```
// Define the method for finding intersections between a line and a circle
public static int FindLineCircleIntersections(Vector3 circleCenter, double radius,
                                              Vector3 lineStart, Vector3 lineEnd,
                                              out Vector3 intersection1, out Vector3 intersection2)
{
    // Declare variables for calculations
    double dx, dy, A, B, C, det, t;

    // Calculate delta x and delta y which are the differences in the x and y coordinates
    // between the end and start of the line
    dx = lineEnd.X - lineStart.X;
    dy = lineEnd.Y - lineStart.Y;

    // Compute the coefficients A, B, and C of the quadratic equation representing
    // the intersection points between the line and the circle
    A = dx * dx + dy * dy;
    B = 2 * (dx * (lineStart.X - circleCenter.X) + dy * (lineStart.Y - circleCenter.Y));
    C = (lineStart.X - circleCenter.X) * (lineStart.X - circleCenter.X) +
        (lineStart.Y - circleCenter.Y) * (lineStart.Y - circleCenter.Y) -
        radius * radius;

    // Calculate the determinant to determine if there are real solutions to the quadratic equation
    det = B * B - 4 * A * C;

    // Check if there are no real solutions or if A is too close to 0 for a valid solution
    if ((A <= 0.0000001) || (det < 0))
    {
        // If no real solutions, set intersection points to NaN (Not a Number) and return 0 intersections
        intersection1 = new Vector3(float.NaN, float.NaN, float.NaN);
        intersection2 = new Vector3(float.NaN, float.NaN, float.NaN);
        return 0;
    }
    else if (det == 0)
    {
```

```
        // If the determinant is 0, there's exactly one solution on the line (tangent to circle)
        t = -B / (2 * A);
        // Compute this single intersection point using the parameter t
        intersection1 = new Vector3((float)(lineStart.X + t * dx), (float)(lineStart.Y + t * dy), lineStart.Z);
        // The second intersection point is not applicable here, set to NaN
        intersection2 = new Vector3(float.NaN, float.NaN, float.NaN);
        return 1; // Return 1 to indicate one intersection point
    }
    else
    {
        // Two solutions exist if determinant is positive (line intersects circle at two points)
        t = (float)((-B + Math.Sqrt(det)) / (2 * A));
        // Calculate the first intersection point using one root of the quadratic equation
        intersection1 = new Vector3((float)(lineStart.X + t * dx), (float)(lineStart.Y + t * dy), lineStart.Z);
        // Calculate the second intersection point using the other root
        t = (float)((-B - Math.Sqrt(det)) / (2 * A));
        intersection2 = new Vector3((float)(lineStart.X + t * dx), (float)(lineStart.Y + t * dy), lineStart.Z);
        return 2; // Return 2 to indicate two intersection points
    }
}

// Define the method to determine if a point is located on a given arc
public static bool IsPointOnArc(Vector2 point, Vector2 arcCenter, double arcRadius, double startAngle, double endAngle)
{
    double buffer = 0.000001; // Define a small tolerance to account for floating-point precision issues

    // Calculate the distance from the point to the center of the arc
    double dist = Vector2.Distance(point, arcCenter);
    // If the distance is not approximately equal to the arc's radius, the point is not on the arc
    if (Math.Abs(dist - arcRadius) > buffer)
    {
        return false; // Point is not on the circle defined by the arc
    }
```

```
    // Calculate the angle from the arc center to the point
    double angleOfPoint = Math.Atan2(point.Y - arcCenter.Y, point.X - arcCenter.X) * (180.0 / Math.PI);
    // Normalize the angle to be within the range [0, 360]
    angleOfPoint = (angleOfPoint < 0) ? 360 + angleOfPoint : angleOfPoint;

    // Normalize the start and end angles of the arc to be within [0, 360]
    startAngle = startAngle % 360;
    endAngle = endAngle % 360;

    // Determine if the angleOfPoint lies within the sweep of the arc
    if (startAngle < endAngle)
    {
        // The arc does not cross the 0 degree line, so simply check if the point's angle is between the start and end angles
        return startAngle <= angleOfPoint && angleOfPoint <= endAngle;
    }
    else if (startAngle > endAngle)
    {
        // The arc crosses the 0 degree line, check if the point's angle is either greater than startAngle or less than endAngle
        return angleOfPoint >= startAngle || angleOfPoint <= endAngle;
    }
    else // case where startAngle == endAngle, meaning it's a full circle or just a point
    {
        // For a full circle, any point on the circle is on the arc; for a point, the angle will match exactly
        return Math.Abs(startAngle - angleOfPoint) < buffer;
    }
}

// Extension method for the Vector2 class to determine if a point is between two other points along a straight line
public static class Vector2Extensions
{
    // Extension method to check if a Vector2 point lies between two other Vector2 points
    public static bool IsBetween(this Vector2 point, Vector2 start, Vector2 end)
    {
        // Calculate vectors from start to point and start to end
```

```
        Vector2 startToPoint = point - start;
        Vector2 startToEnd = end - start;

        // Compute the dot product of the two vectors
        double dotProduct = (startToPoint.X * startToEnd.X) + (startToPoint.Y * startToEnd.Y);
        // Calculate the squared length of the startToEnd vector
        double squaredLength = (startToEnd.X * startToEnd.X) + (startToEnd.Y * startToEnd.Y);

        // The point is between start and end if the dot product is non-negative
        // and less than or equal to the squared length of the startToEnd vector
        return dotProduct >= 0 && dotProduct <= squaredLength;
    }
}
```

```
 /// good!!!
// Load the DXF file from a specified location on the file system
DxfDocument loaded = DxfDocument.Load("C:\\Users\\Dave\\Downloads\\new block\\new\\arcline.dxf");
// Set the active layout to Model Space within the DXF document
loaded.Entities.ActiveLayout = netDxf.Objects.Layout.ModelSpaceName;
// Retrieve the collection of entities present in the Model Space block (default space for drawing)
EntityCollection entitiesBlocks = loaded.Blocks[Block.DefaultModelSpaceName].Entities;

// Define an amount by which line entities will be extended (not currently used in the logic)
double extensionAmount = 0.1;

// Initialize lists to store entities that intersect lines and arcs
List<EntityObject> intersectedLines = new List<EntityObject>();
List<EntityObject> intersectedArcs = new List<EntityObject>();

// Loop through each line entity within Model Space
foreach (Line line in entitiesBlocks.OfType<Line>())
```

```
{
// Extend both ends of the line in 2D
            //Convert the Vector3 to a Vector2 and extend
            Vector2 startPoint2D = new Vector2(line.StartPoint.X, line.StartPoint.Y);
            Vector2 endPoint2D = new Vector2(line.EndPoint.X, line.EndPoint.Y);
            //Not used atm and not needed, a the line projects to infinity
            //startPoint2D = startPoint2D - extensionAmount * new Vector2(line.Direction.X, line.Direction.Y);
            //endPoint2D = endPoint2D + extensionAmount * new Vector2(line.Direction.X, line.Direction.Y);

            // Update the 3D line endpoints while preserving the original Z value
            //Not used atm and not needed, a the line projects to infinity
            //line.StartPoint = new Vector3(startPoint2D.X, startPoint2D.Y, line.StartPoint.Z);
            //line.EndPoint = new Vector3(endPoint2D.X, endPoint2D.Y, line.EndPoint.Z);
}

// Loop through each arc entity within Model Space
foreach (Arc myarc in entitiesBlocks.OfType<Arc>())
{
    // Logic for extending the arc's start and end angles is commented out
    // It indicates a future provision for arc extension if necessary
    // Currently, arcs remain unaltered as the extension is not applied
}

// A list to keep track of unique intersection points as strings formatted to three decimal places
List<string> uniqueIntersections = new List<string>();

// A HashSet to store tuples consisting of entity handles, ensuring each intersection pair is only printed once
HashSet<Tuple<string, string>> printedPairs = new HashSet<Tuple<string, string>>();

// Nested loops to compare each entity with every other entity to find intersections
for (int i = 0; i < entitiesBlocks.Count; i++)
{
    for (int j = i + 1; j < entitiesBlocks.Count; j++)
    {
```

```
// If both entities are lines, check for intersection between them
if (entitiesBlocks[i] is Line line1 && entitiesBlocks[j] is Line line2)
{
    // Use a helper method to find the intersection point between two infinite lines
    Vector2 intersection = MathHelper.FindIntersection(
        line1.StartPoint.ToVector2(),
        line1.Direction.ToVector2(),
        line2.StartPoint.ToVector2(),
        line2.Direction.ToVector2(),
        1e-6);

    // Check if the intersection point is valid (not NaN)
    if (!double.IsNaN(intersection.X) && !double.IsNaN(intersection.Y))
    {
        // Ensure the found intersection point is within the segments of both lines
        bool isWithinLine1 = intersection.IsBetween(line1.StartPoint.ToVector2(), line1.EndPoint.ToVector2());
        bool isWithinLine2 = intersection.IsBetween(line2.StartPoint.ToVector2(), line2.EndPoint.ToVector2());

        // If the intersection is valid for both line segments
        if (isWithinLine1 && isWithinLine2)
        {
            // Format the intersection point into a string identifier
            string identifier = $"{intersection.X:F3},{intersection.Y:F3}";

            // Check if this intersection is unique (has not been recorded yet)
            if (!uniqueIntersections.Contains(identifier))
            {
                // Add the lines and the identifier to the respective lists for tracking
                intersectedLines.Add(line1);
                intersectedLines.Add(line2);
                uniqueIntersections.Add(identifier);

                // Output the details of the intersection to the console
                Console.WriteLine($"Intersection at ({intersection.X:F3},{intersection.Y:F3}) between Line: {line1.Handle} and
```

```
Line: {line2.Handle}");
                    }
                }
            }
        }
        // If one entity is an arc and the other is a line, check for intersection between them
        else if (entitiesBlocks[i] is Arc arc && entitiesBlocks[j] is Line line)
        {
            // Prepare variables to store potential intersection points
            Vector3 intersection1, intersection2;
            // Find the intersections between the line and the circle defined by the arc's center and radius
            int intersections = FindLineCircleIntersections(
                new Vector3((float)arc.Center.X, (float)arc.Center.Y, (float)arc.Center.Z),
                arc.Radius,
                line.StartPoint,
                line.EndPoint,
                out intersection1, out intersection2);

            // Iterate through the found intersection points
            for (int k = 1; k <= intersections; k++)
            {
                // Select the first or second intersection point based on the loop's iteration
                Vector2 intersectionPoint = k == 1 ?
                    new Vector2(intersection1.X, intersection1.Y) :
                    new Vector2(intersection2.X, intersection2.Y);

                // Check if the intersection point lies on the arc segment and the line segment
                if (IsPointOnArc(intersectionPoint, arc.Center.ToVector2(), arc.Radius, arc.StartAngle, arc.EndAngle) &&
                    intersectionPoint.IsBetween(line.StartPoint.ToVector2(), line.EndPoint.ToVector2()))
                {
                    // Format the intersection point into a string identifier
                    string identifier = $"{intersectionPoint.X:F3},{intersectionPoint.Y:F3}";

                    // Ensure this intersection has not been recorded before
```

```
                if (!uniqueIntersections.Contains(identifier))
                {
                    // Output the details of the intersection to the console
                    Console.WriteLine($"Intersection at ({intersectionPoint.X:F3},{intersectionPoint.Y:F3}) between Line:
{line.Handle} and Arc: {arc.Handle}");
                    // Add the identifier to the list of unique intersections
                    uniqueIntersections.Add(identifier);
                }
            }
        }
    }
    // ... Additional logic for other types of entity intersections could be added here
  }
}

// Save the modified DXF file to a specified location on the file system
loaded.Save("C:\\Users\\Dave\\Downloads\\new block\\new\\1a.dxf");
```