Design for Rectilinear Edge Routing

Contents

[Overview 4](#_Toc298426878)

[Terminology 5](#_Toc298426879)

[Key External classes used 7](#_Toc298426880)

[ApproximateComparer.cs 7](#_Toc298426881)

[BinaryHeapWithComparer 7](#_Toc298426882)

[EdgeGeometry.cs 7](#_Toc298426883)

[GenericBinaryHeapPriorityQueueWithTimestamp 7](#_Toc298426884)

[Point.cs 7](#_Toc298426885)

[PortEntryOnCurve.cs 7](#_Toc298426886)

[RbTree.cs 7](#_Toc298426887)

[RectangleNode.cs 7](#_Toc298426888)

[Rectangle.cs 7](#_Toc298426889)

[Shape.cs 7](#_Toc298426890)

[TransientVisibilityEdge.cs 8](#_Toc298426891)

[VisibilityGraph.cs, VisibilityVertex.cs, VisibilityEdge.cs 8](#_Toc298426892)

[RectilinearEdgeRouter classes 8](#_Toc298426893)

[External interface 8](#_Toc298426894)

[RectilinearEdgeRouter.cs 8](#_Toc298426895)

[RectilinearInteractiveEditor.cs 8](#_Toc298426896)

[Internal implementation 8](#_Toc298426897)

[Obstacle management 8](#_Toc298426898)

[Obstacle.cs 8](#_Toc298426899)

[ObstacleTree.cs 8](#_Toc298426900)

[Preprocessing overlapping obstacles 8](#_Toc298426901)

[Clump.cs 8](#_Toc298426902)

[OverlapConvexHull.cs 8](#_Toc298426903)

[VisibilityGraph creation 9](#_Toc298426904)

[ScanSegment.cs 9](#_Toc298426905)

[ScanSegmentTree.cs 9](#_Toc298426906)

[NeighborSides.cs 9](#_Toc298426907)

[VisibilityVertexRectilinear.cs 9](#_Toc298426908)

[VisibilityGraphGenerator.cs 9](#_Toc298426909)

[FullVisibilityGraphGenerator.cs 9](#_Toc298426910)

[LookaheadScan.cs 9](#_Toc298426911)

[SegmentIntersector.cs 9](#_Toc298426912)

[EnumeratorWrapper.cs 9](#_Toc298426913)

[ScanSegmentVector.cs 10](#_Toc298426914)

[Scanline execution 10](#_Toc298426915)

[RectilinearScanLine.cs 10](#_Toc298426916)

[BasicObstacleSide.cs 10](#_Toc298426917)

[LowObstacleSide.cs 10](#_Toc298426918)

[HighObstacleSide.cs 10](#_Toc298426919)

[Event handling 10](#_Toc298426920)

[EventQueue.cs 10](#_Toc298426921)

[AxisCoordinateEvent.cs 10](#_Toc298426922)

[BasicVertexEvent.cs 10](#_Toc298426923)

[OpenVertexEvent.cs 11](#_Toc298426924)

[CloseVertexEvent.cs 11](#_Toc298426925)

[LowBendVertexEvent.cs 11](#_Toc298426926)

[HighBendVertexEvent.cs 11](#_Toc298426927)

[BasicReflectionEvent.cs 11](#_Toc298426928)

[LowReflectionEvent.cs 11](#_Toc298426929)

[HighReflectionEvent.cs 11](#_Toc298426930)

[Groups 11](#_Toc298426931)

[GroupBoundaryCrossing.cs 11](#_Toc298426932)

[GroupBoundaryCrossingMap.cs 11](#_Toc298426933)

[PointAndCrossings.cs 11](#_Toc298426934)

[PointAndCrossingsList.cs 12](#_Toc298426935)

[Internal Utilities 12](#_Toc298426936)

[PointComparer.cs 12](#_Toc298426937)

[ScanDirection.cs 12](#_Toc298426938)

[SpliceUtility.cs 12](#_Toc298426939)

[StaticGraphUtility.cs 12](#_Toc298426940)

[Shortest Path generation 12](#_Toc298426941)

[Ports and port splicing 12](#_Toc298426942)

[ObstaclePort.cs 12](#_Toc298426943)

[ObstaclePortEntrance.cs 12](#_Toc298426944)

[FreePoint.cs 12](#_Toc298426945)

[WaypointPort.cs 13](#_Toc298426946)

[PortManager.cs 13](#_Toc298426947)

[TransientGraphUtility.cs 13](#_Toc298426948)

[ShortestPaths 13](#_Toc298426949)

[MsmtRectilinearPath.cs 13](#_Toc298426950)

[SsstRectilinearPath.cs 13](#_Toc298426951)

[VertexEntry.cs 13](#_Toc298426952)

[Details of Selected Implementation Areas 13](#_Toc298426953)

[Creating and using a RectilinearEdgeRouter 13](#_Toc298426954)

[Preprocessing Overlapping Obstacles into Clumps and Convex Hulls 14](#_Toc298426955)

[VisibilityGraph Generation 16](#_Toc298426956)

[Creating ScanSegments: Full Visibility Graph 16](#_Toc298426957)

[Creating the ScanSegments 16](#_Toc298426958)

[Overlapped ScanSegments 17](#_Toc298426959)

[Reflection Segments 18](#_Toc298426960)

[Creating ScanSegments: Sparse Visibility Graph 21](#_Toc298426961)

[Accumulating Vertex Coordinates 22](#_Toc298426962)

[Creating the ScanSegments 22](#_Toc298426963)

[Overlapped ScanSegments 22](#_Toc298426964)

[Generating Sparse Intersections By Binary Division 23](#_Toc298426965)

[Port Visibility Splicing 23](#_Toc298426966)

[Creating the initial VisibilityVertex 24](#_Toc298426967)

[Extending the Edge Chain From the Initial Vertex 25](#_Toc298426968)

[LimitPortVisibilitySpliceToEndpointBoundingBox 26](#_Toc298426969)

[Direct Visibility Intersection to Bypass Port Visibility Splicing 26](#_Toc298426970)

[Shortest-Path generation 27](#_Toc298426971)

[SsstRectilinearPath Retaining Multiple Directional Incoming Paths at Vertices 27](#_Toc298426972)

[Routing a Path With Waypoints 27](#_Toc298426973)

[Group Boundary Crossings 27](#_Toc298426974)

[Out-of-Bounds (OOB) FreePoints and Waypoints 28](#_Toc298426975)

[Port Entries 29](#_Toc298426976)

[Nudging 29](#_Toc298426977)

[Main steps of nudging 30](#_Toc298426978)

[Nudging in direction 31](#_Toc298426979)

[PathRefiner.RefinePaths(IEnumerable<Path> paths, bool mergePaths) 32](#_Toc298426980)

[Testing 32](#_Toc298426981)

[Unit Testing Architecture 32](#_Toc298426982)

[Overriding default values 32](#_Toc298426983)

[Datafiles 34](#_Toc298426984)

[Creating and Regenerating 34](#_Toc298426985)

[DebugDevTrace build configuration 35](#_Toc298426986)

[#define TEST\_MSAGL 35](#_Toc298426987)

[TestRectilinear.exe 35](#_Toc298426988)

[Useful Parameters For Test Creation and Debugging 35](#_Toc298426989)

[Scripts 37](#_Toc298426990)

[The Display 37](#_Toc298426991)

[TestForGDI.exe 39](#_Toc298426992)

[Debugging Tips 39](#_Toc298426993)

[Conditional Breakpoints 39](#_Toc298426994)

[Reducing Datafile failures 39](#_Toc298426995)

[Investigating failures 39](#_Toc298426996)

[Debugging failures 1: Port visibility splicing (TransientVisibilityEdge creation) 40](#_Toc298426997)

[Debugging failures 2: Path crosses an Obstacle 40](#_Toc298426998)

[Other test notes 41](#_Toc298426999)

[Merging with Progression 41](#_Toc298427000)

[Unimplemented areas 41](#_Toc298427001)

[Out of Scope 41](#_Toc298427002)

[Waypoints With Dimension 41](#_Toc298427003)

[Potential Improvements 42](#_Toc298427004)

[Additional Nudging pass to utilize between-ConvexHull space after staircase removal 42](#_Toc298427005)

[Splicing out from overlapped obstacles to free space outside clump 42](#_Toc298427006)

[Bend reduction in Sparse VisibilityGraph 42](#_Toc298427007)

[Eliminate Shallow Bends in Obstacle Sides 43](#_Toc298427008)

[Performance 43](#_Toc298427009)

[Direct visibility intersection for Waypoints 43](#_Toc298427010)

[Combine all Shortest-Path Sources into a single pass 43](#_Toc298427011)

[Make a Lookup Table for Direction-converting CompassVector methods 43](#_Toc298427012)

[Cache the First Splice Intersection in ObstaclePortEntrance and FreePoint 43](#_Toc298427013)

[Reduce GC impact of Port Splicing and Shortest Paths 44](#_Toc298427014)

# Overview

Also known as orthogonal edge routing, Rectilinear Edge Routing routes paths between obstacles as segments that are parallel to the X or Y axis of the graph. Our implementation is based upon the paper "Orthogonal Connector Routing" by Wybrow et al. (<http://www.csse.monash.edu.au/~mwybrow/papers/wybrow-gd-2009.pdf>), with several additions such as:

* Support for overlapping obstacles
* Waypoints (specific points that the path must pass through on the way from source to target)
* Groups (boundaries around groups of obstacles that paths must not cross unless connecting cross-group obstacles)
* Port Entries (also known as connection regions; areas along an obstacle's surface that define the ranges within which paths can connect to that obstacle).

Although very accurate, this algorithm creates an n-squared order of vertices and edges, so we also added an alternative compact visibility-graph implementation which we refer to as “Sparse Visibility graph”, based upon "Rectilinear shortest paths through polygonal obstacles in O(n (log n)2) time" by Clarkson et al. This implementation creates VisibilityVertices based upon intersections in intervals resulting from the binary division of visibility distance rather than at every possible intersection. This handles the same obstacle and group layouts that the full visibility graph does. Sparse mode can result in the selection of paths with additional bends, because the vertices at the optimal bend points may not be in the graph. These routings are still quite reasonable and in many cases this is acceptable given the improvements to speed and memory usage, which may be significant for large graphs; for example, GeometryGraph\_1138bus.msagl.geom has 1.25M VisibilityVertices and 2.5M VisibilityEdges with FullVg, while SparseVg has 62K VisibilityVertices and 113K VisibilityEdges.

This document discusses high-level design and flow between the classes. Specifics of individual class implementation are left to the detailed source-code comments.

# Terminology

A number of terms have specific meanings within Rectilinear Edge Routing (RER). Some of them correspond to class names and are “forward-defined” here for readability.

Clump: A group of two or more overlapping rectangular obstacles. No convex hull is formed in this case; the obstacles remain separate and their sides are appropriately ordered in the scanline.

Convex Hull: A convex hull around two or more overlapping non-rectangular obstacles. Convex hulls are considered a single obstacle in the visibility graph; each obstacle (“sibling”) inside the convex hull has the same polyline in the visibility graph.

Edge: this may be either a connection between two VisibilityVertex instances (in which case it is a VisibilityEdge) or an term for a partial or complete path between a source and a target.

Extreme Vertex: This is an obstacle vertex that is the furthest vertex in one direction or the other along the horizontal or vertical axis – i.e. its X is the lowest or highest X, and similarly for Y. A vertex may be an extreme vertex on both axes.

Group: An obstacle that specifically contains other obstacles and prevents routed edges from traversing across the group boundaries, unless the source and target of the edge have different group memberships.

High direction: This means looking along an axis in the direction in which the value of the coordinate increases; in other words, to the right for X, and upward for Y (because we use Cartesian coordinates).

HighObstacleSide: An obstacle side on the highest scanline-parallel coordinate side. For example, given a diamond, the two sides to the right are on the high side for horizontal scan, and the two sides on the top are on the high side for a vertical scan.

Low direction: This means looking along an axis in the direction in which the value of the coordinate decreases; in other words, to the left for X, and downward for Y (because we use Cartesian coordinates).

LowObstacleSide: An obstacle side on the lowest scanline-parallel coordinate side. For example, given a diamond, the two sides to the left are on the low side for horizontal scan, and the two sides on the bottom are on the low side for a vertical scan.

Obstacle: This is an object on the graph. In Visual Studio it is derived from a node; the term “node” is not used in RER. An Obstacle instance contains a Shape instance and carries RER-specific information.

Overlapped: when referring to obstacles, it means one obstacle lies on top of the other. When used in reference to VisibilityEdges or ScanSegments, it means that it passes through an overlapped obstacle and thus has a higher weight to discourage taking it.

PaddedPolyline: An obstacle’s polyline that wraps its Shape’s curve with the padding specified to the RER constructor.

Port: A specification of a source or target for routing. Normally Ports are associated with an obstacle. Their visibility is dynamically added to the graph on a per-path basis after the initial VisibilityGraph is generated. This trades some performance for a smaller visibility graph (especially for a Full Visibility Graph), as well as providing the flexibility to add and remove ports without having to re-create the entire VisibilityGraph.

Reflection: For RER this refers to a “bounce” between two angled obstacle sides in the Full Visibility Graph. This allows paths to pass between close non-rectangular obstacles, but because multiple bounces (staircases) can be undesirable, a Reflection has a higher weight than a normal path.

RER: Acronym for RectilinearEdgeRouter, used both for the class name and the as a reference to rectilinear edge routing in general.

Scan Direction: The direction when looking along the sweep line. When we are moving a horizontal line vertically (from lower to higher Y), then that is a Vertical sweep and a Horizontal scan, because we are sweeping up and scanning to the left or right. Similarly, moving a vertical line from low to high X is a Horizontal sweep and a Vertical scan.

ScanSegment: An extension of visibility from one point to the other. It usually spans a large number of collinear VisibilityEdges and is used both as a VisibilityVertex generator in Full Visibility Graph, and as an indexing method to acquire the initial vertex for port visibility splicing.

Shape: This is a curve for one obstacle, and may contain parents or children if the shape is a group or a member of a group.

Splice: Used in conjunction with Port visibility, this adds VisibilityVertexes and VisibilityEdges to the graph to provide direct access to a port.

Vertex: This is sometimes ambiguous between an Obstacle vertex (a point on the obstacle’s VisibilityPolyline) and a VisibilityVertex, so be sure to know which context it is used in.

VisibilityPolyline: An obstacle’s polyline in the visibility graph, which may be the PaddedPolyline or may be a convex hull.

# Key External classes used

##### ApproximateComparer.cs

Provides a number of static utility functions for comparing points and doubles, and defines the constants for values such as DistanceEpsilon (the granularity of axis coordinate differences) and IntersectionEpsilon (the distance within which we consider two intersections to be the same, accounting for floating-point indeterminacy).

##### BinaryHeapWithComparer

Provides the priority queue used by EventQueue (for scanline advancement).

##### EdgeGeometry.cs

Defines a SourcePort and a TargetPort for a path. Its PathPoints member is populated by ShortestPaths, and its Curve by the Nudger.

##### GenericBinaryHeapPriorityQueueWithTimestamp

Provides the priority queue for SsstRectilinear (Shortest Paths).

##### Point.cs

Manages a single X,Y coordinate pair.

##### PortEntryOnCurve.cs

Defines a PortEntry that contains one or more Spans which are pairs of curve parameters which define the allowable connection region.

##### RbTree.cs

Red-black tree, used by ScanSegmentTree and RectilinearScanLine.

##### RectangleNode.cs

A node in a rectangle hierarchy, containing user data. RER uses it for its ObstacleTree. RectangleNode defines a rectangular decomposition of the space for fast searching, first dividing it into large rectangles and then dividing further until arriving at leaf nodes, which are single rectangles defined for a user data item. RectangleNode provides a number of functions to search and walk the tree; these methods take delegates which indicate whether the “current” RectangleNode in the traversal satisfies some criteria.

##### Rectangle.cs

The class that manages an axis-parallel rectangle, exposing members such as Left, Right, Top, Bottom, LeftTop, LeftBottom, RightTop, RightBottom. A number of constructors are provided as are methods for common operations such as adding points to expand the rectangle or growing or shrinking it by padding.

##### Shape.cs

A shape on the graph, corresponding to a “node” with a curve (the unpadded curve in RER) and the Ports collection. Also may contain parents and children; if it contains children it is a group.

##### TransientVisibilityEdge.cs

A subclass of VisibilityEdge that is used to add temporary edges to the VisibilityGraph (in RER, this happens in Port visibility splicing).

##### VisibilityGraph.cs, VisibilityVertex.cs, VisibilityEdge.cs

The Visibility Graph; in RER, some subset of the intersection of visibility between obstacles on perpendicular axes.

# RectilinearEdgeRouter classes

Here is a summary of the classes and their relationships in RER (other than Nudger) along with a brief description. Subsequent sections will build upon this to provide more detailed descriptions of various implementation details (for example, details of the flow of Reflection generation).

## External interface

These are the files that are used directly by external callers.

##### RectilinearEdgeRouter.cs

The main class of the router. It contains methods for adding and removing obstacles.

##### RectilinearInteractiveEditor.cs

This is a utility class that wraps creation of a Rectilinear Edge Router.

## Internal implementation

### Obstacle management

Obstacles are what we must route between and around. This section describes the major classes in that regard.

##### Obstacle.cs

This is a single object on the graph, and supplies methods for carrying the corresponding InputShape, creating the PaddedPolyline, and indirecting the VisibilityPolyline to the PaddedPolyline or ConvexHull as appropriate.

##### ObstacleTree.cs

This contains the rectangle hierarchy of the obstacles in the visibility graph, as well as providing numerous utility functions to operate upon that hierarchy (such as restricting a visibility segment’s length to not run into obstacles).

#### Preprocessing overlapping obstacles

Overlapping obstacles are handled in one of two ways: if they are “close enough” to rectangular, then their padded polyline is a rectangle, and they are allowed to overlap directly, forming a Clump. Otherwise, the convex hull that contains them is calculated and used in the visibility graph.

##### Clump.cs

A Clump is two or more overlapping rectangular obstacles. Because there are no angled side intersections, we can handle this reliably in the scanline and not suffer from convex hull artifacts. (“Clump” may seem a dull name but “cluster” was already taken for a different purpose).

##### OverlapConvexHull.cs

If two or more obstacles overlap and any are not rectangular, they form a Convex Hull of their combined padded polylines. This can entail artifacts due to transitive overlaps where the convex hull overlaps an obstacle that does not actually overlap any of the obstacles within the hull.

### VisibilityGraph creation

The visibility graph consists of VisibilityVertices at the intersections of line segments that are adjacent to obstacles, and the VisibilityEdges that connect them.

##### ScanSegment.cs

A ScanSegment is the longest visibility between two points on the graph. It usually spans multiple collinear VisibilityEdges. For the Full Visibility Graph, horizontal and vertical ScanSegments are generated and intersected as in Wybrow.

##### ScanSegmentTree.cs

This wraps an RBTree and provides “indexing” to find the nearest ScanSegments in the horizontal and vertical directions and walk them to find the first point for splicing Port visibility into the graph. Also, during Full Visibility Graph generation, it is used to search for existing segments for Reflections.

##### NeighborSides.cs

This wraps the next neighbor to each side as well as whether there is an obstacle side before it that indicates we are potentially leaving an overlapped segment for a non-overlapped on (for Full VisibilityGraph).

##### VisibilityVertexRectilinear.cs

This extends VisibilityVertex to support Shortest-Paths.

##### VisibilityGraphGenerator.cs

This is the base VisibilityGraph generator providing a number of common functions and virtual methods that are overridden for the Full and Sparse implementations. For ease of implementation there are still some things in it (such as Reflections) that are only used by the Full VisibilityGraph generator; those virtual methods are overridden to simply Assert fail for Sparse.

##### FullVisibilityGraphGenerator.cs

The specialization of VisibilityGraphGenerator for the full (Wybrow) visibility graph. As in that paper, the approach is to generate ScanSegments to the fullest extent possible at every extreme obstacle vertex for each axis, then intersecting them. This implementation also supports Reflections.

##### LookaheadScan.cs

This stores the sites for possible Reflections; the term “lookahead” refers to the fact that until we get to the next obstacle side “above” that point, we don’t know if we have a reflection situation – another obstacle could open up and intercept the reflection (and in turn generate its own).

##### SegmentIntersector.cs

This class performs the intersection of horizontal with vertical ScanSegments.

SparseVisibilityGraphGenerator.cs

The specialization of VisibilityGraphGenerator for the Sparse (Clarkson) visibility graph. As in that paper, the approach is to determine the vertex coordinates. Unlike that paper, we only need the extreme vertices, and we do not generate angled VisibilityEdges along obstacle sides (nor are Reflections done, so longer paths will be chosen).

##### EnumeratorWrapper.cs

Wraps an IEnumerator to provide information such as whether it has been positioned to the first item or after the last one.

##### ScanSegmentVector.cs

This is the vector of ScanSegmentVectorItems along a given axis, where each coordinate has its own item.

ScanSegmentVectorItem.cs

A coordinate along the axis which has visibility segments. Its name comes from the fact that it generates ScanSegments along that coordinate. It supplies the logic for adding points to the appropriate (overlapped or not) ScanSegment along that coordinate and for creating the ScanSegments and their group crossings.

### Scanline execution

For both Full and Sparse VisibilityGraph, we run a scanline (sweepline) to determine obstacle coordinates and visibility extension. The “direction” of the scanline is the direction we scan along, not the direction of the sweep; thus the horizontal scanline is along the X axis. The scanline maintains an RBTree of the sides of obstacles (including sentinels which are outside the graph boundaries). Because we do not support intersecting angled sides in the scanline (we preprocess those into convex hulls), once an obstacle side is in the scanline, it never needs to be reordered.

##### RectilinearScanLine.cs

The scanline implementation, managing the insertion, removal, searching, and traversal of obstacle sides.

##### BasicObstacleSide.cs

Base class for a side of an obstacle, wrapping a segment of the obstacle’s polyline (from one PolylinePoint to the NextOnPolyline).

###### LowObstacleSide.cs

A side of the obstacle that is between LowestVertex and HighestVertex in the “lower” direction of the scanline; i.e. to the left (lower X-coordinate) for horizontal scan (vertical sweep), and to the right (due to Cartesian coordinates) for Y.

###### HighObstacleSide.cs

A side of the obstacle that is between LowestVertex and HighestVertex in the “higher” direction of the scanline; i.e. to the right (higher X-coordinate) for horizontal scan (vertical sweep), and to the left (due to Cartesian coordinates) for Y.

### Event handling

The scanline is driven by events, which created for all obstacle vertices (points on the polyline), and possible Reflections (intersections with other obstacles which may “bounce” forward along the sweep direction).

##### EventQueue.cs

The priority queue which contains events ordered by “lowest-first” coordinate first in the sweep direction, then in the scan direction.

##### AxisCoordinateEvent.cs

For SparseVisibilityGraph, this is an event that indicates we are starting a low-to-high traversal of a coordinate at which one or more extreme vertices exists (regardless of whether it is visible across the full graph at that coordinate).

##### BasicVertexEvent.cs

The base class that carries information about the obstacle and its current vertex for ordering in the EventQueue and operations in the VisibilityGraphGenerator when it is dequeued.

###### OpenVertexEvent.cs

The opening vertex of an obstacle, defined as the lowest-lowest vertex – for horizontal scan, this is the lowest X coordinate at the lowest Y coordinate, and for vertical scan it is the lowest Y coordinate at the lowest X coordinate.

###### CloseVertexEvent.cs

The closing vertex of an obstacle, defined as the highest-highest vertex – for horizontal scan, this is the highest X coordinate at the highest Y coordinate, and for vertical scan it is the highest Y coordinate at the highest X coordinate.

###### LowBendVertexEvent.cs

A vertex between one LowObstacleSide and another (thus forming a bend).

###### HighBendVertexEvent.cs

A vertex between one HighObstacleSide and another (thus forming a bend).

##### BasicReflectionEvent.cs

The base class for reflections, tracking information about the two obstacles the reflection bounces between.

###### LowReflectionEvent.cs

A reflection on a LowObstacleSide.

###### HighReflectionEvent.cs

A reflection on a HighObstacleSide.

### Groups

Groups are boundaries around one or more obstacles (ideally, without overlaps at its boundaries) that defines the region in which all routing must occur when all sources and targets are within the group, and no routing between two obstacles outside the group should cross its borders. (Progresssion, our consumer in Visual Studio, calls this a Cluster). This boundary crossing is enforced within RER by creating tiny (one DistanceEpsilon length) VisibilityEdges which have a delegate that indicates which groups whose borders are crossed by that edge are currently passable (“transparent”). When routing between an obstacle inside and an obstacle outside a group, that group’s borders are made passable; otherwise it is opaque (unless some heavy overlapping occurs, in which case group boundaries are opened as needed to perform the routing). If two or more groups share a boundary, then if any of them is passable, the boundary is.

##### GroupBoundaryCrossing.cs

This class defines a single crossing of a group boundary, from a point on the group boundary. It includes the direction toward the inside of the group, which is the direction the tiny edge is created in; so consider the group boundary to be one DistanceEpsilon thick, with the point on the outside and the border crossed in the direction to the inside.

##### GroupBoundaryCrossingMap.cs

This class maps between intersection points on Group boundaries and the GroupBoundaryCrossings (groups crossed and the direction to their interiors) at those intersection points. This allows ScanSegments to query the map for all GroupBoundaryCrossings along the ScanSegment’s length, which is returned as a PointAndCrossingsList.

##### PointAndCrossings.cs

A single point together with all GroupBoundaryCrossings at that point.

##### PointAndCrossingsList.cs

An ordered list of PointAndCrossings, so that RER can traverse it in ascending order and splice in the tiny crossing edges. This is done once for the base VisibilityGraph, and dynamically for Port visibility splicing.

### Internal Utilities

These are the utility classes in RER that supply functionality for the Visibility Graph and Shortest Path stages.

##### PointComparer.cs

Some Point and Double comparison utilities, and a few shortcuts for other functionality such as ApproximateComparer and CompassVector. Most of the original implementation has been moved into ApproximateComparer.

##### ScanDirection.cs

This supplies functionality that lets the VisibilityGraphGenerator track the current direction (horizontal or vertical) and perform simple operations without having to pepper the code with “if horizontal… else” clauses.

##### SpliceUtility.cs

Some utilities to help with splicing. Most has been subsumed into ObstacleTree and TransientGraphUtility.

##### StaticGraphUtility.cs

A number of static utility functions for simple, common operations such as traversing edges and vertices, determining edge direction, determining whether a point is on an interval (or its interior) or if intervals intersect, and a number of TEST\_MSAGL output functions.

### Shortest Path generation

Once the base Visibility Graph is complete, we must each EdgeGeometry’s paths along its edges. This is done in two parts: first, VisibilityEdges for the Ports defined for the EdgeGeometry are spliced into the graph, and then the shortest paths are calculated.

#### Ports and port splicing

##### ObstaclePort.cs

A port that is in an Obstacle’s InputShape’s Ports collection is considered an Obstacle port. Usually this port is located at the obstacle center. RER calculates the intersections of that port with both its InputShape’s curve (the UnpaddedBorderIntersect) and its VisibilityPolyline (the VisibilityBorderIntersect, which may be the intersection with its PaddedPolyline or a ConvexHull). in both directions along each axis. Each of these intersections is then wrapped by an ObstaclePortEntrance. Alternatively, an ObstaclePort’s port may have a PortEntry defined, in which case that defines the ObstaclePortEntrances.

##### ObstaclePortEntrance.cs

This is an entry to the obstacle from a single direction to a single point on its border. ObstaclePortEntrance keeps track of the maximum visibility extension, whether the entrance is overlapped (by another obstacle), its group PointAndCrossingsList, and a few other things so they do not have to be recalculated.

##### FreePoint.cs

A FreePoint is a port that is not in an Obstacle’s InputShape’s Ports collection (although it may be within an Obstacle’s InputShape spatially). Like an ObstaclePort it tracks its maximum visibility and group PointAndCrossings list. FreePoints implement waypoints, so this efficiency is important.

##### WaypointPort.cs

This is a minimal specialization of Port that exists only to make Waypoints fit into the EdgeGeometry’s SourcePort and TargetPort, and also is used in PortManager’s Port-to-FreePoint map.

##### PortManager.cs

This manages the Port-to-ObstaclePort and Port-to-FreePoint maps and provides a number of functions to splice port visibility into the graph, such as finding the nearest ScanSegments and doing the initial vertex creation and splicing before handing off the TransientGraphUtility for the splice chain extension, and then removing the transient edges afterwards.

##### TransientGraphUtility.cs

“Transient” refers to the fact this class provides all the functions that create, track, and remove TransientVisibilityEdges (and temporary VisibilityVertices, which do not have a special subclass).

#### ShortestPaths

Once all port visibility splicing is done, we walk the Visibility Graph to generate shortest paths.

##### MsmtRectilinearPath.cs

This is the driver, with “Msmt” meaning Multiple-Source, Multiple-Target, where the source and target consist of multiple vertices (one for each ObstaclePortEntrance or FreePoint). For an EdgeGeometry without waypoints, MsmtRectilinearPath calls SsstRectilinearPath to calculate the shortest path for each pair of source and target vertices via a nested loop, with a cost cutoff to ensure that poor-scoring paths are not extended past the known least cost. For an EdgeGeometry with waypoints, MsmtRectilinearPath does a multistage approach, where a stage is source -> first waypoint, first waypoint -> next waypoint… final waypoint -> target.

##### SsstRectilinearPath.cs

This calculates the shortest path from one vertex to another, with “Ssst” meaning Single-Source, Single-Target. It is called multiple times by MsmtRectilinearPath. It preserves the best incoming path to each vertex in each direction so that a globally-optimal (between source and target) path is obtained.

##### VertexEntry.cs

This is an element on the traversal priority queue in SsstRectilinearPath.

# Details of Selected Implementation Areas

This section provides deeper explanation of the details of more complex implementation areas. This should be read as a high-level view, while the extensive code comments provide finer detail.

## Creating and using a RectilinearEdgeRouter

RectilinearInteractiveEditor.FillRouter is a good example of creating and using a RectilinearEdgeRouter. RectilinearTests are also a good example. Generally the sequence is:

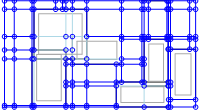
* Create a set of Shapes. It is common for this to be a LINQ statement over the Nodes of a GeometryGraph.
* Add ports to the Shapes. This may be done before or after creating the router.
* Create the router, passing the Shapes collection and other information such as padding and so on.
* Add an EdgeGeometry for each path to be routed.
* Call router.RouteEdges (or router.Run).

Shapes may be added to or removed from the router after its creation, but there is no “incremental rebuild” – the entire graph is redone on each RouteEdges call. This is because Nudging is usually the largest time consumer and it operates on the entire graph. However, the caller may track which obstacles have been changed, such as one or more being dragged, and create a smaller RER with only the affected obstacles (those being dragged, those connected to those being dragged, and possibly those within the bounding box of those directly affected), route only those edges, and re-route the entire graph when the mouse is released (or perhaps paused for some period of time).

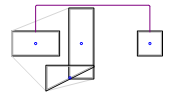
## Preprocessing Overlapping Obstacles into Clumps and Convex Hulls

As described above, we do not allow angled sides to intersect because this introduces floating-point indeterminacy and requires a number of special cases to handle well, and is difficult to ensure that all possibilities have been accounted for. Instead, only rectangular obstacles (including groups) are allowed to overlap individually; non-rectangular obstacles that intersect are formed into their composite convex hulls.

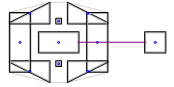
For Clumps, we therefore have the concept of overlapped segments and VisibilityEdges; these are from ScanSegments that are created from extreme obstacle vertices that are contained within other obstacles. In the below illustration, the light-blue edges are overlapped; they are inside an obstacle, and have high weight. By having high weight, they are not taken unless no other (reasonable) path exists; for example, obstacles that are landlocked by other obstacles. They also minimize crossing of obstacles when such a landlock exists, or when routing to or from a nested obstacle.



For Convex Hulls, we may have artifacts from the new Hull segments causing additional overlaps. In heavily overlapped graphs this can become significant, but we are optimizing for the real-world case in which overlaps are rare, usually the temporary result of dragging. In most cases this convex hull extension will not matter, such as this illustration (where the convex hull is the light gray outline):

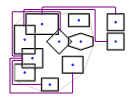


Objects in convex hulls are not considered individually in the Visibility Graph; rather, the Hull’s outer polyline is shared by all siblings within that hull as their VisibilityPolyline. This means that in cases where all 4 directions from a heavily nested interior obstacle to the border of the convex hull cross other obstacles, then the resultant path will cross the obstacle:

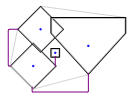


This was determined to be an acceptable tradeoff for the gains in code simplicity and testability.

When routing between obstacles in the same convex hull, normally the unpadded-to-padded border intersect edge is taken for the first obstacle, the routing outside the hull is done to a padded border intersect for the target, and then the padded-to-unpadded border intersect edge is taken to complete the routing – i.e., instead of the direct path, routing goes outside the hull and then back inside.



The unpadded-to-padded border intersect edge is weighted according to whether it crosses over a sibling obstacle; if not it has normal weight, otherwise it has overlapped weight. There is a special case where this unpadded-to-padded border intersect edge crosses the source or target obstacle of a path. In that case, the path will be trimmed by CalculateArrowheads to the first intersection with the crossed obstacle; for example, below the Diamond-to-littleRectangle path passes across the diamond so is trimmed from the 5-segment curve that existed after nudging (which went to the bottom of the graph, then up) to a single segment.



This intersection may not be at a valid span on the obstacle’s PortEntry, if such is defined; in most cases, the PortEntry of an obstacle inside a Convex Hull is honored, because it is defined on the InputShape.

Groups are a slightly special case. Although they have a convex hull, that hull is specific to the group rather than shared. Also, to avoid intersecting non-orthogonal obstacle sides in the scanline, when a group and an obstacle overlap, the group’s convex hull is formed around the obstacle’s LooseVisibilityPolyline, to have a little padding between them.

The “logic at a glance” for forming clumps or convex hulls (collectively sometimes referred to as obstacle accretions) is:

* Shapes that don't intersect anything are not affected
* Non-Group Rectangular shapes that intersect each other are overlapped into a clump
* Rectangular Groups intersecting other rectangular groups or nonGroup rectangular shapes are not affected
  + Groups are not in a clump; they are translucent
* Intersecting obstacles where either is non-Rectangular become convex hulls:
  + If neither is a group, they become one convex hull with one primary and one internal (primary for scanline purposes)
  + If one is a group, the group's convex hull grows to encompass it and a little bit more (using the Obstacle’s LooseVisibilityPolyline) so the boundary sides don't touch in the scanline
  + If both are a group, the larger grows similarly to encompass the smaller’s LooseVisibilityPolyline.
  + If a rectangular obstacle in a clump is added to a convex hull, all obstacles in that clump are added

This process is iterative, repeated until no new overlaps are detected.

## VisibilityGraph Generation

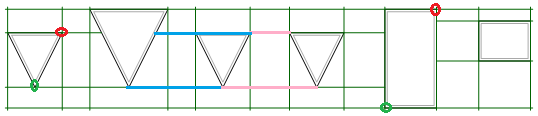
Once the Obstacles have been finalized, the VisibilityGraph is generated. This is done in two phases:

* Obstacle-derived visibility is generated once
* Port-derived visibility is added (“spliced”) into the VisibilityGraph for each EdgeGeometry (ShortestPath routing), and is removed when ShortestPath is complete. This reduces the size of the VisibilityGraph and allows ports to be changed without regenerating the initial (Obstacle-derived) VisibilityGraph.

### Creating ScanSegments: Full Visibility Graph

With Full Visibility Graph, we create ScanSegments to the fullest length possible without crossing an obstacle, and then intersect those ScanSegments. ScanSegments are created during the Scanline sweep, at the lowest-lowest (OpenVertexEvent) and highest-highest (CloseVertexEvent) vertices of an obstacle (ignoring reflections, which are discussed elsewhere).

#### Creating the ScanSegments



In the horizontal scan (vertical sweep), the green circles are the lowest-lowest vertex, which is the lowest X at the lowest Y. For the inverted triangles there is only one lowest-Y vertex; for rectangles there are two. Similarly, the red vertices are the highest-highest; the highest X at the highest Y. What may be non-intuitive at first is that the same lowest-lowest and highest-highest relationship holds in the vertical scan (horizontal sweep) as well, because RER uses the MSAGL standard of Cartesian rather than page coordinates. In Cartesian, Y increases as it goes up. On the vertical sweep we scan horizontally and the lower scanline-parallel coordinate is to the left. On the horizontal sweep we scan vertically and the lower Y coordinate is to the right.

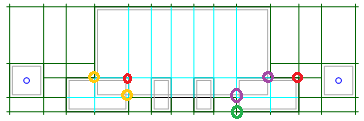
The light-blue and pink lines in the illustration show the ScanSegments created by the VertexEvents of the third-from-left triangle. On the OpenVertexEvent, we create the lower blue segment by looking toward the first intersection with a neighbor side in the lower direction. This includes an intersection at the border of the neighbor obstacle; ScanSegments are added to by obstacle VertexEvents at the same scanline-perpendicular (sweep-direction) coordinate until they cross an obstacle (or hit the sentinels at the boundary of the graph). In this illustration, the blue line is actually extending the (green) ScanSegment that starts at the left graph boundary; this was created by the leftmost triangle’s OpenVertexEvent.

After we’ve created (or extended) a ScanSegment in the low direction, it becomes the hintSegment. Then we look toward the high direction of the scanline to find the high neighbor and create a ScanSegment, which in this example will once more extend the hintSegment. In fact, for non-overlapped obstacles with no reflections involved, we usually just create hintSegment at the first VertexEvent at a given scanline coordinate, extend it until we hit an obstacle, then start a new one on the other side of it.

Similar logic applies to the CloseVertexEvent, and in this illustration we also see what happens for a flat obstacle side. These sides are not put into the Scanline because they have zero size in the sweep direction. We don’t need them there; when we look for a neighbor in the low or high direction, we simply skip any intervening side of the VertexEvent obstacle. In the above illustration, that happens with the blue segment; we look low (left) and skip over the side of the event obstacle, and find the neighbor obstacle. The pink segment of the CloseVertexEvent is simply an extension of the hintSegment as described for the OpenVertexEvent.

#### Overlapped ScanSegments

Only rectangular obstacles are allowed to overlap. If this happens, then the ScanSegments derived from any vertex that is within another obstacle’s boundary is “overlapped” until it passes outside all enclosing obstacles. For overlaps, an additional aspect of the definition of “neighbor” becomes relevant: a neighbor is the first obstacle adjacent to the current obstacle in the specified direction such that when searching in the low direction, we encounter its HighObstacleSide; or when searching in the high direction, we encounter its LowObstacleSide. When searching in the low direction, any LowObstacleSide we pass through belongs to an enclosing (overlapping) obstacle; when searching in the high direction, any HighObstacleSide we pass through belongs to an enclosing (overlapping) obstacle. Note that an obstacle may be both a neighbor and an overlapper, as shown in the following diagram.

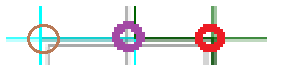


In TestRectilinear, overlapped ScanSegments are shown in light blue. As in previous examples, the green vertex is an OpenVertexEvent, and the red vertices are CloseVertexEvents. For the green vertex and the rightmost red vertex, the large rectangle is a neighbor, despite overlapping the VertexEvent obstacle; this is because we encounter its outer edge (at the purple-circled intersections) when scanning from the VertexEvents:

* For the green vertex, on vertical scan we will be looking to the high side (up) and find its LowObstacleSide
* For the rightmost red vertex, on horizontal scan we will be looking to the low (left) side and find its HighObstacleSide
* For the leftmost red vertex, on vertical scan we will be looking low (down) and find its LowObstacleSide; on horizontal scan we will be looking low (left) and find its LowObstacleSide.

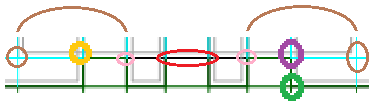
As shown here, there may be overlapping and non-overlapping ScanSegments end-to-end. For an interior vertex, we create or extend an overlapped ScanSegment to the intersection with the border of the outermost overlapper. If there is a neighbor beyond that, we also create (or extend) a non-overlapped segment from the overlapping-border intersection to the neighbor intersection. This can be seen at the gold intersections.

The purple intersections are created differently. They illustrate that we scan in both directions from both endpoints of the event obstacle’s LowObstacleSide and HighObstacleSide. If those ends meet, for example at the point of a triangle, then we start the low and high neighbor scans from the same point. But if they are different, for example two sides of a rectangle, then we scan in both directions from the endpoint of the LowObstacleSide *and* from the endpoint of the HighObstacleSide. In the non-overlapped case, it makes no difference, because we skip intervening sides of the event obstacle. However, in the overlapped case, we may have an overlapping side between them. That is what happens above, at the purple vertices. Repeating and enlarging this portion of the illustration here for detail:



At the red vertex we have a CloseVertexEvent. We look left, find the neighbor at purple, and create the scansegment (then we look right for another neighbor and extend in that direction). However, because the two active sides of the event obstacle do not meet, we have looked for both low and high neighbors from the LowObstacleSide’s endpoint (brown vertex) as well as from the high vertex. So at the brown vertex we find a neighbor in the low direction and create or extend an overlapped ScanSegment to it; then we find a neighbor in the high direction, and seeing that we pass through an overlapping obstacle side and emerge from it into non-overlapped space, we extend the overlapped scansegment to the overlapping border, then create a non-overlapped scansegment beyond it (so in reality, the high vertex simply subsumes a new Scansegment into the existing non-overlapped Scansegment just created from its obstacle’s low side endpoint).

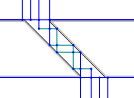
One more aspect of the above diagram is important: we can have a gap in the ScanSegments if we are sufficiently overlapped. Again focusing on a particular area of the diagram:



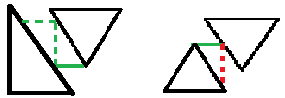
The two brown-circled vertices are the lower corners of the large rectangle. Creating ScanSegments as described above results in an overlapped and then non-overlapped ScanSegment sequence ending at the pink-circled intersections (under the brown arcs), leaving a gap in the ScanSegments at the portion of the large rectangle’s border circled in red. In practice this is not a problem, because the overlapping rectangles will in turn create some ScanSegments, as can be seen above by the single green ScanSegment across the bottom, and the graph will be connected.

#### Reflection Segments

Reflections are used to go betwheen non-rectangular obstacle sides when there is no straight line through that gap that does not touch an obstacle; thus, they only bounce between the same two obstacles. The below graph shows a path that uses Reflections and a close-up of the visibility graph showing the edges for Reflections. The Reflection edges are shown in teal, and have a higher weight than normal (but much less than overlapped) to reduce the chance they will be taken except when the alternative is to take a much longer path around.

A path with Reflections: The VisibilityGraph for that path: 

Reflections are always done “upward”, in the direction of the sweep; hence the term “lookahead”. In the below illustration, the leftward (green) vertical obstacle intersection of the ScanSegment derived from the middle obstacle will reflect upward but the rightmost (red) one would reflect downward (where the scanline has already passed) so is not done:

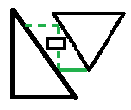


Reflections are done only in Full Visibility Graph and only for non-overlapped segments. If one of the reflecting obstacles is a group, then reflections are done only on the outside boundary. Reflections up to the inside boundary of a group are automatically ignored because the group breaks the “same two obstacles” reflection requirement.

The sequence of Reflections through the event queue is:

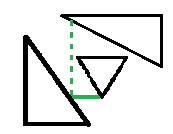
* StoreLookaheadSite (the bottom of the vertical green dotted segment above)
* When an ObstacleSide is loaded, it looks to see if there are any pending lookahead sites in its range along the scanline. If so it calls LoadReflectionEvent which calculates the point along the ObstacleSide that the reflection would hit (the top of the vertical green dotted segment above) and puts a LowReflectionEvent (above) or HighReflectionEvent into the EventQueue
* When that ReflectionEvent is pulled from the queue, if the site is still in the LookaheadScan, then a perpendicular segment is generated. If this still spans the perpendicular range of the other obstacle in this reflection staircase, then a parallel segment is generated (the horizontal green dotted segment above) and if that generates a reflection, as it would in this example, the cycle is repeated.

Because these are looking ahead in the sweep direction, there are two complications that arise, related to not knowing whether obstacles are opening or closing until we get to them. In the first case, the reflection can be intercepted:



In this case we do not want the ReflectionEvent (at the top of the vertical green dotted segment) to create a perpendicular segment, because that would cross the obstacle in the middle. That is why we defer removing the lookahead site until we have processed the event, or as in this case, the middle obstacle has removed and discarded the event because it is not one of the two original reflection partner obstacles.

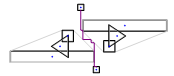
The second case is the opposite; a current obstacle may close while reflection events it generated are still in the LookaheadScan. In this case we want to tell neighbor sides that are already loaded to look again for reflection events to load (or more likely in this case, discard):



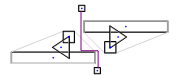
For more details of the Reflection implementation, see the extensive code comments.

When Convex Hulls exist, reflections happen on the convex hull boundary, not on the individual obstacles. For Nudging, the original PaddedPolylines are used, and fortunately the Nudger is quite good about removing staircases between Convex Hulls that would otherwise be in free space. However, the path is not always positioned evenly within the free space:

Before Nudging:

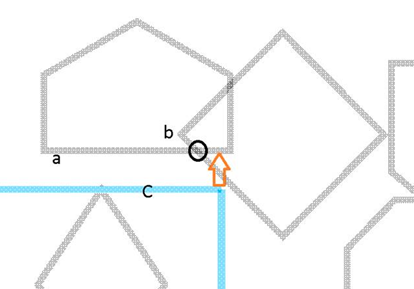


After Nudging:



Ideally, the Nudger could then center the path in the freed space (the horizontal segment in this example), but currently this is not done; it would require another iteration of the Nudger to do so (see “Potential Improvements”) and the priority of improvements to multiple overlaps is low.

Another aspect of Nudger and overlapped obstacles is that, for the same reasons that the main Rectilinear scanline does not support intersecting angled sides, Nudger does not and therefore does not reorder sides after an intersection. This means that even in some cases where ShortestPaths does not cross over an obstacle, Nudger may nudge into an obstacle if it is overlapped with another obstacle in such a way that Nudger has not updated the obstacle sequence from the side intersection, as shown below:



Here, the sweep goes from left to right; path C at the arrow is restricted by side *a* rather than side *b*, because the Nudger does not reorder sides after the intersection. Therefore the path may be nudged up as shown by the arrow.

### Creating ScanSegments: Sparse Visibility Graph

For Sparse Visibility Graph, we do multiple passes. Our implementation is close to the Clarkson paper in a number of ways so reading that paper will be instructive. However, unlike that paper we do not create angled edges at obstacle sides, and lacking reflections we may choose a longer path. However this will usually not be a problem – the longer path may sometimes be more understandable. In some dense graphs with non-rectangular obstacles with or without overlapping obstacles we may cross over obstacles where the Full Visibility Graph will not.

The main differences between the two approaches are:

* Full Visibility Graph
  + Limit ScanSegments to their blocking intersection with neighbors (if it doesn’t skim along the border of an obstacle, but rather would pass through it, the ScanSegment ends), with no extension of the ScanSegment on the other side of the obstacle unless there is a collinear VertexEvent on that side of the obstacle.
  + Create ScanSegments on VertexEvents
* Generate a VisibilityVertex at every intersection of a horizontal and a vertical ScanSegment
* Sparse Visibility Graph
  + ScanSegments also end at neighbor sides, but effectively pass across the entire graph; multiple collinear ScanSegments are created at each VertexEvent coordinate, separated by the space between obstacles (or by overlapped ScanSegments crossing through those obstacles).
* Generate a VisibilityVertex only at binary-division intervals within the ScanSegment range

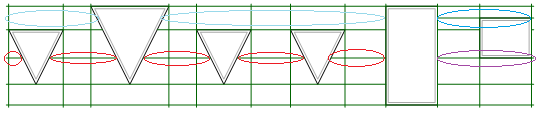
SparseVG first goes through the obstacles and accumulates the coordinates along both axes of the obstacles’ bounding boxes (this is another difference from the paper; we only create ScanSegments on the borders of the obstacle, ignoring (for the purposes of ScanSegment creation) “bend” vertices – vertices that join two obstacle sides but are not on one of the obstacle’s boundingbox borders). It then creates vectors along both axes where each element is at a coordinate derived above. These ScanSegmentVectors contain ScanSegmentVectorItems which accumulate intersections and provide other services to create the ScanSegments. Then we run the Scanline, adding additional events to it at the points of the low graphbox boundary corresponding to the ScanSegmentVector coordinates; at these coordinates we create the ScanSegments, as in FullVg limited to the obstacle side visibility as carried in the Scanline, and adding additional Steiner points at the bounding box corners of each obstacle to make sure we have a tight turn around each obstacle. Finally, we do the “sparse” intersections, as determined by binary division of the distance from the vertex points.

#### Accumulating Vertex Coordinates

As described above, we create an array of elements on each axis; this array corresponds to the perpendicular coordinate of an OpenVertexEvent or a CloseVertexEvent. Conceptually there is a line for each of these across the entire graph, but that line skips the interior of obstacles (unless they are overlapped) so is broken into multiple ScanSegments.

#### Creating the ScanSegments

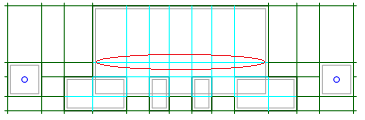
As in FullVg the scanline is run. However we do not create the ScanSegments at the VertexEvents; those are used simply to keep the Scanline updated with obstacle sides. The EventQueue for SparseVg has additional AxisCoordinateEvents defined for each point on the perpendicular graph boundary, and it is on those events that we create the ScanSegments (as well as the bounding box Steiner points).



Here, we see that the red-circled ScanSegments do not pass through an extreme vertex; instead they are extensions of the purple-circled ScanSegment (or rather, of the coordinate of its VertexEvent). The darker-blue circled ScanSegment is similarly extended, but due to the binary division intersection generation, there are no intersections along its extensions (ligher-blue circles), so they were removed. However, this coordinate was still available during binary division (described below). This means that for this simple example, the ScanSegmentVector along the Y axis has 6 elements: the 4 extreme-vertex derived elements that are visible from the left side of the graph, an element for the red coordinate, and an element for the blue coordinate.

#### Overlapped ScanSegments

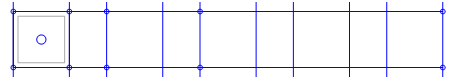
As with FullVg, ScanSegments may pass through obstacle interiors in the event of overlaps. If there are intersections (or bounding box corners) on them, they are retained.



Here, the red-circled ScanSegment is an extension from extreme vertex coordinates outside the obstacle, and is retained because the binary-division intersection generation placed intersections on it.

#### Generating Sparse Intersections By Binary Division

Unlike the exhaustive intersection done by FullVG, SparseVG uses binary division to determine which intersections to put into the Visibility Graph. This is why all extreme coordinates conceptually span the entire graph; otherwise there would be additional (possibly scanline-like) logic to track which segments are visible at any given point, and it is not clear that this would help because the use of a constant set of perpendicular coordinates means that it is more likely the binary division will find the same coordinates, which may “smooth” the number of bends traversed by ShortestPath and thus reduce the chance of spurious suboptimal-path selection.



This diagram shows a simple example of extreme vertices at the top and bottom. There are only 6 intersections to the right of the obstacle; FullVG would have 16. The reduced number of vertices comes from dividing the distance (in terms of the number of perpendicular ScanSegments in the interval) by 2 for each intersection we put in. Here is a more detailed explanation of the algorithm:

Given a set of obstacles with vertices *P*, build horizontal edges from the points of *P* (vertical edges are done similarly).

Let *px* be the projection of point *p* to the *x*-axis. Let *X* be the set of all unique *px* from *P*, sorted in ascending order; *X* = [*r0 ,r1, … rn-1*] where *ri* < *ri+1*. Let *Q* be a mapping from *P* to natural numbers such that *rQ(p*) == *px*; in other words *Q* gives the index of the projection of *p* in *X*.

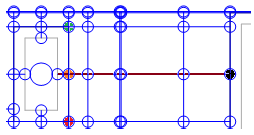
Make a sweep with a horizontal line. When we hit an obstacle vertex *p*, find the ScanSegment *S* containing *p*; this contains the maximum visibility of *p*, ending at the side of another obstacle or at the outer boundary of the graph. Let *a* = *S*.*Start* and *b* = *S.End*. Create intersections at *a* and *b*.

Determine which half of the graph *px* is in: Let *k* = *Q(p)*. If *k* <= ((*n*-1)/2) then set *i* = 0 and *j* = ((*n*-1)/2); otherwise set *i* = ((*n*-1)/2) and *j* = (*n*-1). (This is half of the number of perpendicular coordinates, which may not be half the graph spatially).

Now build the rest of the horizontal edges from *p*. If *ri* >= *a* and *i* < *k* then add the intersection at the vertical line with the *x*-coordinate *ri*. If *rj* <= *b* and *k* < *j* then add the edge from p to the vertical line with the *x*-coordinate *rj*. Let *m* = ((*i* + *j*)/2). If *m* <= *k*, then let *i* = *m*, otherwise let *j* = *m*. Stop if *i* == *j*, otherwise go to the previous step.

### Port Visibility Splicing

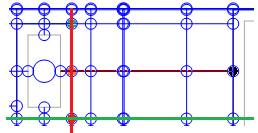
This refers to adding VisibilityEdges and VisibilityVertices to the graph for an ObstaclePortEntrance or FreePoint. The approach is to first find the closest location on an existing VisibilityEdge that we can create a vertex at (splitting the edge), then splice in a new VisibilityEdge chain extending from that initial vertex.



In the above diagram, because the obstacle side is parallel to an axis, the middle (orange) vertex is both the VisibilityBorderIntersect for the obstacle and the initial vertex for splicing, and we are extending the edge chain Eastward (creating the brown edges) to its maximum visibility (the black vertex, which is on another obstacle’s side).

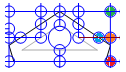
#### Creating the initial VisibilityVertex

The first thing is to find the edge containing this vertex (assuming the vertex does not yet exist). Because of the large number of vertices and edges we do not keep a direct index of them and therefore we do not know the locations of the vertices or edges near our desired initial vertex. Therefore we use the ScanSegmentTrees as indexes, finding the lowest ScanSegment that intersects the extension line in the perpendicular extension direction; in this case, because the extension is horizontal, we want to find the vertical ScanSegment with the lowest X coordinate. Then we look for the highest ScanSegment intersecting the lowest perpendicular intersector below the desired vertex location; this will be parallel to the extension direction, as shown in the following diagram, where red is the lowest initial intersector and green is the highest intersector of that intersector below the desired vertex location.

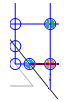


At the intersection of the red and green lines, we either have a vertex if we are in Full Visibility Graph, or we traverse the edges of the ScanSegments to insert one if we are in Sparse Visibility Graph. With that vertex, we have the edge that crosses the initial vertex location (in this case, the N edge from the red vertex to the green vertex), so we split it.

In this example the initial vertex is the VisibilityBorderIntersect. If the sides are angled, then the initial vertex is created on the ScanSegment as described, and then an edge is created connecting it to the VisibilityBorderIntersect, as in the following diagram where the red, orange, and green vertices are as above, and the light-blue one is the VisibilityBorderIntersect.



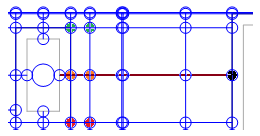
If sides are angled, the second intersecting ScanSegment may not exist:



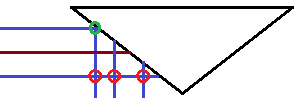
In this case the brown horizontal line is the edge visibility chain we want to extend to and past the orange initial vertex. This is below the lowest segment intersecting the perpendicular ScanSegment (at the green vertex). In this case, the initial visibility vertex is prepended (or appended, if it is higher instead of lower) to the perpendicular ScanSegment’s visibility edges.

#### Extending the Edge Chain From the Initial Vertex

Now that we have the initial vertex created, we can walk along the existing edge chains (which are in ScanSegments) to either side of the chain we’re creating (which is not inside a ScanSegment). So we look to the left of the extension direction for the first bracketing vertex (called spliceSource) that we’re splicing across (the initial green vertex), and in the rightward direction for the second bracketing vertex (called spliceTarget). From these, we find the next vertex in the extension direction; these are the second green and red vertices below. From these we can determine the location of the next vertex to extend to (the second orange vertex), and we proceed until we are at the end of the desired range (if we are limiting visibility splicing to the minimal rectangle containing the source and target), or the end of visibility (the black vertex).



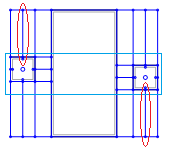
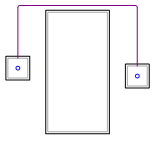
For angled sides, we may not arrive at the end of the extension with both spliceSource and spliceTarget:



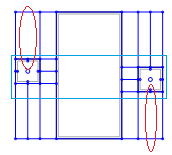
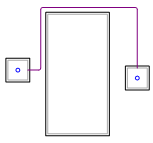
Here, the spliceSource chain ends at the green vertex, matching the first red vertex. In this case, we flip spliceSource and spliceTarget sides, and then continue the splice to the second red vertex and beyond, as long as we would not cross the interior of an obstacle to do so (as we would if we tried to extend as far East as the third red vertex above).

#### LimitPortVisibilitySpliceToEndpointBoundingBox

In many cases, we do not need to extend the Port visibility edge chain past the opposite endpoint of the path. However, there are cases where an intervening obstacle will cause extra bends to be taken if the visibility extension does not go past that intervening obstacle. GroupTest\_Simple\_NoGroup shows an example:

VisibilityGraph Path 

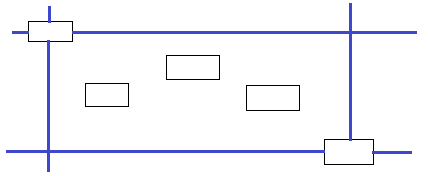
The red-circled VisibilityEdges are created from port splicing. The light-blue rectangle is the bounding box containing the two endpoints, and if we limit to this bounding box, we get:

VisibilityGraph Path 

Limiting the port visibility splice to the endpoint bounding box means that the shortest path around the intervening obstacle does not exist. “Limiting” means “to or past”, so if the limiting rectangle is in the middle of the next edge of the new visibility chain, the edge will be created.

### Direct Visibility Intersection to Bypass Port Visibility Splicing

Because each ObstaclePortEntrance carries its MaxVisibilitySegment, we can optimize routing by selecting the first intersecting (or collinear) pair of ObstaclePortEntrances. If the obstacles are widely separated with a number of obstacles between them, this can save a considerable amount of time in some cases.



The blue lines indicate visibility of the ObstaclePortEntrances on the upper left and lower right obstacles. Depending on the enumeration sequencing, we may choose either the lower left or upper right intersection.

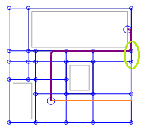
This is done only for ObstaclePorts; Waypoints are part of a multistage path and thus would require more work across the waypoints.

## Shortest-Path generation

If direct visibility intersection is not done for a path, then once the VisibilityGraph is ready we route the edges along it via MsMtRectilinearPath (MultiSourceMultiTargetRectilinearPath), which calls SsStRectilinearPath (SingleSourceSingleTargetRectilinearPath).

### SsstRectilinearPath Retaining Multiple Directional Incoming Paths at Vertices

The below drawing illustrates behavior with only a single best-incoming-path retained at a vertex, and shows how this is improved by retain one best incoming path per compass direction. The lower of the green-encircled vertices was arrived at first in the Eastbound direction, from the path that went North and then explored East. Subsequently the orange arrived at the vertex, but because its score was the same as the previous path, it was discarded. By keeping both and scoring “through” the vertex to the following vertices, the globally best path is retained (in this case, by following the orange path North).



### Routing a Path With Waypoints

The Nudger does not know about waypoints; a single path can be made to go through the waypoints, but the Nudger will most likely move it off those points. Thus, we route waypoint paths in multiple stages, where a stage is source -> first waypoint, first waypoint -> next waypoint… final waypoint -> target. This takes advantage of the fact that each vertex retains the best incoming path for each compass direction.

* From all the paths from each source vertex to the Waypoint, it preserves the best entry into the Waypoint from each direction, up to a cutoff of cost increase from the best path to that vertex (currently this allows the cost of an additional bend in the path, so that any extension from the waypoint to the first vertices of the following path stage is not prematurely pruned).
* Continues each incoming path across one waypoint to the next waypoint until arriving at the final waypoint.
* Continues each incoming path across the final waypoint to each target vertex, selecting the best.

### Group Boundary Crossings

A Group is a shape that contains other shapes by design, in its Children collection. This is different from overlapping obstacles, which have no relationship other than spatial intersection. Normally, a group’s children are spatial children as well, and do not overlap the group boundary (except perhaps when dragging), and obstacle not in the group’s children do not intersect its space.

A Group’s boundary should not be crossed when routing unless the source and target are in different groups (for example, the source is outside the group and the target is inside it). This is primarily implemented by the GroupBoundaryCrossing\* and PointAndCrossing\* classes. During visibility graph generation (including port visibility splicing), any edge that crosses a group boundary is actually 3 edges: the edge outside the group, a tiny (one DistanceEpsilon in length) edge that crosses the boundary, and the edge inside the group. A VisibilityEdge has an IsPassable delegate which, if non-null, means that edge is a group crossing edge and the delegate is a closure that returns whether any group in the set of groups crossed by that edge is passable (also called transparent).

When groups are present, there may be three passes of RER path generation; the first to return a non-null path is taken.

* The normal case is as described above, the source and target have different group memberships. This ancestry is recursive, so if obstacles are nested in multiple groups, the right thing is done even if they are in only some of the groups.
* If two obstacles are in the same group’s Children collection but one is spatially contained and the other isn’t, then for the entire graph, “spatial children” are added to the RER’s copy of its spatial parents ancestry hierarchy, and nonspatial ones removed. (Because this is done only as needed, some paths will be different when routing multiple paths vs. individual paths. This may be confusing when debugging.)
* Finally, all group boundaries are temporarily made transparent, under the assumption that non-crossable groups are forming a landlock. This is done on a per-path rather than whole-graph basis, so a single landlock will not create a degenerate graph.

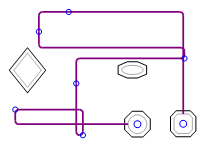
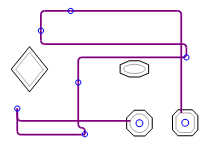
### Out-of-Bounds (OOB) FreePoints and Waypoints

One complicating factor for FreePoints (including Waypoints) is that they may be outside the graphbox – the bounding box around all obstacles in the graph. For non-Waypoints, this is fairly simple, just adding an edge to a point on the graph (or via two sets of two edges, in both directions, if the FreePoint is out of bounds in both axes).

Waypoints introduce the additional complexity of at least trying to create attractive paths across the waypoint, so we add additional visibility between successive waypoints:

* If both are OOB and on the same side, then we add a single edge between them if they are collinear, or two sets of two edges if not.
* If one Waypoint is inbounds, or the routing is being done from an OOB waypoint to an in-bounds endpoint (and obstacles will always be in-bounds by definition), then we add an edge from the OOB waypoint (if it is OOB in only one direction) to the point on the graphbox boundary intersect of the inbounds port.
* If there are two OOB Waypoints at opposite corners, we create two sets of two edges between them.

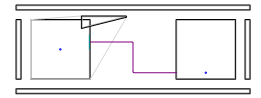
For both inbounds and OOB Waypoints, the foregoing is not always sufficient to avoid backtracks (see, for example, RectilinearTests.Waypoints\_MultipleOobCrossesGraph); Shortest Paths will not backtrack if a “loop” through the waypoint is possible, but the Nudger does not know about other stages of the waypoint path so may Nudge these paths into a backtrack.

Before Nudging: After Nudging: 

However, it should be possible for the caller to automatically generate Waypoints positions and obstacle assignments in such a way that successive path stages do not move in a backtracking direction.

### Port Entries

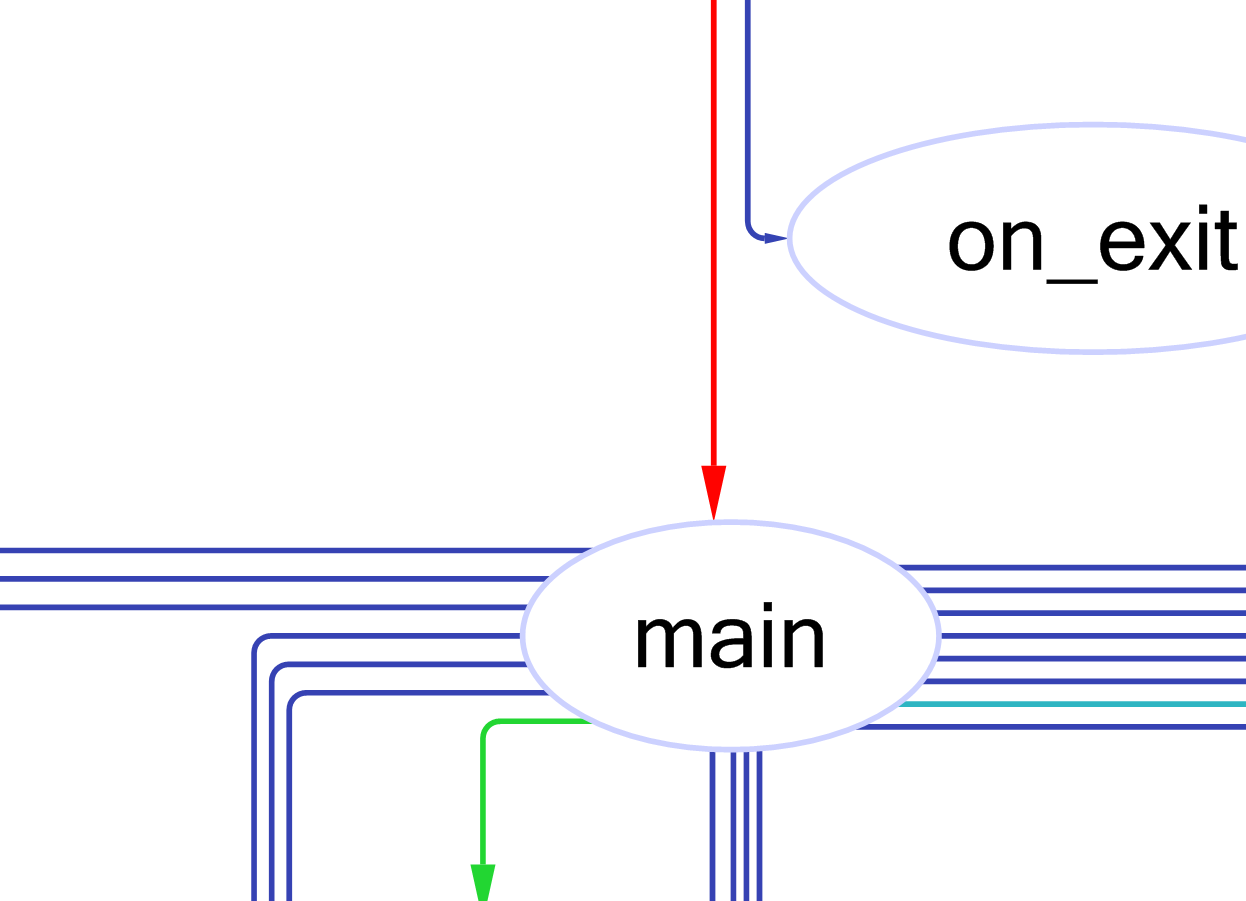
A Port Entry (also called a region) defines one or more areas on the obstacle to which a path may attach. The only current implementation is PortEntryOnCurve, which consists of one or more pairs of curve parameters, each of which is the start and end of a valid attachment range. If a Port has a PortEntry defined, then the ObstaclePort created for that Port will have its ObstaclePortEntrances defined from those ranges, with the midpoint of each facet serving as the ObstaclePortEntrance’s UnpaddedBorderIntersect. Because this is defined on the Obstacle.InputShape.BoundaryCurve, the PortEntry is honored even if it is inside a Convex Hull, as shown below (the turquoise area of the left square is the PortEntry span).



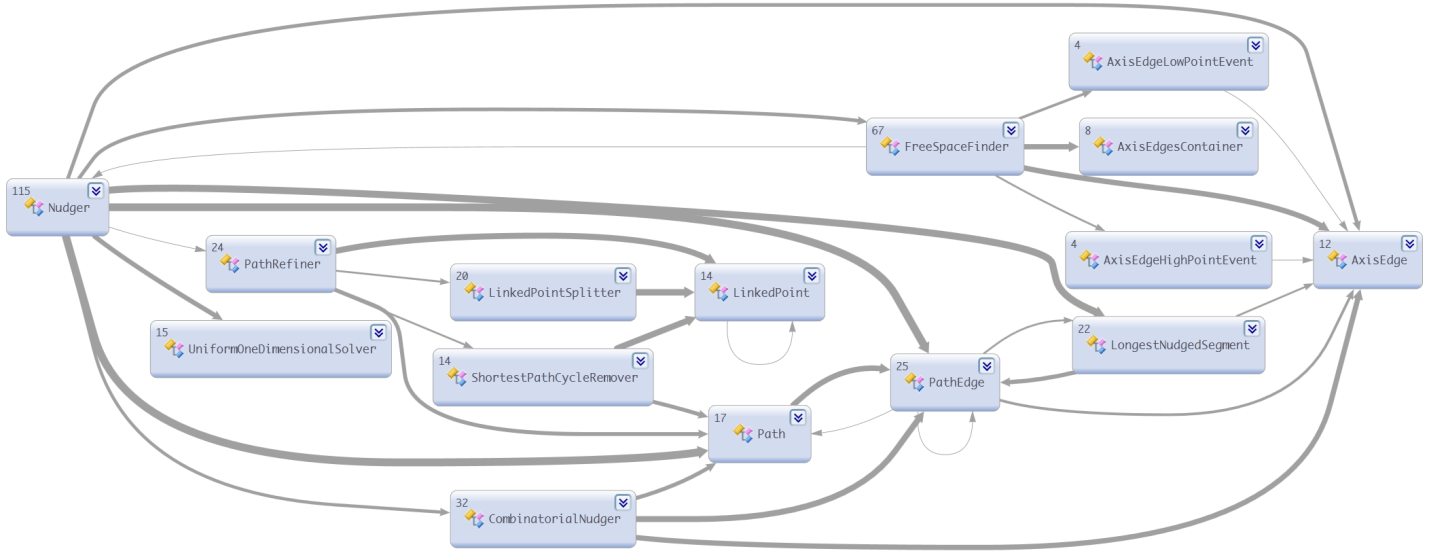
For most shapes that are used in RER, it is possible to wrap around the endpoint of the closed curve by defining, for example, {3.75, 0.25} to have an ObstaclePortEntrance that has the last quarter of the last segment and the first quarter of the first segment. These shapes, such as Ellipse currently, throw a NotImplementedException on TrimWithWrap.

## Nudging

Nudging is a term from the Wybrow paper. Here it means that edges nudge one another when they would be otherwise overlapping, or are nudged to evenly share space between obstacles or at an obstacle attachment point. The space between edges makes them more readable. During the nudging process we try to remove unnecessary bends and crossings. The drawing below clarifies the concept.



Here is the hierarchy of main types responsible for nudging in MSAGL



## 

### Main steps of nudging

Our implementation, while initially based upon the Wybrow paper, has a number of corrections and improvements. Here is the main method of Nudger.

internal static void NudgePaths(

IEnumerable<Path> paths,

double cornerFitRadius,

IEnumerable<Polyline> paddedObstacles) {

if (!paths.Any())

return;

var nudger = new Nudger(paths, cornerFitRadius, paddedObstacles);

nudger.Calculate(Direction.N, true);

nudger.Calculate(Direction.E, false);

nudger.Calculate(Direction.N, false);

foreach (var path in paths)

path.EdgeGeometry.Curve = new Polyline(BuildPolylineForPath(path));

The input for Nudger is an enumeration of Paths to nudge, a number representing the desired radius of the arcs inscribed into the polyline corners, and an enumeration of the polylines representing padded polygons around the obstacles. A nudged edge should not overlap paddedObstacles unless the edge is adjacent to the obstacles. The fields of a Path which have to be set at the input time are EdgeGeometry, PathPoints, and Width. PathPoints, an enumeration of Points, are calculated by the shortest path router.

Nudging works by making a pass over the paths going north, then east and we make another pass north. Here is an illustration of a change the can happen to and edge when a pass east is done.

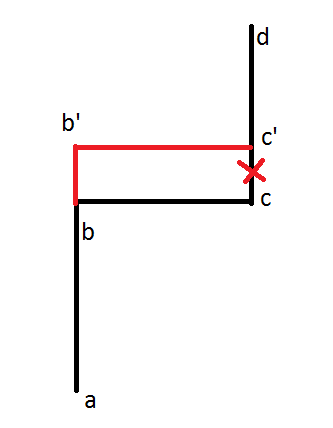


Figure : Edge *bc* is nudged to *b'c'*. This also changes edges *ab* and *cd*. If *c’* coincides with *d* then edge *cd* disappears.

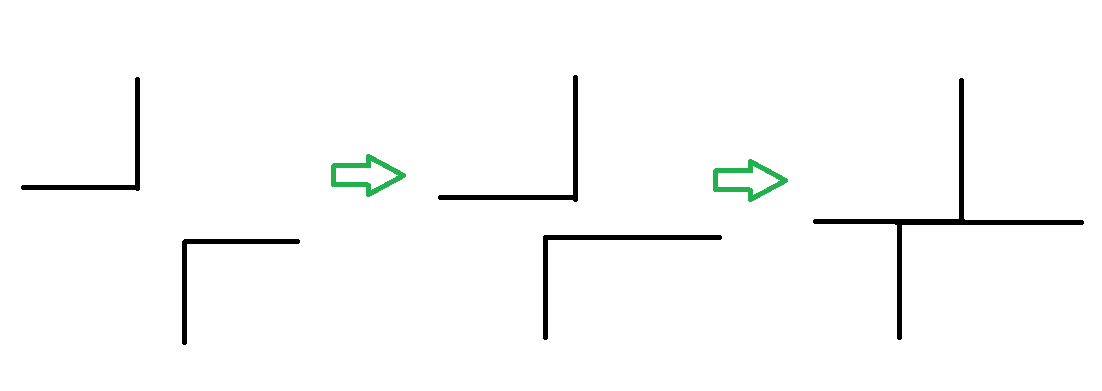
Why do we call Calculate() three times? The picture below illustrates it 

Figure . Two paths don’t overlap before the nudging, and they don’t overlap after nudging in the north direction, but the start overlapping after nudging in the east direction.

As the picture explains each nudging step may introduce new path overlaps, but we experimentally found that three passes is a reasonable compromise. A TODO item here is to try to find a way to check that the third nudging is really needed and avoid the nudging if it is not needed.

### Nudging in direction

Let us describe how one nudging pass works. The code of the function *Calculate()* making a pass is given below.

internal void Calculate(Direction direction, bool mergePaths) {

NudgingDirection = direction;

PathRefiner.RefinePaths(Paths, mergePaths);

GetPathOrdersAndPathGraph();

MapAxisEdgesToTheirObstacles();

DrawPaths();

}

Let us give a high level description of this function. Its input is the Direction of nudging that can only be north or east. The function PathRefiner.RefinePaths() modifies the paths in a way that the some conditions, that will be specified below, hold. Function GetPathOrdersAndPathGraph() creates an underlying visibility graph with the edges contributed by the path segments. In addition for each edge of the graph this function creates an ordered list of paths passing through the edge. Function DrawPaths() calculates the final path points.

### PathRefiner.RefinePaths(IEnumerable<Path> paths, bool mergePaths)

The function input is an enumeration of paths to be modified and a Boolean variable mergePaths. The function modifies the paths that the following conditions hold;

* If two paths intersect then both paths have a vertex at the intersection point
* If mergePaths is true then each two paths may have not more than one common sub-path
* No path crosses itself, in other words, each path is a simple curve
* If a path passes through a vertex of another path then the vertex has to belong to the first path too.

We ask to merge paths only in the first call to Calculate() and do not merge paths in the second and third time as an optimization.

# Testing

## Unit Testing Architecture

Testing is based upon the MSTest Unit-Testing architecture via the MsaglTestBase class. The core set of testing utilities for Rectilinear is RectilinearVerifier which inherits from MsaglTestBase. Following that we split by whether we are doing programmatic testing (test methods containing code) via RectilinearTests or file-based testing (test methods that load and execute a test data file) via RectilinearFileTests, both of which inherit from RectilinearVerifier. Additionally, TestRectilinear.exe extends RectilinearTests as described below.

These tests and RectilinearVerifier are closely related to the options available in TestRectilinear.exe. RectilinearVerifier contains a number of members that are set by a programmatic test, a data file, or TestRectilinear.exe, and these modify how the test is run. For example, to evaluate whether a path crosses an obstacle due to ShortestPath or Nudger, run TestRectilinear with the -noNudger option. These options are set directly by the programmatic tests vs. indirectly via Override\* members in the file-based tests, reflecting the fact that the data files set these values directly.

Additionally, the RER itself contains virtual methods that are overridden by RectilinearEdgeRouterWrapper (which inherits from RectilinearEdgeRouter and provides validation for the Unit Tests) and TestRectilinearEdgeRouter (which inherits from RectilinearEdgeRouterWrapper and provides TestRectilinear-specific extensions).

#### Overriding default values

Control values for RER execution are set to non-default values in 4 ways:

* In a programmatic test, via base.<MemberName>
* In a datafile, as read by RectFileReader, via base.<MemberName>
* In a file-based test, via base.Override<MemberName>
* From TestRectilinear, which sets both base.<MemberName> and base.Override<MemberName>

The reason for two members, <MemberName> and <OverrideMemberName>, arises from the fact that RectFileReader sets the <MemberName> values from the data file, and Initialize() is called for each test, so we need another level of override to apply to all datafiles. Therefore the two different branches (programmatic vs. file-based) override default values in the following ways:

* base.Override\* are nullable types and are ignored if null - this means there was no commandline override
* TestRectilinear: sets base.<var> and base.Override<var>
* MSTest -> RectilinearFileTests method:
  + Initialize()
    - ClearOverrideMembers() -> clears base.Override<var> (removes any remnants of a prior method)
    - clears base.<var>
    - calls OverrideMembers() but they've been nulled so nothing happens
  + Test method:
    - sets base.Override<var>
    - calls RunRectFile
      * calls InitializeMembers(reader) which sets base.<var> to what was in the file
        + calls OverrideMembers() -> sets base.<var> to base.Override<var>
* MSTest -> RectilinearTests method
  + Nothing in this chain sets OverrideMembers
  + TestMethod sets base.<var>, similar to how it is set by a RectFileReader
    - Calls CreateRouter or DoRouting
      * calls OverrideMembers() which will do nothing, as there is no TestRectilinear commandline parsing
* TestRectilinear -infile
  + For both -infile and test methods, does commandline parsing and Initialize()
    - TestRectilinear commandline args set both base.<var> and base.Override<var>
  + Initialize() -> does \*not\* call ClearOverrideMembers(); any base.Override<var> is set from commandline
    - * clears base.<var>
      * calls OverrideMembers() which may set base.<var> to base.Override<var> (to a commandline value)
  + -infile
    - This has \*nothing to do\* with RectilinearFileTests; they are separate ways to run the files
      * there is no way to run a RectilinearFileTests.<method> via TestRectilinear
      * Therefore there is no call to ClearOverrideMembers, nor is there a test method which can set base.Override<var>
      * And therefore base.OverrideVar stays with whatever value it has after the TestRectilinear commandline is parsed
    - does commandline parsing and Initialize() as above
    - calls ProcessFiles which calls RecordFileInfoAndDoRouting
      * calls InitializeMembers(reader) which sets base.<var> to what was in the file
        + calls OverrideMembers, which may again set base.<var> to base.Override<var>
  + RectilinearTests() Method
    - TestRectilinear derives from RectilinearTests, so all those methods are methods of the base of TestRectilinear
    - does commandline parsing and Initialize() as above
    - TestMethod sets base.<var>, similar to how it is set by a RectFileReader
    - Calls CreateRouter or DoRouting
      * + calls OverrideMembers() which may again set base.<var> to base.Override<var>

#### Datafiles

RER uses its own format for datafiles, located in the MSAGLTests\Resources\Rectilinear\Data directory. Files are parsed with regular expressions by RectFileReader. The regexes allow an extremely condensed format, which reduces the time and space requirement. Initially these files were much simpler and it was felt that storing all the sections would be useful in regression detection. Subsequently they acquired a number of different sections, and we no longer store data for all sections – for example the VisibilityGraph is no longer written to most of them.

Datafiles contain the commandline that generated them as the first, commented-out line in the file.

##### Creating and Regenerating

Initial creation of a datafile is done by running TestRectilinear.exe with the -outFile argument. In many cases this is done as the result of a failed test from a run of TestRectilinear with a given commandline containing -random or -randrect, and the -reps and -errorlog arguments; the seed that generates the failure is recorded in the errorlog and is then applied to the commandline to generate the datafile that reproduces the failure. Once fixed, the test is checked in to guard against future regressions. For example:

* Overnight run:
  + testrectilinear -random 100 50 -overlap -rotate -ports 0 -1 -reps 100000 -quiet -errorlog e:\temp\errlog\_random\_100\_50\_overlap\_rotate\_ports\_25\_-1.errorlog.txt
* If there are any failures, they will be in the errorlog, together with the seed value for that invocation of TestRectilinear.exe, and the source and target obstacles that were being routed between at the time of failure.
* Run:
  + testrectilinear -random 100 50 -overlap -rotate -ports 0 <targetObstacle> -outfile <newFileName>
* Tf Add newFileName from the commandline and reload the project, or add it in Visual Studio and then edit MSAGLTests and clean up the mess of specific filenames VS added; we have \*.txt already copied so we don’t need the individual file names.
* Create a new test in RectilinearFileTests that is named exactly the same as the newFileName; the tests use the name of the test method as the filename to load.
* Verify the failure reproduces.
* Fix the failure and verify the test passes
* Check in

TestRectilinear supports -updateFile and -regenFile arguments; the former updates from the obstacle etc. definitions in the file, and the latter rebuilds the obstacle definitions from the first-line commandline.

For datafiles derived from .dot files, creating and regenerating requires running testforgdi and then testrectilinear on the original .dot file; this is not obtainable from the first-line commandline. TestRectilinear\Scripts\regenRectFiles.ps1 is now the preferred way to regenerate data files.

For creating datafiles with data from non-default runs of .dot files, use TestRectilinear\Scripts\makeVariationOfDotRectFile.ps1. This takes two arguments: the suffix for the filename, and the additional parameter(s) to add to the TestRectilinear command line to create the file.

It is advisable to store off a recent build of TestRectilinear in two flavors, DebugDevTrace and Release, and then run viewRectFile to compare and judge diffs. This is also useful for debugging. For viewRectFile, pass the parent of the bin\<flavor> directory containing the old build, for example e:\ArchivedTestRectilinear\<date>.

## DebugDevTrace build configuration

DevTrace is DEBUG plus the additional DEVTRACE #define which enables extended in-code checking and tracing via the DevTrace class which wraps Trace operations. These allow simple to verbose trace dumps and can be very useful in tracking problems. Also, some of the DevTrace classes are used to switch within #if DEVTRACE blocks to determine if some time-consuming in-code validations should be performed, based on the level of the Trace switch.

There are multiple instances of DevTrace, one for each major functional area. Modify TestRectilinear.exe.config to determine what level of Trace output is generated.

## #define TEST\_MSAGL

This is defined for DEBUG and DebugDevTrace configurations in MSAGL (Progression turns it off). It wraps utility classes like DebugCurve for visual displays, and supplies some additional in-code validations.

## TestRectilinear.exe

This is the main RER test application, specific to RER. Its implementation derives from RectilinearTests, via which it gains access to the programmatic tests by MethodName. File-based tests are performed by using -infile <filename>.

It has a large number of customizable options, from simply running tests to modifying various parameters for “what-if” scenarios or to analyze test failures.

### Useful Parameters For Test Creation and Debugging

TestRectilinear provides a plethora of parameters for test creation and debugging. The usage message provides an explanation of each; read that for a comprehensive list. Here are a few that are most common and useful.

* RER parameters: These parameters adjust properties of the Rectilinear Edge Router
  + -sparseVg
  + -bendPenalty
  + -portSpliceLimitRect
  + -padding
  + -useRect
  + -routeToCenter
  + -edgeSeparation (which actually maps to RER.CornerFitRadius)
  + -arrowHeadLength
* Scenario creation:
  + -random, -randrect: These lay out either random shapes or rectangles. The graph is formed in a square and each obstacle has its own “cell” of that “grid”, unless -overlap is specified.
  + -overlap: The shapes are allowed to overlap. Otherwise, they are each in their own “cell”.
  + -rotate: Rotates the shapes to test handling of all possible side angles.
  + -ports: A single source->target port pair, unless one of the numbers is -1, in which case paths are formed from all obstacles (if source == -1) and/or to all targets (if target == -1).
  + -sources, -targets: Allow specifying multiple routings, from all those obstacles specified in -source to all those specified in -target.
  + -groups <num> <size> <format>: Creates groups and assigns obstacles to them.
  + -minSize/-maxSize: The size range for each obstacle
  + -seed: The seed from which -random/-randrect and other things obtain a reproducible random layout.
  + -reps: The number of repetitions to run, for example 100000 for an overnight run. Each rep is with a different seed in the stream started by the -seed argument
  + -quiet: Does not show the graph; necessary for -reps
  + -outFile: Writes the results to a rectilinear-format file
  + -errorLog: Writes errors (including the commandline and seed) to this file
* Executing
  + -infile: Loads a rectilinear-format data file
  + -geom: Loads a GeometryGraph (.msagl.geom) file
  + -updateFile: Recreates the file from the obstacle definitions and header information in it
  + -regenFile: Recreates the file from the first line, which is the commented-out TestRectilinear commandline that created it.
  + -showAll: Shows all TestMethods whose names match a regex
  + -runAll: Executes all TestMethods whose names match a regex
* Debugging
  + -noVerify: Do not do verifications. Very useful for bringing up the graph to look at while debugging.
  + -noNudger: Do not execute the Nudger pass. Useful for determining if obstacle overwrites are from ShortestPath (which would be a VisibilityGraph-generation issue) or from the Nudger (normally only from cases with overlapped obstacles).
  + -noPaths: Splice in the ports but do not execute ShortestPaths (and therefore no Nudger).
  + -noPorts: Do not splice in the ports (and therefore no ShortestPaths or Nudger).
  + -interactive: Bring up an Assert dialog instead of throwing an exception. Often used with -noVerify; -noVerify applies only to the Test code, whereas -interactive allows “ignore” of Assert failures in the Production side.
  + -limitRect: Limits the obstacles to those from the original -random/-randrect generation that fall within the given rectangle. To find the desired rectangle, click with the graph and the point you clicked on will be displayed in the console window that launched TestRectilinear.exe.
  + -findObstacle: Click within an obstacle in the graph, copy the coordinate that was printed to the console window, and pass that to -findObstacle. This will ignore all other parameters and print the obstacle ordinal, which can then be supplied to -ports or -sources/-targets.
  + -showVg: Shows the VisibilityGraph in the resultant display; this is after port visibility splices have been removed
  + -showVgPerPath: Show the VisiblityGraph before executing each ShortestPath; this includes the port visibility splices
  + -showPerPath: Displays each path as it is created. Useful for multi-stage waypoints.
  + -showScanSegs: Shows the ScanSegments in the final display
  + -devTrace - send trace output to the file. Modify TestRectilinear.exe.config to determine what level of output is generated.

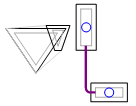
### Scripts

The TestRectilinear\Scripts subdirectory has a number of useful scripts:

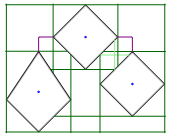
* run\_perf.cmd: Does a timing run and records the result to the Metrics directory of the current working directory; see the file for more detailed usage information.
* testRectFile.cmd: A shortcut to execute a test datafile without having to type in the full path name.
* updateRectFile.cmd: Updates a test datafile. Provides options to allow verification of the tf diff and results and retry with full regeneration if needed – this happens when the read-from-file values are just a tiny bit off from the originally calculated values, which usually doesn’t happen now that we use Convex Hulls as the operations are simplified.
* viewAndUpdateRectFile.cmd: Allows viewing a run of the current build and of a previous, stored-off build – this takes the directory of that stored-off build, brings up TestRectilinear for both build on the same file, and lets you judge the diffs before accepting them and updating the file.
* viewRectFile.cmd: Allows viewing a run of the current build and of a previous, stored-off build – this takes the directory of that stored-off build and brings up TestRectilinear for both build on the same file. Mostly used as a preliminary step in viewAndUpdateRectFile.cmd.
* regenRectFiles.ps1: This allows regenerating either simple rect-format files or .dot-derived rect-format files; prints a usage message.
* makeVariationOfDotRectFile.ps1: Allows doing a run of a .dot file with non-default arguments; takes the new suffix for the result filename and the new args to pass. See for example Resources\Rectilinear\Data\abp.dot\_SparseVg.txt, which was generated by
  + powershell makeVariationOfDotRectFile.ps1 SparseVg –sparsevg

### The Display

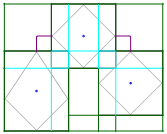
The TestRectilinear display shows the obstacles, paths if any, and optionally other things such as ScanSegments and the VisibilityGraph. Here are some examples:



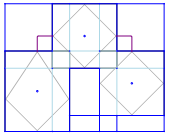
This is a simple path showing dark gray for the obstacle’s InputShape (from the caller), black for its PaddedPolyline (the InputShape padded as specified in the router constructor), a Convex Hull on the left (the light-gray outer outline), and a path between the two obstacles on the right.



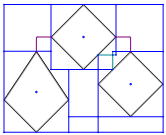
The test Diamond3 with -showScanSegs. Normal ScanSegments are in green, with the lighter-green segments between the middle and right diamonds; these are Reflection segments.



The same Diamond3 test with -showScanSegs, but with -useRect, forcing use of the obstacles’ bounding boxes. This makes the PaddedPolyline a rectangle, so they overlap and have overlapped-weight ScanSegments in light blue.



The same Diamond3 test with -useRect but with -showVg. Normal-weight ScanSegs are dark blue; overlapped-weight ScanSegments are light blue.



Another view of Diamond3 with -showVg; between the middle and right obstacles, the DarkCyan edges are Reflection-derived.

Whenever the mouse is in the display, the status line will show its coordinates. Clicking in the display will write the coordinates of that point to the console that launched TestRectilinear. This is very useful for debugging or reducing; see -limitRect and -findObstacle arguments to TestRectilinear.

## TestForGDI.exe

TestForGDI is MSAGL-general, so most of the TestRectilinear-specific parameters are not there, but those which set properties on the RectilinearEdgeRouter are included. This app does not load rectilinear-format files. It loads .dot files (with the -p argument) or .msagl.geom files (-g or -geom argument). It can also convert .dot to .msagl.geom with the “converttogeom” (no dash) argument, and those .msagl.geom files can be loaded with TestRectilinear (and saved to rectilinear format, which has been done for a number of unit-test files).

## Debugging Tips

Here are some tips for debugging failures in UnitTests and/or TestRectilinear.

### Conditional Breakpoints

Be careful of floating-point rounding. Sometimes the display will round and you won’t get equivalence. If your breakpoint refuses to fire, you may need to use something like “point.X > 22.4 && point.X < 22.5”. There is also a static StaticGraphUtility.IsEqualForDebugger() callable from the debugger, with two overloads that do the foregoing to within 1.0.

### Reducing Datafile failures

Here are some tips to investigate a datafile-failure as revealed by a failure of the RectilinearFileTests method.

* Most failures are on a single path, and the error message will tell you what their ordinals are. Copy the first line of the datafile, which has the TestRectilinear arguments, and replace the -ports specification with the source and target ordinals of the failure.
* Another way to get an obstacle ordinal is by running the command with -findObstacle; click on the display inside an obstacle’s unpadded boundary, copy the coordinates that are printed to the launching console, and add -findObstacle <coordinates> to the command line. Multiple X, Y coordinate pairs may be passed. After obtaining the ordinals, TestRectilinear prints them to the launching console and exits.

### Investigating failures

Here are some tips to investigate failures (after having reduced them from datafiles if needed).

* If the failure is an Assert failure in the production side, rather than Test-side validation:
  + It will probably not provide ordinal information. Generally you can look up the callstack at PortManager.AddControlPointsToGraph and the sourceOport and targetOport will have the obstacles and their IDs.
  + Run with -interactive; this will show the assert-failure dialog rather than simply throwing an exception.
* Run the reduced (with specific source/target obstacles) command line with -noVerify (and possibly -interactive). This will allow TestRectilinear to continue and display the graph. This will give you the reference to look at while debugging. Running with -showVgPerPath will usually be helpful too; this shows the Port visibility splice. If the problem is in the port splicing, then you will probably get several assert failures to Ignore before the graph displays. Another useful argument here is -showScanSegs.
* If there are a lot of overlapped obstacles, run with -noNudger and see if it verifies. We've decided not to worry about cases where the nudger gets confused by heavy overlaps, so this may be a won’t-fix. Verification won’t complain about obstacles being crossed if they are in the same convex hull or clump, but sometimes the Nudger can nudge into obstacles that are not siblings in such an accretion.
* If there are groups, then spatial children may have been enabled if there are any disagreements between hierarchical and spatial group membership (children of a group are outside the boundaries, or non-children are within them). Because we don’t enable spatial children unless needed, it is possible that validation failure could happen on a path that is after the path that caused spatial children to be enabled. In that case, simply running the failing pair will not repro the problem. If you suspect this, set a breakpoint in RectilinearEdgeRouter. RetryPathsWithAdditionalGroupsEnabled, then look up the callstack at PortManager.AddControlPointsToGraph and the sourceOport and targetOport will have the obstacles and their IDs. Then use -sources and -targets to pass two targets when repro’ing the failure.
* As mentioned above, it is advisable to store off a recent build of TestRectilinear in two flavors, DebugDevTrace and Release, for example e:\ArchivedTestRectilinear\<date>\bin\DebugDevTrace|Release. This will let you run viewRectFile.cmd (if it’s a Unit Test failure) or just the free-form commandline against both. The TestRectilinear window will show the location of testRectilinear.exe, so you will know which version you’re looking at.

A couple of the common debugging scenarios follow.

### Debugging failures 1: Port visibility splicing (TransientVisibilityEdge creation)

To debug problems in this area:

* For an ObstaclePort, use the mouse to find the coordinate of the unpaddedBorderIntersect, then set a breakpoint in ObstaclePortEntrance to find that one. A conditional breakpoint can be used, though if you have reduced it as above to a single pair, it is probably faster to just hit f5 a few times.
* The constructor may give you the answer by stepping if it is the unpadded/padded border intersect.
* Otherwise, give that obstacle an Object ID in VS to make it easier to track, then set a (possibly conditional) breakpoint in ObstaclePortEntrance.ExtendEdgeChain.
  + The problem may be before that, on creating the initial target vertex and edge. In that case, back up to where ExtendEdgeChain was called, set the breakpoint there, and re-run. This will probably be a problem in FindOrCreateNearestPerpEdge.
* For a FreePoint, most of the above applies but the initial breakpoint should be in the FreePoint constructor.

### Debugging failures 2: Path crosses an Obstacle

To debug problems in this area:

* If there are a lot of overlaps, run with -noNudger and see if it verifies. If so, then as noted above, this is probably a won’t-fix.
* Validation will not complain about crossing obstacles that are siblings in a clump or convex hull, so visually inspect the graph and see if it looks like they are siblings; if so there may be an error in validation
* Normally the error message will tell you which target; get the first line from the file and repro with that target (ports 40 -1 becomes ports 40 25, for example)
* Run that with -noverify and -showVgPerPath; you'll get the window up and can look at it and see from the error message what path section to look at. -showvgperpath is important to look at, to see the port splicing before the path starts to route; that's usually the problem.
* If there's a freePort it won't show source and target ids but will show the port coords; find that in the file's port definitions, find the routing spec for the pair, and copy the file to a temp file that has only that pair and run it.

## Other test notes

A number of tests reflect cases with tricky overlaps that caused problems for intersecting angled sides in the scanline. These tests now form convex hulls, but have been retained to ensure we handle these cases. We do not care much about the attractiveness of the paths for heavily-overlapped cases; the absence of catastrophic failure is generally sufficient. However, the path output is retained and verified so that examining diffs can reveal unexpected behavior applicable to less-overlapped cases.

There are two varieties of test files:

* Those derived from .dot files, which will be named for the dot file, such as abp.dot(\_<suffix>).txt. <suffix> indicates nondefault arguments passed; currently these are for HighBendPenalty, SparseVg, and RouteToCenter.
* All other files are generated by runs of TestRectilinear with one of -random or -randrect and any number of other arguments.

# Merging with Progression

“Progression” is the name for the Dev11 Visual Modeling tool, which consumes RER (the old Tuvalu group is now in Progression). There are 4 directories that should remain in sync when merging to Progression:

* GraphLayout\MSAGL\Routing\Rectilinear (and subdirectories, currently only Nudging)
* GraphLayout\MSAGLTests\Rectilinear
* GraphLayout\MSAGLTests\Infrastructure\Rectilinear
* GraphLayout\MSAGLTests\Resources\Rectilinear\Data

# Unimplemented areas

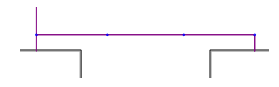
The following areas are not implemented in this release.

## Out of Scope

The following features are considered out of scope for this release.

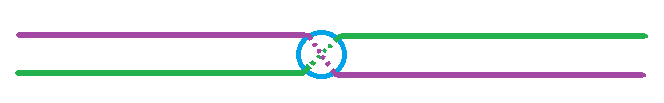
### Waypoints With Dimension

When multiple paths go through a waypoint, they are currently squeezed together, because the waypoint has no size and thus the Nudger does not separate paths that go through them. This can make for a graph that is not as easy to read as it could be. This is illustrated in 1a; it is desirable to provide some amount of separation as in 1b.

1a. 1b. 

The Nudger considers restrictions only for the channels formed by the surrounding obstacles, and the dimensions of the port curves at the start/end of each path. It does not allow ports in the middle of the path, and it does some initial path modifications (such as removing staircases) which can pull a path off its waypoints. Therefore, the current approach for waypointed paths is to create a globally optimal path in multiple stages, where a stage is source -> waypoint 1, waypoint1 -> waypoint2…, waypointN -> target. Then each of these stages is nudged as a separate path, and finally the stages of each waypointed path are combined into a single path.

One problem with nudging paths with multiple stages through a point with dimensions is that each stage may be nudged in separate directions, since the connection between stages is not known to the Nudger:



Some implementation possibilities here are:

1. Do not modify current Nudger behavior, and instead let the app try to handle this. For example it may decide whether to group multiple waypoints near a given location, assigned to separate paths and separated by whatever spacing the app desires. Selecting which of such grouped waypoints should be assigned to which path specifications would be difficult to precalculate given that obstacle layout makes it difficult to predict shortest-path behavior.
2. Pass only a single path to the Nudger and modify the Nudger to be aware of mid-path waypoints, possibly by simulating a narrowing of the channel at that point for any path for which it is a waypoint (and avoiding moving those paths outside that point during the initial path-adjustment phase).
3. Continue to pass multiple paths to the Nudger and modify the Nudger to add path-specific constraints around the endpoints at WaypointPorts, such that adjoining endpoints of stages must be equal, and those endpoints may only be nudged within a certain distance of the waypoint (the waypoint’s size).

## Potential Improvements

### Additional Nudging pass to utilize between-ConvexHull space after staircase removal

As mentioned above in the Convex Hull discussion, the Nudger removes staircases between Convex Hull borders but does not guarantee that the path is positioned evenly in the free space. This would require an additional run of the Nudger, which has performance implications, and accuracy improvements for highly-overlapped graphs is not priority.

### Splicing out from overlapped obstacles to free space outside clump

Currently Port visibility splicing stops at the first “neighbor”, even if this neighbor is a member of the same clump. In some cases this renders a path with more bends. This could be modified to splice to the first neighbor outside the clump, probably by modifying how the MaxVisibilitySegment is calculated in this case, and modifying TransientGraphUtility.ExtendSpliceWorker asserts to include the “in same clump” allowance. However, accuracy improvements for highly-overlapped graphs is not a priority.

### Bend reduction in Sparse VisibilityGraph

In some cases, we don’t have all the vertices that would yield an optimal path in SparseVg. Here are some (very speculative) possibilities that could be investigated, determining if the tradeoff of VisibilityGraph growth helps accuracy:

* Currently the opposite half of the graph is not given any intersections for a vertex. If a vertex is, for example, just below the halfway point, then there are no intersections in the upper half of the graph between the middle and the far border. One possibility is to add those intersections until we get to the same distance above the halfway point as the vertex is below, as in these two examples for a graph of width 100 (that is, 100 vertical intersections on the X axis):
  + The vertex is at 40. In that case, add intersections at 75 (half of the upper half), 62 (one-quarter), 56 (one-eighth), and stop because now we are closer to the halfway point than the vertex is.
  + The vertex is at 10. Add an intersection at 75 and stop.
* Do a fuller intersection of ScanSegments that span some portion of the graph. For example, for any that span more than half the graph, add some additional intersections, skipping adjacent ScanSegments of the same length. This could still grow the graph quite a bit with no real gain in accuracy.
* Add more “corner” points to obstacles at distances further from the obstacle.

### Eliminate Shallow Bends in Obstacle Sides

With Convex Hulls we have eliminated the major sources of difficulty from floating-point inconsistencies, but we could make this a little more bulletproof by enforcing that all obstacle sides will have a minimum angle from either axis, for example 1/8. Lev had a good idea for doing this for 45 degrees that involved projecting as an octagon rather than a square; this might be an even simpler and more robust restriction. However, the larger the angle required, the higher the likelihood of additional transitive convex hull creation.

### Performance

Following are a few performance areas that could be investigated; profiling should be done first to evaluate time spent in relevant areas.

#### Direct visibility intersection for Waypoints

Currently we only support direct visibility intersection, bypassing Port splicing and ShortestPaths, for ObstaclePorts. This is potentially a perf improvement for Waypoints as well, but since Waypoints are a composite of multi-stage paths, this would need to evaluate whether picking the direct path(s) to the next waypoint prematurely eliminates a non-direct visibility segment that may lead to the globally optimal path.

#### Combine all Shortest-Path Sources into a single pass

This would only reduce a fraction of Shortest-Path cost, because we have cutoffs that are designed to bail out as soon as we exceed some distance from the best cost, and we start with the sources and targets that are closest together. If this were done it would have to send in a delegate to get the initial cost (this would calculate the difference between the individual source vertex and the rectangle center of the ports). Then we would simply put those vertices on the stack rather than just the single one as we do now.

#### Make a Lookup Table for Direction-converting CompassVector methods

Currently there are a few Direction-converting CompassVector methods that incur a function call to a switch statement to do things like OppositeDirection, RotateLeft or RotateRight, or convert a Direction to an index. If the function call count warrants it, it should be possible to inline these by using a small lookup table of size 9, where the Direction’s enum value is used as the index and non-power-of-two entries are unused.

#### Cache the First Splice Intersection in ObstaclePortEntrance and FreePoint

For an obstaclePort or freePoint that has more than a couple calls, this would reduce the amount of time doing the initial PortManager.FindOrCreateNearestPerpEdge, storing off the VisibilityEdge and its intersection rather than doing the ScanSegmentTree lookup and then walking to the desired edge. This would have to cache the ScanSegment as well, in case the initial point is before or after it.

#### Reduce GC impact of Port Splicing and Shortest Paths

With Direct Visibility intersection bypassing these in many cases, the benefits may not be so much, but it still may be worth a look as we can create a large number of very short-lived objects in these phases.

* Have VertexEntry derive from GenericHeapElementWithTimestamp
  + This would allow GenericBinaryHeapPriorityQueueWithTimestamp to avoid creation of a GenericHeapElementWithTimestamp to wrap the T element, and allow using a cast rather than the lookup dictionary.
* TransientGraphUtility recycling
  + For large graphs, TransientGraphUtility creates a number of VisibilityVertices and TransientVisibilityEdges. We already maintain a list of these for cleanup, and it may reduce load on the GC to recycle them to the next path.