewAnnotation =
      resizingTextPointType == null && activeTextValue == null;

    let finalText = activeTextValue;
    if (isNewAnnotation) {
      finalText = prompt("Enter annotation text:");

      if (!finalText) {
        setCurrentTextTail(null);
        setResizingTextPointType(null);
        setActiveTextValue(null);
        return;
      }
    }

    setCurrentMouse(null);

   // Store temporarily instead of saving
setPendingTextAnnotation({ head, tail, text: finalText });
setIsConfirmingText(true);

    setActiveTextValue(null);
```

```
      // Reset state
      setCurrentTextTail(null);
      setResizingTextPointType(null);
    } else if (isDeleteAnnotationEnabled) {
      // Delete the ruler that is selected. In this case, if any ruler is highlighted, we will not be able to create a new ruler. Hence, if else
      if (hoveredRulerIndex != null && rulerPointsCT[ZCoordinate]) {
        const updatedRulers = [...rulerPointsCT[ZCoordinate]];
        updatedRulers.splice(hoveredRulerIndex, 1);
        setRulerPointsCT((prev) => ({
          ...prev,
          [ZCoordinate]: updatedRulers,
        }));
        setHoveredRulerIndex(null);
      }

      const currentSliceProtractors = anglePointsCT[ZCoordinate] || [];
      if (hoveredProtractorIndex != null && currentSliceProtractors) {
        const updatedSliceProtractors = [...currentSliceProtractors];
        updatedSliceProtractors.splice(hoveredProtractorIndex, 1);
        setAnglePointsCT((prev) => ({
          ...prev,
          [ZCoordinate]: updatedSliceProtractors,
        }));
        setHoveredProtractorIndex(null);
      }

      const currentSliceCircles = circlePointsCT[ZCoordinate] || [];
      if (hoveredCircleIndex != null && currentSliceCircles) {
        const updatedCircles = [...currentSliceCircles];
        updatedCircles.splice(hoveredCircleIndex, 1);
```

```
  setCirclePointsCT((prev) => ({
    ...prev,
    [ZCoordinate]: updatedCircles,
  }));

  setHoveredCircleIndex(null);
}
const currentSliceRects = rectangleProbesCT[ZCoordinate] || [];

if (hoveredRectangleIndex != null) {
  const updatedRects = [...currentSliceRects];
  updatedRects.splice(hoveredRectangleIndex, 1);

  setRectangleProbesCT((prev) => ({
    ...prev,
    [ZCoordinate]: updatedRects,
  }));

  setHoveredRectangleIndex(null);
}

const currentSliceEllipses = ellipseProbesCT[ZCoordinate] || [];
if (hoveredEllipseIndex != null) {
  const updatedEllipses = [...currentSliceEllipses];
  updatedEllipses.splice(hoveredEllipseIndex, 1);

  setEllipseProbesCT((prev) => ({
    ...prev,
    [ZCoordinate]: updatedEllipses,
  }));
```

```
        setHoveredEllipseIndex(null);
      }

      if (hoveredPixelIndex != null && pixelProbesCT[ZCoordinate]) {
        const updatedPixels = [...pixelProbesCT[ZCoordinate]];
        updatedPixels.splice(hoveredPixelIndex, 1);
        setPixelProbesCT((prev) => ({
          ...prev,
          [ZCoordinate]: updatedPixels,
        }));
        setHoveredPixelIndex(null);
      }
      const currentSliceTextAnnotations = textAnnotations[ZCoordinate] || [];

if (hoveredTextAnnotationIndex != null && currentSliceTextAnnotations) {
  const annotationToDelete =
    currentSliceTextAnnotations[hoveredTextAnnotationIndex];

  try {
    await fetch(
      `http://localhost:5000/api/annotations/${annotationToDelete.id}`,
      {
        method: "DELETE",
      }
    );
  } catch (err) {
    console.error("Failed to delete annotation:", err);
  }

  const updatedTextAnnotations = [...currentSliceTextAnnotations];
  updatedTextAnnotations.splice(hoveredTextAnnotationIndex, 1);
```

```
  setTextAnnotations((prev) => ({
    ...prev,
    [ZCoordinate]: updatedTextAnnotations,
  }));

  setHoveredTextAnnotationIndex(null);
}

      } else {
        if (!isBrushing && event.shiftKey && !isBrushEnabled) {
          // allow panning only when drawing is not done
          //Panning event

          setIsPanning(true);
          setStartPanX(event.clientX);
          setStartPanY(event.clientY);
        } else {
          // Add the common checks for all the types of brushes
          if (!selectedStructureUID) return;
          setIsMouseDown(true);

          if (
            isBrushEnabled &&
            mouseX > 0 &&
            mouseX <= canvasWidth &&
            mouseY > 0 &&
            mouseY <= canvasHeight
          ) {
            // SIRISHA: BRUSH IMPROVEMENT
            if (contourStyle === "Brush" || contourStyle === "Hollow Brush") {
```

```
const point = getCanvasPoint(event);

if (!point) return;
const selectedStructureSet = structureSets.find(
  (set) => set.structureSetUID === selectedStructureSetUID
);
const selectedStructure =
  selectedStructureSet.structuresList.find(
    (structure) => structure.structureID === selectedStructureUID
  );
let polygonsList = selectedStructure.polygonsList[ZCoordinate];

const mousePointCT = translateDisplayToWorld(
  point,
  canvasWidth,
  canvasHeight,
  translateX,
  translateY,
  positionRect,
  bodyHeight,
  bodyWidth
);
const polygonsCount = polygonsList?.length || 0;
const insidePolygonIndices = [];
for (let i = 0; i < polygonsCount; i++) {
  const polygon = polygonsList[i];
  if (isPointInPolygon(mousePointCT, polygon)) {
    insidePolygonIndices.push(i);
    //setIsPointInside({ status: true, polygonIndex: i });
    // console.log("Point is inside polygon");
  }
```

```
  }

  if (insidePolygonIndices.length > 0) {
    setIsPointInside({
      status: true,
      polygonIndices: insidePolygonIndices,
    });
  } else {
    setIsPointInside({ status: false, polygonIndices: [] });
  }

  setIsBrushing(true);
  setPrevPoint(point);
  const circlePoly = createCirclePolygon({
    ...point,
    radius: brushSize / 2,
  });

  const unioned = unionPolygons([circlePoly]);

  setBrushPolygons(unioned);
 } else {
  activeBrush.mousedown(
    event,
    ctx,
    mouseX,
    mouseY,
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
```

```
            positionRect,
            bodyHeight,
            bodyWidth
          );
        }
      }
      // ERASER FUNCTIONALITY SIRISHA100725
      else if (isEraserEnabled) {
        const point = getCanvasPoint(event);
        if (!point) return;
        setIsErasing(true);
        setPrevPoint(point);
        const circlePoly = createCirclePolygon({
          ...point,
          radius: brushSize / 2,
        });
        const unioned = unionEraser([circlePoly]);
        // console.log("Setting eraser polygons: ", unioned);
        setEraserPolygons(unioned);
      }
    }
  }
};
const handleMouseMove = (event) => {
  const rect = canvas.getBoundingClientRect();
  const mouseX = event.clientX - rect.left;
  const mouseY = event.clientY - rect.top;

  if (isCrosshairEnabled && HUValues) {
    const physicalPoint = translateDisplayToWorld(
      new Point(mouseX, mouseY),
```

```
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth
  );
  const Xslice = Math.floor((physicalPoint.x - minX) / pixelSpacing[0]);
  const Yslice = Math.floor((physicalPoint.y - minY) / pixelSpacing[1]);
  if (
    Xslice >= 0 &&
    Xslice < originalWidth &&
    Yslice >= 0 &&
    Yslice < originalHeight
  ) {
    canvas.style.cursor = "crosshair";

    setCTCoordinate(
      (prevCoordinate) =>
        new Point(physicalPoint.x.toFixed(2), physicalPoint.y.toFixed(2))
    );

    const HUVal = HUValues[Yslice][Xslice][currentZSlice];
    setCurrentHU(HUVal);
  } else {
    canvas.style.cursor = "default";
  }
} else {
  if (draggedRlAp === "RL") {
    canvas.style.cursor = "grabbing";
```

```
  setMiddleDisplayPoint((prev) => ({ ...prev, y: mouseY }));
  //setCurrentYSlice(200);
  // Find the slice corresponding to new mouseY
  // find the CT point corresponding to mouseY
  const newCTpoint = translateDisplayToWorld(
    new Point(mouseX, mouseY),
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth
  );
  const ySliceToGo = Math.floor(
    (newCTpoint.y - minY) / pixelSpacing[1]
  );
  setCurrentYSlice(ySliceToGo);
  return;
 } else if (draggedRlAp === "AP") {
  canvas.style.cursor = "grabbing";
  setMiddleDisplayPoint((prev) => ({ ...prev, x: mouseX }));
  const newCTpoint = translateDisplayToWorld(
    new Point(mouseX, mouseY),
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth
```

```
    );
    const xSliceToGo = Math.floor(
      (newCTpoint.x - minX) / pixelSpacing[0]
    );
    setCurrentXSlice(xSliceToGo);
    //setCurrentXSlice(147);

    return;
  }

  // Only detect hover if not dragging
  if (Math.abs(mouseY - middleDisplayPoint.y) < 5) {
    canvas.style.cursor = "grabbing";
    setHoveredRlAp("RL");
    return;
  } else if (Math.abs(mouseX - middleDisplayPoint.x) < 5) {
    canvas.style.cursor = "grabbing";
    setHoveredRlAp("AP");
    return;
  } else {
    canvas.style.cursor = "default";
    setHoveredRlAp(null);
  }
}

if (openAutoSegDialog) {
  if (!dragging && !resizing) return;
  if (
    mouseX < 0 ||
    mouseY < 0 ||
    mouseX > canvasWidth ||
```

```
  mouseY > canvasHeight
) {
  const clamp = (v, min, max) => Math.max(min, Math.min(v, max));

  const mx = clamp(mouseX, 0, canvasWidth);
  const my = clamp(mouseY, 0, canvasHeight);
}
const ctRect = new Rect(
  ROI3dCT.x,
  ROI3dCT.y,
  ROI3dCT.width,
  ROI3dCT.height
);

const displayRect = convertRectWorldToDisplay(
  ctRect,
  canvasWidth,
  canvasHeight,
  translateX,
  translateY,
  positionRect,
  bodyHeight,
  bodyWidth
);

let newDisplayRect = { ...displayRect };
if (dragging) {
  newDisplayRect = {
    x: mouseX - offset.x,
    y: mouseY - offset.y,
    width: displayRect.width,
```

```
    height: displayRect.height,
  };
 } else if (resizing) {
  const xMinDisplay = displayRect.x;
  const xMaxDisplay = displayRect.x + displayRect.width;
  const yMinDisplay = displayRect.y;
  const yMaxDisplay = displayRect.y + displayRect.height;

  switch (resizing) {
   case "top-left":
    newDisplayRect = {
     x: Math.min(mouseX, xMaxDisplay),
     y: Math.min(mouseY, yMaxDisplay),
     width: Math.abs(xMaxDisplay - mouseX),
     height: Math.abs(yMaxDisplay - mouseY),
    };
    break;

   case "top-right":
    newDisplayRect = {
     x: xMinDisplay, // xMin will remain same
     y: Math.min(mouseY, yMaxDisplay),
     width: Math.abs(mouseX - xMinDisplay),
     height: Math.abs(yMaxDisplay - mouseY),
    };
    break;

   case "bottom-left":
    newDisplayRect = {
     x: Math.min(mouseX, xMaxDisplay),
     y: yMinDisplay,
```

```
      width: Math.abs(xMaxDisplay - mouseX),
      height: Math.abs(mouseY - yMinDisplay),
    };
    break;

  case "bottom-right":
    newDisplayRect = {
      x: xMinDisplay,
      y: yMinDisplay,
      width: Math.abs(mouseX - xMinDisplay),
      height: Math.abs(mouseY - yMinDisplay),
    };
    break;

  default:
    return;
 }
}

const MIN_SIZE = 3;
newDisplayRect.width = Math.max(MIN_SIZE, newDisplayRect.width);
newDisplayRect.height = Math.max(MIN_SIZE, newDisplayRect.height);

const newCTRect = convertRectDisplayToWorld(
 newDisplayRect,
 canvasWidth,
 canvasHeight,
 translateX,
 translateY,
 positionRect,
 bodyHeight,
```

```
    bodyWidth
  );

  setROI3dCT((prev) => ({
    x: newCTRect.x,
    y: newCTRect.y,
    z: prev.z,

    width: newCTRect.width,
    height: newCTRect.height,
    depth: prev.depth,
  }));
} else {
  if (
    mouseX > 0 &&
    mouseX <= canvasWidth &&
    mouseY > 0 &&
    mouseY <= canvasHeight
  ) {
    if (isRulerEnabled) {
      if (isRulerDrag) {
        // searching for a second point of ruler
        const physicalPoint = translateDisplayToWorld(
          new Point(mouseX, mouseY),
          canvasWidth,
          canvasHeight,
          translateX,
          translateY,
          positionRect,
          bodyHeight,
          bodyWidth
```

```
  );
  setCurrentMouse(physicalPoint);
 } else if (isRulerMoving && movedRulerIndex != null) {
  // We have a ruler that needs to move
  const physicalPoint = translateDisplayToWorld(
   new Point(mouseX, mouseY),
   canvasWidth,
   canvasHeight,
   translateX,
   translateY,
   positionRect,
   bodyHeight,
   bodyWidth
  );
  const deltaX = physicalPoint.x - rulerStartMovePoint.x;
  const deltaY = physicalPoint.y - rulerStartMovePoint.y;
  setRulerPointsCT((prev) => {
   const updated = [...(prev[ZCoordinate] || [])];
   const ruler = updated[movedRulerIndex];
   if (ruler) {
    updated[movedRulerIndex] = {
     ...ruler,
     start: new Point(
      ruler.start.x + deltaX,
      ruler.start.y + deltaY
     ),
     end: new Point(ruler.end.x + deltaX, ruler.end.y + deltaY),
     distance: ruler.distance,
    };
   }
   return {
```

```
      ...prev,
      [ZCoordinate]: updated,
    };
  });

  // Update drag start to current position for smooth dragging
  setRulerStartMovePoint(physicalPoint);
 } else {
  // we are trying to drag any point of the ruler to resize the ruler
  let resizeRulerIndexPoint = null;
  let foundMovedRulerIndex = null;
  const currentSliceRulers = rulerPointsCT[ZCoordinate] || [];
  for (let i = 0; i < currentSliceRulers.length; i++) {
   const ruler = currentSliceRulers[i];

   const p1 = translateWorldToDisplay(
    ruler.start,
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth
   );
   const p2 = translateWorldToDisplay(
    ruler.end,
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
```

```
        positionRect,
        bodyHeight,
        bodyWidth
      );

      const distanceToStartPoint = Math.sqrt(
        (p1.x - mouseX) ** 2 + (p1.y - mouseY) ** 2
      );
      const distanceToEndPoint = Math.sqrt(
        (p2.x - mouseX) ** 2 + (p2.y - mouseY) ** 2
      );

      const distanceFromMouseToSegment = pointToSegmentDistance(
        { x: mouseX, y: mouseY },
        p1,
        p2
      );

      if (distanceToStartPoint < 3) {
        resizeRulerIndexPoint = { index: i, point: "start" }; // We will first check the start point
        break;
      } else if (distanceToEndPoint < 3) {
        resizeRulerIndexPoint = { index: i, point: "end" }; // Then check end point
        break;
      } else if (distanceFromMouseToSegment < 3) {
        // If none of points of the line are near mouse, then check for the line - as line also
contains the points - this must be done at last
        foundMovedRulerIndex = i;
        break;
      }
    }
```

```
    setResizeRulerIndex(resizeRulerIndexPoint);
    setMovedRulerIndex(foundMovedRulerIndex);
   }

   // For removing the ruler
  } else if (isProtractorEnabled) {
   const physicalPoint = translateDisplayToWorld(
    new Point(mouseX, mouseY),
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth
   );

   if (draggingType === "resize" && resizeProtractorInfo) {
    const { index, pointIndex } = resizeProtractorInfo;

    setAnglePointsCT((prev) => {
     const slice = [...(prev[ZCoordinate] || [])];
     const triangle = [...slice[index]];
     triangle[pointIndex] = physicalPoint;
     slice[index] = triangle;
     return { ...prev, [ZCoordinate]: slice };
    });
   } else if (
    draggingType === "arm" &&
    moveProtractorArmInfo &&
```

```
  lastPhysicalPoint
) {
  const { index, arm } = moveProtractorArmInfo;
  const deltaX = physicalPoint.x - lastPhysicalPoint.x;
  const deltaY = physicalPoint.y - lastPhysicalPoint.y;

  setAnglePointsCT((prev) => {
    const slice = [...(prev[ZCoordinate] || [])];
    const triangle = [...slice[index]];

    const i1 = arm === "p1p2" ? 0 : 1;
    const i2 = arm === "p1p2" ? 1 : 2;

    triangle[i1] = {
      ...triangle[i1],
      x: triangle[i1].x + deltaX,
      y: triangle[i1].y + deltaY,
    };
    triangle[i2] = {
      ...triangle[i2],
      x: triangle[i2].x + deltaX,
      y: triangle[i2].y + deltaY,
    };

    slice[index] = triangle;
    return { ...prev, [ZCoordinate]: slice };
  });

  setLastPhysicalPoint(physicalPoint);
} else {
  // Update hover states (only when not dragging)
```

```
const protractors = anglePointsCT[ZCoordinate] || [];
let foundResizeInfo = null;
let foundArmInfo = null;

for (let i = 0; i < protractors.length; i++) {
 const triangle = protractors[i];

 // Check for point hover
 for (let j = 0; j < 3; j++) {
  const dispPt = translateWorldToDisplay(
   triangle[j],
   canvasWidth,
   canvasHeight,
   translateX,
   translateY,
   positionRect,
   bodyHeight,
   bodyWidth
  );

  const dist = Math.sqrt(
   (dispPt.x - mouseX) ** 2 + (dispPt.y - mouseY) ** 2
  );
  if (dist < 4) {
   foundResizeInfo = { index: i, pointIndex: j };
   break;
  }
 }

 if (!foundResizeInfo) {
  // Check for arm hover (either p1p2 or p2p3)
```

```
const checkArm = (a, b, armName) => {
  const pt1 = translateWorldToDisplay(
    triangle[a],
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth
  );
  const pt2 = translateWorldToDisplay(
    triangle[b],
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth
  );

  // Distance from mouse to segment
  const distToSegment = pointToSegmentDistance(
    mouseX,
    mouseY,
    pt1.x,
    pt1.y,
    pt2.x,
    pt2.y
  );
```

```
        if (distToSegment < 4) {
          foundArmInfo = { index: i, arm: armName };
        }
      };

      checkArm(0, 1, "p1p2");
      checkArm(1, 2, "p2p3");
    }

    if (foundResizeInfo || foundArmInfo) break;
  }

  setResizeProtractorInfo(foundResizeInfo);
  setMoveProtractorArmInfo(foundArmInfo);

  // Show dynamic line while placing 2nd or 3rd point
  if (currentAnglePoints.length >= 1) {
    setCurrentMouse(physicalPoint);
  }
 }
} else if (isMeasureCircleEnabled) {
 if (currentCircleStart) {
  const physicalPoint = translateDisplayToWorld(
    new Point(mouseX, mouseY),
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth
```

```
  );
  setCurrentMouse(physicalPoint);
 } else if (isCircleMoving && movedCircleIndex != null) {
  const physicalPoint = translateDisplayToWorld(
   new Point(mouseX, mouseY),
   canvasWidth,
   canvasHeight,
   translateX,
   translateY,
   positionRect,
   bodyHeight,
   bodyWidth
  );
  const deltaX = physicalPoint.x - circleStartMovePoint.x;
  const deltaY = physicalPoint.y - circleStartMovePoint.y;

  setCirclePointsCT((prev) => {
   const updated = [...(prev[ZCoordinate] || [])];
   const circle = updated[movedCircleIndex];
   if (circle) {
    updated[movedCircleIndex] = {
     p1: new Point(circle.p1.x + deltaX, circle.p1.y + deltaY),
     p2: new Point(circle.p2.x + deltaX, circle.p2.y + deltaY),
    };
   }
   return { ...prev, [ZCoordinate]: updated };
  });

  setCircleStartMovePoint(physicalPoint);
 } else {
  const currentSlice = circlePointsCT[ZCoordinate] || [];
```

```
let foundMovedCircleIndex = null;
let foundResizeIndex = null;

for (let i = 0; i < currentSlice.length; i++) {
  const circle = currentSlice[i];

  const dp1 = translateWorldToDisplay(
    circle.p1,
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth
  );
  const dp2 = translateWorldToDisplay(
    circle.p2,
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth
  );

  const distToP1 = Math.hypot(dp1.x - mouseX, dp1.y - mouseY);
  const distToP2 = Math.hypot(dp2.x - mouseX, dp2.y - mouseY);

  const center = {
```

```
      x: (dp1.x + dp2.x) / 2,
      y: (dp1.y + dp2.y) / 2,
    };
    const radius = Math.hypot(dp1.x - dp2.x, dp1.y - dp2.y) / 2;
    const distToCircumference = Math.abs(
      Math.hypot(center.x - mouseX, center.y - mouseY) - radius
    );

    if (distToP1 < 5) {
      foundResizeIndex = { index: i, point: "p1" };
      break;
    } else if (distToP2 < 5) {
      foundResizeIndex = { index: i, point: "p2" };
      break;
    } else if (distToCircumference < 5) {
      foundMovedCircleIndex = i;
    }
  }

  setResizeCircleIndex(foundResizeIndex);
  setMovedCircleIndex(foundMovedCircleIndex);
 }
} else if (isPixelProbeEnabled) {
 const probes = pixelProbesCT[ZCoordinate] || [];

 // Find hover index
 if (!isMovingPixelProbe) {
  let foundIndex = null;
  for (let i = 0; i < probes.length; i++) {
   const displayPoint = translateWorldToDisplay(
    probes[i].point,
```

```
      canvasWidth,
      canvasHeight,
      translateX,
      translateY,
      positionRect,
      bodyHeight,
      bodyWidth
    );

    const distance = Math.hypot(
      displayPoint.x - mouseX,
      displayPoint.y - mouseY
    );
    if (distance < 3) {
      foundIndex = i;
      break;
    }
  }
  setMovedPixelProbeIndex(foundIndex);
}
// Update live preview if moving
if (isMovingPixelProbe) {
  const ctPoint = translateDisplayToWorld(
    new Point(mouseX, mouseY),
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth
```

```
    );
    const hu = findHU(ctPoint);
    setPreviewPixelPoint({ point: ctPoint, hu: hu });
   }
  } else if (isRectangleProbeEnabled) {
   if (currentRectangleStart) {
    const physicalPoint = translateDisplayToWorld(
     new Point(mouseX, mouseY),
     canvasWidth,
     canvasHeight,
     translateX,
     translateY,
     positionRect,
     bodyHeight,
     bodyWidth
    );
    setCurrentMouse(physicalPoint);
   } else if (isRectangleMoving && movedRectangleIndex != null) {
    const physicalPoint = translateDisplayToWorld(
     new Point(mouseX, mouseY),
     canvasWidth,
     canvasHeight,
     translateX,
     translateY,
     positionRect,
     bodyHeight,
     bodyWidth
    );

    const deltaX = physicalPoint.x - rectangleStartMovePoint.x;
    const deltaY = physicalPoint.y - rectangleStartMovePoint.y;
```

```
  setRectangleProbesCT((prev) => {
    const updated = [...(prev[ZCoordinate] || [])];
    const rect = updated[movedRectangleIndex];
    if (rect) {
      const newP1 = new Point(
        rect.p1.x + deltaX,
        rect.p1.y + deltaY
      );
      const newP2 = new Point(
        rect.p2.x + deltaX,
        rect.p2.y + deltaY
      );
      updated[movedRectangleIndex] = {
        p1: newP1,
        p2: newP2,
        ...computeRectangleStats(newP1, newP2),
      };
    }
    return { ...prev, [ZCoordinate]: updated };
  });

  setRectangleStartMovePoint(physicalPoint);
} else {
  const currentSlice = rectangleProbesCT[ZCoordinate] || [];
  let foundMovedIndex = null;
  let foundResizeIndex = null;

  for (let i = 0; i < currentSlice.length; i++) {
    const rect = currentSlice[i];
```

```
const dp1 = translateWorldToDisplay(
  rect.p1,
  canvasWidth,
  canvasHeight,
  translateX,
  translateY,
  positionRect,
  bodyHeight,
  bodyWidth
);
const dp2 = translateWorldToDisplay(
  rect.p2,
  canvasWidth,
  canvasHeight,
  translateX,
  translateY,
  positionRect,
  bodyHeight,
  bodyWidth
);

const distToP1 = Math.hypot(dp1.x - mouseX, dp1.y - mouseY);
const distToP2 = Math.hypot(dp2.x - mouseX, dp2.y - mouseY);

if (distToP1 < 5) {
  foundResizeIndex = { index: i, point: "p1" };
  break;
} else if (distToP2 < 5) {
  foundResizeIndex = { index: i, point: "p2" };
  break;
} else {
```

```
    const minX = Math.min(dp1.x, dp2.x);
    const maxX = Math.max(dp1.x, dp2.x);
    const minY = Math.min(dp1.y, dp2.y);
    const maxY = Math.max(dp1.y, dp2.y);

    const nearLeft =
      Math.abs(mouseX - minX) < 3 &&
      mouseY >= minY &&
      mouseY <= maxY;
    const nearRight =
      Math.abs(mouseX - maxX) < 3 &&
      mouseY >= minY &&
      mouseY <= maxY;
    const nearTop =
      Math.abs(mouseY - minY) < 3 &&
      mouseX >= minX &&
      mouseX <= maxX;
    const nearBottom =
      Math.abs(mouseY - maxY) < 3 &&
      mouseX >= minX &&
      mouseX <= maxX;

    if (nearLeft || nearRight || nearTop || nearBottom) {
      foundMovedIndex = i;
    }
   }
  }

  setResizeRectangleIndex(foundResizeIndex);
  setMovedRectangleIndex(foundMovedIndex);
}
```

```
} else if (isEllipseProbeEnabled) {
  const ctPoint = translateDisplayToWorld(
    new Point(mouseX, mouseY),
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth
  );

  if (currentEllipseStart) {
    setCurrentMouse(ctPoint);
  } else if (isEllipseMoving && movedEllipseIndex != null) {
    const deltaX = ctPoint.x - ellipseStartMovePoint.x;
    const deltaY = ctPoint.y - ellipseStartMovePoint.y;

    setEllipseProbesCT((prev) => {
      const updated = [...(prev[ZCoordinate] || [])];
      const ellipse = updated[movedEllipseIndex];
      if (ellipse) {
        updated[movedEllipseIndex] = {
          ...ellipse,
          p1: new Point(ellipse.p1.x + deltaX, ellipse.p1.y + deltaY),
          p2: new Point(ellipse.p2.x + deltaX, ellipse.p2.y + deltaY),
        };
      }
      return { ...prev, [ZCoordinate]: updated };
    });
```

```
  setEllipseStartMovePoint(ctPoint);
 } else {
  const currentSlice = ellipseProbesCT[ZCoordinate] || [];
  let foundMovedEllipseIndex = null;
  let foundResizeIndex = null;

  for (let i = 0; i < currentSlice.length; i++) {
   const ellipse = currentSlice[i];

   const dp1 = translateWorldToDisplay(
    ellipse.p1,
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth
   );
   const dp2 = translateWorldToDisplay(
    ellipse.p2,
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth
   );

   const distToP1 = Math.hypot(dp1.x - mouseX, dp1.y - mouseY);
```

```
    const distToP2 = Math.hypot(dp2.x - mouseX, dp2.y - mouseY);

    const center = {
      x: (dp1.x + dp2.x) / 2,
      y: (dp1.y + dp2.y) / 2,
    };
    const rx = Math.abs(dp1.x - dp2.x) / 2;
    const ry = Math.abs(dp1.y - dp2.y) / 2;

    const angle = Math.atan2(mouseY - center.y, mouseX - center.x);
    const ex = center.x + rx * Math.cos(angle);
    const ey = center.y + ry * Math.sin(angle);

    const distToEdge = Math.hypot(ex - mouseX, ey - mouseY);

    if (distToP1 < 5) {
      foundResizeIndex = { index: i, point: "p1" };
      break;
    } else if (distToP2 < 5) {
      foundResizeIndex = { index: i, point: "p2" };
      break;
    } else if (distToEdge < 3) {
      foundMovedEllipseIndex = i;
    }
   }

   setResizeEllipseIndex(foundResizeIndex);
   setMovedEllipseIndex(foundMovedEllipseIndex);
  }
 } else if (isTextAnnotationEnabled) {
  const ctPoint = translateDisplayToWorld(
```

```
  new Point(mouseX, mouseY),
  canvasWidth,
  canvasHeight,
  translateX,
  translateY,
  positionRect,
  bodyHeight,
  bodyWidth
);

// When drawing or resizing, show dynamic line (head → mouse)
if (currentTextTail) {
  setCurrentMouse(ctPoint);
}
// When moving annotation
else if (isTextMoving && movedTextAnnotationIndex != null) {
  const deltaX = ctPoint.x - textStartMovePoint.x;
  const deltaY = ctPoint.y - textStartMovePoint.y;

  setTextAnnotations((prev) => {
    const updated = [...(prev[ZCoordinate] || [])];
    const annotation = updated[movedTextAnnotationIndex];
    if (annotation) {
      updated[movedTextAnnotationIndex] = {
        ...annotation,
        head: new Point(
          annotation.head.x + deltaX,
          annotation.head.y + deltaY
        ),
        tail: new Point(
          annotation.tail.x + deltaX,
```

```
          annotation.tail.y + deltaY
        ),
      };
    }
    return { ...prev, [ZCoordinate]: updated };
  });

  setTextStartMovePoint(ctPoint);
}
// When hovering over annotations
else {
  const currentSlice = textAnnotations[ZCoordinate] || [];
  let foundHover = null;

  for (let i = 0; i < currentSlice.length; i++) {
    const ann = currentSlice[i];

    const dTail = translateWorldToDisplay(
      ann.tail,
      canvasWidth,
      canvasHeight,
      translateX,
      translateY,
      positionRect,
      bodyHeight,
      bodyWidth
    );
    const dHead = translateWorldToDisplay(
      ann.head,
      canvasWidth,
      canvasHeight,
```

```
  translateX,
  translateY,
  positionRect,
  bodyHeight,
  bodyWidth
);

const distToTail = Math.hypot(
  dTail.x - mouseX,
  dTail.y - mouseY
);
const distToHead = Math.hypot(
  dHead.x - mouseX,
  dHead.y - mouseY
);
const nearLine = checkIfPointNearLine(
  dTail,
  dHead,
  mouseX,
  mouseY,
  3
);

if (distToTail < 5) {
  foundHover = { index: i, point: "tail" };
  break;
} else if (distToHead < 5) {
  foundHover = { index: i, point: "head" };
  break;
} else if (nearLine) {
  foundHover = { index: i, point: "body" };
```

```
        break;
      }
    }

    if (foundHover?.point === "body") {
      setMovedTextAnnotationIndex(foundHover.index);
      setResizedTextAnnotationsIndex(null);
    } else if (foundHover) {
      setResizedTextAnnotationsIndex(foundHover);
      setMovedTextAnnotationIndex(null);
    } else {
      setMovedTextAnnotationIndex(null);
      setResizedTextAnnotationsIndex(null);
    }
  }
} else if (isDeleteAnnotationEnabled) {
  // Finding for ruler
  let foundRulerIndex = null;
  const currentSliceRulers = rulerPointsCT[ZCoordinate] || [];
  for (let i = 0; i < currentSliceRulers.length; i++) {
    const ruler = currentSliceRulers[i];

    const p1 = translateWorldToDisplay(
      ruler.start,
      canvasWidth,
      canvasHeight,
      translateX,
      translateY,
      positionRect,
      bodyHeight,
      bodyWidth
```

```
  );
  const p2 = translateWorldToDisplay(
    ruler.end,
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth
  );

  const distance = pointToSegmentDistance(
    { x: mouseX, y: mouseY },
    p1,
    p2
  );

  if (distance < 3) {
    foundRulerIndex = i;
    break;
  }
}

setHoveredRulerIndex(foundRulerIndex);

let foundProtractorIndex = null;
const currentSliceProtractors = anglePointsCT[ZCoordinate] || [];
for (let i = 0; i < currentSliceProtractors.length; i++) {
  const [p1, p2, p3] = currentSliceProtractors[i];
```

```
const dp1 = translateWorldToDisplay(
  p1,
  canvasWidth,
  canvasHeight,
  translateX,
  translateY,
  positionRect,
  bodyHeight,
  bodyWidth
);
const dp2 = translateWorldToDisplay(
  p2,
  canvasWidth,
  canvasHeight,
  translateX,
  translateY,
  positionRect,
  bodyHeight,
  bodyWidth
);
const dp3 = translateWorldToDisplay(
  p3,
  canvasWidth,
  canvasHeight,
  translateX,
  translateY,
  positionRect,
  bodyHeight,
  bodyWidth
);
```

```
  const d1 = pointToSegmentDistance(
    { x: mouseX, y: mouseY },
    dp1,
    dp2
  );
  const d2 = pointToSegmentDistance(
    { x: mouseX, y: mouseY },
    dp3,
    dp2
  );

  if (d1 < 3 || d2 < 3) {
    foundProtractorIndex = i;
    break;
  }
}

setHoveredProtractorIndex(foundProtractorIndex);

let foundCircleIndex = null;
const currentSliceCircles = circlePointsCT[ZCoordinate] || [];

for (let i = 0; i < currentSliceCircles.length; i++) {
  const { p1, p2 } = currentSliceCircles[i];
  const dp1 = translateWorldToDisplay(
    p1,
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
```

```
    bodyHeight,
    bodyWidth
  );
  const dp2 = translateWorldToDisplay(
    p2,
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth
  );

  const centerX = (dp1.x + dp2.x) / 2;
  const centerY = (dp1.y + dp2.y) / 2;
  const radius =
    Math.sqrt((dp2.x - dp1.x) ** 2 + (dp2.y - dp1.y) ** 2) / 2;

  const distToCenter = Math.sqrt(
    (mouseX - centerX) ** 2 + (mouseY - centerY) ** 2
  );

  if (Math.abs(distToCenter - radius) < 4) {
    foundCircleIndex = i;
    break;
  }
}

setHoveredCircleIndex(foundCircleIndex);
```

```
let foundRectangleIndex = null;
const currentSliceRects = rectangleProbesCT[ZCoordinate] || [];

for (let i = 0; i < currentSliceRects.length; i++) {
  const { p1, p2 } = currentSliceRects[i];
  const dp1 = translateWorldToDisplay(
    p1,
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth
  );
  const dp2 = translateWorldToDisplay(
    p2,
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth
  );

  const x1 = Math.min(dp1.x, dp2.x);
  const x2 = Math.max(dp1.x, dp2.x);
  const y1 = Math.min(dp1.y, dp2.y);
  const y2 = Math.max(dp1.y, dp2.y);
```

```
  const nearLeft =
    Math.abs(mouseX - x1) < 3 && mouseY >= y1 && mouseY <= y2;
  const nearRight =
    Math.abs(mouseX - x2) < 3 && mouseY >= y1 && mouseY <= y2;
  const nearTop =
    Math.abs(mouseY - y1) < 3 && mouseX >= x1 && mouseX <= x2;
  const nearBottom =
    Math.abs(mouseY - y2) < 3 && mouseX >= x1 && mouseX <= x2;

  if (nearLeft || nearRight || nearTop || nearBottom) {
    foundRectangleIndex = i;
    break;
  }
}

setHoveredRectangleIndex(foundRectangleIndex);

let foundEllipseIndex = null;
const currentSliceEllipses = ellipseProbesCT[ZCoordinate] || [];
for (let i = 0; i < currentSliceEllipses.length; i++) {
  const { p1, p2 } = currentSliceEllipses[i];
  const dp1 = translateWorldToDisplay(
    p1,
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth
  );
```

```
  const dp2 = translateWorldToDisplay(
    p2,
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth
  );

  const centerX = (dp1.x + dp2.x) / 2;
  const centerY = (dp1.y + dp2.y) / 2;
  const radiusX = Math.abs(dp2.x - dp1.x) / 2;
  const radiusY = Math.abs(dp2.y - dp1.y) / 2;

  const dx = mouseX - centerX;
  const dy = mouseY - centerY;

  const ellipseEq =
    (dx * dx) / (radiusX * radiusX) +
    (dy * dy) / (radiusY * radiusY);

  if (Math.abs(ellipseEq - 1) < 0.1) {
    foundEllipseIndex = i;
    break;
  }
}

setHoveredEllipseIndex(foundEllipseIndex);
```

```
if (pixelProbesCT[ZCoordinate]) {
  let foundPixelIndex = null;
  for (let i = 0; i < pixelProbesCT[ZCoordinate].length; i++) {
    const p1 = pixelProbesCT[ZCoordinate][i].point;
    const dp1 = translateWorldToDisplay(
      p1,
      canvasWidth,
      canvasHeight,
      translateX,
      translateY,
      positionRect,
      bodyHeight,
      bodyWidth
    );

    const distance = Math.sqrt(
      (dp1.x - mouseX) ** 2 + (dp1.y - mouseY) ** 2
    );
    if (distance <= 3) {
      foundPixelIndex = i;
      break;
    }
  }

  setHoveredPixelIndex(foundPixelIndex);
}
let foundTextAnnotationIndex = null;
const currentSliceTextAnnotations =
  textAnnotations[ZCoordinate] || [];

for (let i = 0; i < currentSliceTextAnnotations.length; i++) {
```

```
const annotation = currentSliceTextAnnotations[i];

const tail = translateWorldToDisplay(
  annotation.tail,
  canvasWidth,
  canvasHeight,
  translateX,
  translateY,
  positionRect,
  bodyHeight,
  bodyWidth
);

const head = translateWorldToDisplay(
  annotation.head,
  canvasWidth,
  canvasHeight,
  translateX,
  translateY,
  positionRect,
  bodyHeight,
  bodyWidth
);

const distance = pointToSegmentDistance(
  { x: mouseX, y: mouseY },
  tail,
  head
);

if (distance < 3) {
```

```
      foundTextAnnotationIndex = i;
      break;
    }
  }

  setHoveredTextAnnotationIndex(foundTextAnnotationIndex);
}

if (isPanning) {
  const dx = event.clientX - startPanX;
  const dy = event.clientY - startPanY;
  setTranslateX((translateX) => translateX + dx);
  setTranslateY((translateY) => translateY + dy);
  setStartPanX(event.clientX);
  setStartPanY(event.clientY);
}

if (isMouseDown && selectedStructureUID && isBrushEnabled) {
  // SIRISHA: BRUSH IMPROVEMENT
  if (contourStyle === "Brush" || contourStyle === "Hollow Brush") {
    if (!isBrushing || !prevPoint) return;

    const currentPoint = getCanvasPoint(event);
    if (!currentPoint) return;

    const dist = findDistance(prevPoint, currentPoint);
    if (dist < brushSize / 2) return;

    const step = brushSize / 4;
    const tStep = step / dist;
```

```
  const interpolated = [];

  for (let t = 0; t < 1; t += tStep) {
    const x = lerp(prevPoint.x, currentPoint.x, t);
    const y = lerp(prevPoint.y, currentPoint.y, t);
    interpolated.push(
      createCirclePolygon({ x, y, radius: brushSize / 2 })
    );
  }

  const unioned = unionPolygons(interpolated);
  setBrushPolygons(unioned);
  setPrevPoint(currentPoint);
 } else if (typeof activeBrush.mousemove === "function") {
  activeBrush.mousemove(
    event,
    ctx,
    mouseX,
    mouseY,
    canvasWidth,
    canvasHeight,
    translateX,
    translateY,
    positionRect,
    bodyHeight,
    bodyWidth,
    canvasHeight,
    canvasWidth,
    currentZSlice
  );
 }
```

```
    } else if (isEraserEnabled) {
      if (!isErasing || !prevPoint) return;

      const currentPoint = getCanvasPoint(event);
      if (!currentPoint) return;

      const dist = findDistance(prevPoint, currentPoint);
      if (dist < brushSize / 2) return;

      const step = brushSize / 4;
      const tStep = step / dist;

      const interpolated = [];

      for (let t = 0; t < 1; t += tStep) {
        const x = lerp(prevPoint.x, currentPoint.x, t);
        const y = lerp(prevPoint.y, currentPoint.y, t);
        interpolated.push(
          createCirclePolygon({ x, y, radius: brushSize / 2 })
        );
      }

      const unioned = unionEraser(interpolated);
      //   console.log("Setting eraser polygons: ", unioned);
      setEraserPolygons(unioned);
      setPrevPoint(currentPoint);
    }
  }
 }
};
```

```
const handleMouseUp = (event) => {
  setDraggedRlAp(null);
  //  canvas.style.cursor = "default";
  if (openAutoSegDialog) {
    setDragging(false);
    setResizing(null);
  } else {
    setIsMouseDown(false);
    setIsPanning(false);
    // SIRISHA: BRUSH IMPROVEMENT
    if (isBrushEnabled) {
      if (contourStyle === "Brush" || contourStyle === "Hollow Brush") {
        //console.log("Mouse up Point inside polygon: ", isPointInside);
        setIsBrushing(false);
        setPrevPoint(null);
        const selectedStructureSet = structureSets?.find(
          (set) => set.structureSetUID === selectedStructureSetUID
        );

        const selectedStructure = selectedStructureSet?.structuresList.find(
          (structure) => structure.structureID === selectedStructureUID
        );
        const existingPolygons =
          selectedStructure?.polygonsList[ZCoordinate];
        let existingPolygonsModified = existingPolygons?.map((polygon) => [
          polygon.map((point) => [point.x, point.y]),
        ]);
        const brushPolygonsCT = brushPolygons.map((polygon) => {
          // For "Brush", take only the first ring; for "Hollow Brush", take all rings
          const ringsToProcess =
            contourStyle === "Brush" ? [polygon[0]] : polygon;
```

```
  return ringsToProcess.map((ring) =>
    ring.map(([x, y]) => {
      const { x: cx, y: cy } = translateDisplayToWorld(
        new Point(x, y),
        canvasWidth,
        canvasHeight,
        translateX,
        translateY,
        positionRect,
        bodyHeight,
        bodyWidth
      );
      return [cx, cy];
    })
  );
});

//console.log("Existing polygons: ", existingPolygonsModified);
//  console.log("Brush polygons: ", brushPolygonsCT);

let unionedPolygons = [];
if (
  !existingPolygonsModified ||
  existingPolygonsModified.length === 0
) {
  unionedPolygons = brushPolygonsCT; // just take the brush directly
} else {
  if (isPointInside.status) {
    // Get the polygon that was intersected
    const insidePolygonIndices = isPointInside.polygonIndices;
```

```
  insidePolygonIndices.forEach((polygonIndex) => {
    const targetPolygon = existingPolygonsModified[polygonIndex];
    // Union only the intersected polygon with the brush polygons

    const selectiveUnionedPolygons = polygonClipping.union(
      targetPolygon,
      brushPolygonsCT
    );
    unionedPolygons = [
      ...existingPolygonsModified.slice(0, polygonIndex),
      ...selectiveUnionedPolygons,
      ...existingPolygonsModified.slice(polygonIndex + 1),
    ];
    existingPolygonsModified = unionedPolygons;
  });
}
if (!isPointInside.status && isShiftPressed) {
  unionedPolygons = [
    ...existingPolygonsModified,
    ...brushPolygonsCT,
  ]; //directly add the new polygons as they are draw as disjoint
} else {
  unionedPolygons = existingPolygonsModified.flatMap(
    (polygon, index) => {
      // Keep the skipped polygon as-is (if brush is inside polygon)
      if (isPointInside.polygonIndices.includes(index))
        return [polygon];

      // Check if this polygon intersects with brushPolygonsCT

      const intersected = polygonClipping.intersection(
```

```
        polygon,
        brushPolygonsCT
      );
      // If polygon doesn't intersect with brush - keep it as it is
      if (!intersected || intersected.length === 0)
        return [polygon];

      // If there's an intersection, replace polygon with (polygon - brush)
      const diff = polygonClipping.difference(
        polygon,
        brushPolygonsCT
      );
      return diff?.length > 0 ? diff : []; // skip empty diffs
    }
   );
  }
}

//console.log("Unioned polyogons: ", unionedPolygons);
const unionPolygonsFormatted = unionedPolygons.map((polygon) => {
  return polygon.map((ring) => {
    return ring.map(([x, y]) => {
      return { x, y };
    });
  });
});
//   console.log("Union polygons formatted: ", unionPolygonsFormatted);

const newSets = structureSets.map((set) =>
  set.structureSetUID === selectedStructureSetUID
    ? {
```

```
          ...set,
          structuresList: set.structuresList.map((st) =>
            st.structureID === selectedStructureUID
              ? {
                  ...st,
                  polygonsList: {
                    ...st.polygonsList,
                    [ZCoordinate]: unionPolygonsFormatted.flat(),
                  },
                }
              : st
          ),
        }
      : set
  );
  // console.log("New sets: ", newSets);

  setStructureSets(newSets);
  setBrushPolygons([]);
  unionRef.current = [];
 } else if (activeBrush && typeof activeBrush.mouseup === "function") {
  activeBrush.mouseup(event, ctx);
 }
} else if (isEraserEnabled && selectedStructureUID) {
 setIsErasing(false);
 setPrevPoint(null);
 if (!eraserRef.current) return;
 // Get the polygon list of the selected structure
 const selectedStructureSet = structureSets.find(
  (set) => set.structureSetUID === selectedStructureSetUID
 );
```

```
const selectedStructure = selectedStructureSet.structuresList.find(
  (structure) => structure.structureID === selectedStructureUID
);
const polygonsList =
  selectedStructure.polygonsList[ZCoordinate] || [];
const displayPolys = polygonsList.map((ring) =>
  ring.map((pt) => {
    const d = translateWorldToDisplay(
      new Point(pt.x, pt.y),
      canvasWidth,
      canvasHeight,
      translateX,
      translateY,
      positionRect,
      bodyHeight,
      bodyWidth
    );
    return [d.x, d.y];
  })
);
// console.log("Polygon list for erasing", polygonsList);
let updatedDisplayRings = [];
displayPolys.forEach((ring) => {
  let diff = [];
  try {
    diff = polygonClipping.difference([ring], eraserRef.current);
  } catch (err) {
    console.warn("Eraser-difference failed", err);
    diff = [];
  }
```

```
  if (diff && diff.length) {
    diff.forEach((poly) =>
      poly.forEach((r) => updatedDisplayRings.push(r))
    );
  }
});
//  console.log("Updated polygons: ", updatedDisplayRings);
const updatedCtPolygons = updatedDisplayRings.map((ring) =>
  ring.map(([x, y]) => {
    const w = translateDisplayToWorld(
      new Point(x, y),
      canvasWidth,
      canvasHeight,
      translateX,
      translateY,
      positionRect,
      bodyHeight,
      bodyWidth
    );
    return { x: w.x, y: w.y };
  })
);

const newSets = structureSets.map((set) =>
  set.structureSetUID === selectedStructureSetUID
    ? {
        ...set,
        structuresList: set.structuresList.map((st) =>
          st.structureID === selectedStructureUID
            ? {
                ...st,
```

```
                polygonsList: {
                  ...st.polygonsList,
                  [ZCoordinate]: updatedCtPolygons,
                },
              }
            : st
        ),
      }
    : set
  );

  setStructureSets(newSets);

  eraserRef.current = [];
  setEraserPolygons([]);
 }
 }
};

const handleKeyDown = (event) => {
 if (event.shiftKey) {
  setIsShiftPressed(true);
 }
 if (activeView !== "transversal") return;
 if (event.key === "ArrowUp" || event.key === "ArrowDown") {
  event.preventDefault();
  setCurrentZSlice((prevIndex) => {
   let newIndex = prevIndex;
   if (event.key === "ArrowDown" && prevIndex < zSlices - 1) {
    newIndex = prevIndex + 1;
   } else if (event.key === "ArrowUp" && prevIndex > 0) {
```

```
        newIndex = prevIndex - 1;
      }
      return Math.min(Math.max(newIndex, 0), zSlices - 1); // Clamp within valid range
    });
  }
 };
 const handleKeyUp = (event) => {
  if (!event.shiftKey) {
   setIsShiftPressed(false);
  }
 };

 canvas.addEventListener("wheel", handleWheel);
 canvas.addEventListener("mousedown", handleMouseDown);
 canvas.addEventListener("mouseup", handleMouseUp);
 canvas.addEventListener("mousemove", handleMouseMove);
 window.addEventListener("keydown", handleKeyDown);
 window.addEventListener("keyup", handleKeyUp);
 return () => {
  canvas.removeEventListener("wheel", handleWheel);
  canvas.removeEventListener("mousedown", handleMouseDown);
  canvas.removeEventListener("mouseup", handleMouseUp);
  canvas.removeEventListener("mousemove", handleMouseMove);
  window.removeEventListener("keydown", handleKeyDown);
  window.removeEventListener("keyup", handleKeyUp);
 };
}, [
 ZCoordinate,
 images,
 translateX,
 translateY,
```

isBrushEnabled,

isEraserEnabled,

eraserType,

contourStyle,

activeBrush,

isMouseDown,

isPanning,

selectedStructureUID,

canvasWidth,

canvasHeight,

isCrosshairEnabled,

isRulerEnabled,

isProtractorEnabled,

anglePointsCT,

currentAnglePoints,

isMeasureCircleEnabled,

isPixelProbeEnabled,

isTextAnnotationEnabled,

isDeleteAnnotationEnabled,

textAnnotations,

currentTextTail,

isRectangleProbeEnabled,

isEllipseProbeEnabled,

circlePointsCT,

currentCircleStart,

currentRectangleStart,

rectangleProbesCT,

ellipseProbesCT,

currentEllipseStart,

openAutoSegDialog,

dragging,

resizing,

ROI3dCT?.x,

ROI3dCT?.y,

ROI3dCT?.width,

ROI3dCT?.height,

rulerPointsCT,

isRulerDrag,

rulerStartPoint,

currentMouse,

hoveredRulerIndex,

resizeRulerIndex,

movedRulerIndex,

isRulerMoving,

rulerStartMovePoint,

hoveredProtractorIndex,

resizeProtractorInfo,

moveProtractorArmInfo,

lastPhysicalPoint,

draggingType,

hoveredCircleIndex,

movedCircleIndex,

resizeCircleIndex,

circleStartMovePoint,

isCircleMoving,

hoveredRectangleIndex,

movedRectangleIndex,

resizeRectangleIndex,

rectangleStartMovePoint,

isRectangleMoving,

hoveredEllipseIndex,

movedEllipseIndex,

```
  resizeEllipseIndex,
  ellipseStartMovePoint,
  isEllipseMoving,
  pixelProbesCT,
  hoveredPixelIndex,
  movedPixelProbeIndex,
  isMovingPixelProbe,
  previewPixelPoint,
  hoveredTextAnnotationIndex,
  resizedTextAnnotationsIndex,
  movedTextAnnotationIndex,
  textStartMovePoint,
  isTextMoving,
  resizingTextPointType,
  draggedRlAp,
  hoveredRlAp,
  isBrushing,
  prevPoint,
  isErasing,
  isPointInside,
  activeView,
]);

const handleClick = (event) => {
  const canvas = canvasRef.current;
  const ctx = canvas.getContext("2d");

  const rect = canvas.getBoundingClientRect(); // Get canvas position in the viewport

  const canvasX = event.clientX - rect.left; // X coordinate relative to canvas
  const canvasY = event.clientY - rect.top; // Y coordinate relative to canvas
```

```
  const canvasWidth = rect.width;
  const canvasHeight = rect.height;

  const isSeedPointInside =
    canvasX >= 0 &&
    canvasX <= canvasWidth &&
    canvasY >= 0 &&
    canvasY <= canvasHeight;

  eventBus.emit("transversalImgDataReady", {
    transversalImgData: transversalImgDataRef.current,
    seedPoint: { x: Math.round(canvasX), y: Math.round(canvasY) },
    transformParameters: {
      TranslateX: translateX,
      TranslateY: translateY,
      PositionRect: positionRect,
      BodyHeight: bodyHeight,
      BodyWidth: bodyWidth,
    },
  });

  renderCanvas();

  ctx.beginPath();
  ctx.arc(canvasX, canvasY, 5, 0, 2 * Math.PI); // 5px radius dot
  ctx.fillStyle = "red"; // Color of dot
  ctx.fill();
};

return (
```

```
<div style={{ position: "relative", width: "fit-content" }}>
  <div
    style={{
      display: "flex",
      justifyContent: "space-between",
      alignItems: "center",
      width: "100%",
    }}
  >
    {/* Z coordinate on the left */}
    <h4 style={{ margin: "8px", color: "lightgray", fontWeight: "300" }}>
      Z: ({zSlices - 1 - currentZSlice}/{zSlices - 1}){" "}
      {zList[currentZSlice]} mm
    </h4>

    {/* X/Y coordinates in the center */}
    {CTCoordinate && isCrosshairEnabled && currentHU && (
      <div style={{ textAlign: "center", flex: 1 }}>
        <h4
          style={{ margin: "8px", color: "lightgray", fontWeight: "300" }}
        >
          ({CTCoordinate.x}, {CTCoordinate.y}, {zList[currentZSlice]}){" "}
          {currentHU}
        </h4>
      </div>
    )}
  </div>

  <canvas
    ref={canvasRef}
    className="canvas-container"
```

```
  width={canvasWidth}
  height={canvasHeight}
/>

{/* Bottom-left image: directional vector */}
<img
  src="/assets/DirectionalVectorsAxial.png"
  alt="orientation"
  style={{
    position: "absolute",
    bottom: "-100px",
    left: "12px",
    width: "50px",
    height: "50px",
    pointerEvents: "none", // important so canvas still receives mouse events
  }}
/>

{/* Bottom-right image: mascot */}
<img
  src="/assets/mascot_topview.png"
  alt="orientation"
  style={{
    position: "absolute",
    bottom: "-100px",
    right: "12px",
    width: "50px",
    height: "50px",
    pointerEvents: "none", // important so canvas still receives mouse events
  }}
/>
```

```
    </div>
  );
};

export default TransversalView;
```