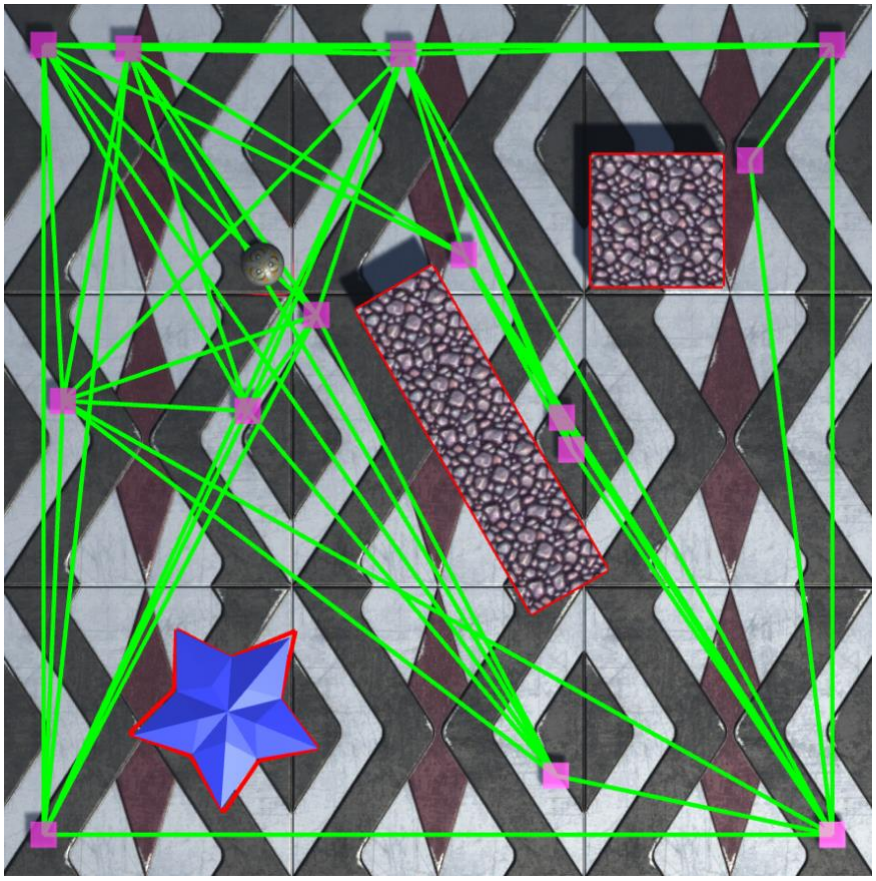
Homework 2: Path Network Navigation

One of the main uses of artificial intelligence in games is to perform *path planning*, the search for a sequence of movements through the virtual environment that gets an agent from one location to another without running into any obstacles. For now, we will assume static obstacles. In order for an agent to engage in path planning, there must be a topography for the agent to traverse that is represented in a form that can be efficiently reasoned about.

Grid topologies discretize the environment and usually assumes an agent can be in one discrete cell or another. However, for many games such as 1st-person shooters, a more continuous model of space is beneficial. Depending on the granularity of the grid, a lot of space around obstacles becomes inaccessible in grid-based approaches. Finally, grids result in unnecessarily large number of cell transitions.



A **path network** is a set of path nodes and edges that facilitates obstacle avoidance. The path network discretizes a continuous space into a small number of points and edges that allow transitions between points. However, unlike a grid topology, a path network does not require the agent to be at one of the path nodes at all times. The agent can be at any point in the terrain. When the agent needs to move to a different location and an obstacle is in the way, the agent can move to the nearest path node accessible by straight-line movement and then find a path through the edges of the path network to another path node near to the desired destination.

In this assignment, you will be provided with different terrains with obstacles and hard-coded path nodes. You must write the code to generate the path network, as a set of edges between path nodes. An edge between path nodes exists when (a) there is no obstacle or boundary wall between the two path nodes, (b) there is sufficient space on either side of the edge so that an agent can follow the line without colliding with any obstacles or boundary wall and (c) a path node that is inside the boundary but outside of any obstacles. Nodes that are outside of the boundary or inside of obstacles, should not connect.

Furthermore, there should be edges in both directions. If there is an edge from Node A to Node B, then there should be an edge from Node B to Node A. Also, if there should be an edge from Node A to Node B, it should appear exactly once in the edge list for Node A (no duplicates). Also, no nodes should link to self.

What you need to know

Please consult **homework 1** for background on the Unity Project. In addition to the information about the game engine provided there, the following applies.

CreatePathNetwork

This is the only file you will be modifying and submitting for this homework. It provided functionality to create a path network from provided pathNodes.

String StudentAuthorName – Please change to your name

Create()

This is the method you will be finishing. You can create helper methods in the same source code file if you like.

Parameters:

- Vector2 canvasOrigin – Bottom left corner of navigable space
- float canvasWidth – Width of navigable space
- float canvasHeight – Height of navigable space
- List<Polygon> obstacles – The polygon obstacles that obstruct candidate pathEdges between nodes. Furthermore, end points cannot be within agentRadius distance (less-than test) of a candidate pathEdges
- float agentRadius – The radius of the agent (don't expect to always be the same)
- float minPoVDist – The minimum distance a single point of visibility can be from the vertex or lines of angle that is bisected. Only used when generating PoVs
- float maxPoVDist – The maximum distance a single point of visibility can be from the vertex or lines of angle that is bisected. Only used when generating PoVs

ref List<Vector2> pathNodes – The nodes of the graph. You use them as provided if mode is PathNetworkMode.Predefined. But you generate them if mode is PathNetworkMode.PointsOfVisibility

out List<List<int>> pathEdges – A list of valid edges, indexing the pathNodes. This is generated by your code. All valid edges should be generated

PathNetworkMode mode – enum that is either PathNetworkMode.Predefined. or PathNetworkMode.PointsOfVisibility. The first specifies that path nodes are provided to Create() for connection. The second specifies that path nodes must be generated according to PoV algorithm.

PathNetwork

The PathNetwork calls your CreatePathNetwork.Create() to generate the path network and also performs visualization and other related tasks. You don't need to modify anything in this file, but may find it useful to see how your code is called.

PathNetworkTest

This file is for unit/integration testing. Feel free to create your own tests.

Miscellaneous utility functions

Methods you need for this assignment are provided at the top of CreatePathNetwork.cs
Also, you will access Polygon member methods such as getIntegerPoints() and getPoints()

In terms of Computational Geometry, the DistanceToLineSegment() works best with floats. You can use the poly.getPoints(), which are unscaled floats. The poly.getIntegerPoints() are the equivalent vectors in scaled integer form (by 1000).

Instructions

To complete this assignment, you must (1) implement code to generate a path network for a set of given path nodes in a given scene. The path network should guarantee that an agent will not collide with an obstacle between any starting point and any destination point in the world.

It is not required, nor graded, but you can consider implementing Points of Visibility support as well. See below for details.

To run the project code, use the following commands:
> **Step 1:** Download the project from github. Open the project in Unity. Open scene PathNetwork
> **Step 2:** If you hit play, you should see a list of path nodes and some obstacles not connected to each other. You can click the obstacles and path nodes to drag them around.

You can press buttons 1,2,3,… to test some preconfigured obstacle/pathNode positions. Clicking outside of an obstacle initiates path planning. Spacebar changes search strategies. Note that any search that relies on AStar won't function until you implement it.
> Step 3: Create the edges in the path network.
Open Assets/Scripts/GameAIStudentWork/PathNetwork/CreatePathNetwork.cs. Update the name string first! Implement the functionality to generate a valid list of PathEdges connecting PathNodes that are not obstructed from each other. Also, obstacles cannot be too close to PathEdges for the agent to fit by while following an edge.

Grading

We will grade your solution based on the following criterion:

- **Reachability:** The path network should be such that an agent can navigate from any path node to any other path node along any edge in the network without colliding with an obstacle or canvas/world boundary. No edges that are out of bounds (outside boundary or inside obstacles).

- **Graph Characteristics:** Valid edges should have a corresponding edge going opposite direction between same two nodes. There should be no duplicate edges in each edge list. Nodes should not have an edge directly back to itself. Edge lists should never be null (empty is allowed). Edge lists should not reference non existent node indexes. PathNodes and pathEdges should be the same length.

- **Points of Visibility:** You don't need to respond to the PathNetworkMode mode parameter. But you can optionally implement Points of Visibility support using the technique described below.

Please remove all print statements before submitting. The autograder will only provide a few hundred lines of feedback and you might overflow the buffer so that the informative part doesn't show up. Also, print statements can cause significant slowdown such that you might fail a test due to timeout (see below). When you remove print statements, please test your code again! Quite often we receive assignments that don't compile due to a hanging *if-statement* where a print() was the consequent.

Your code will be allowed at least 10 seconds to complete each test.

Hints

Make sure any edge in the path network is traversable by an agent that has physical size. That is, edges in the path network should never come too close to any Obstacle such that an agent blindly following the path edge collides with an Obstacle (or the edges of the map).

Create additional presets by adding more configurations to
Assets/Scripts/Config/CustomPresetConfig.cs
However, you do not submit this file.

Also, create Unit/Integration tests with PathNetworkTest.cs (also do not submit).

Submission

To submit your solution, upload your modified CreatePathNetwork.cs with your name properly set in the name string. All work should be done within this file.

You should not modify any other files in the game engine (other than optionally CustomPresetConfig.cs for your own testing purposes).

DO NOT upload the entire game engine.

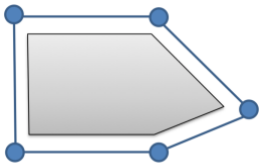
Advanced Method

Not required, but you might optionally try to implement Points of Visibility if the mode parameter is set to PointsOfVisibility.

Points of Visibility Algorithm

There are different ways one might go about placing points of visibility (PoV) around an obstacle. Consider the following algorithm.

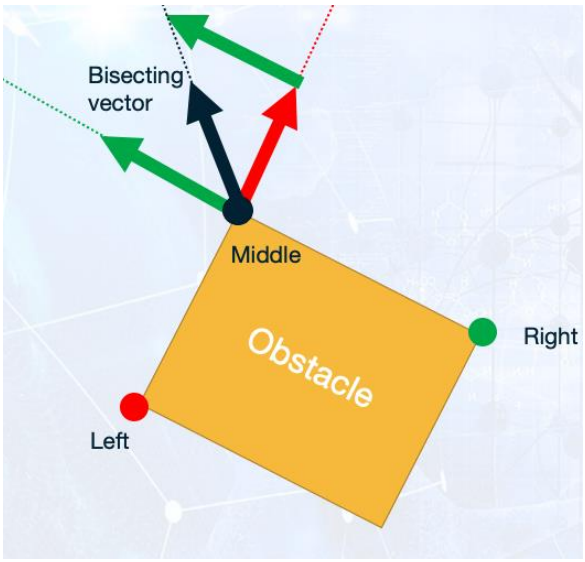
First, recall that PoVs are placed at convex angles of polygons, and offset by a distance that leaves room for the agent.



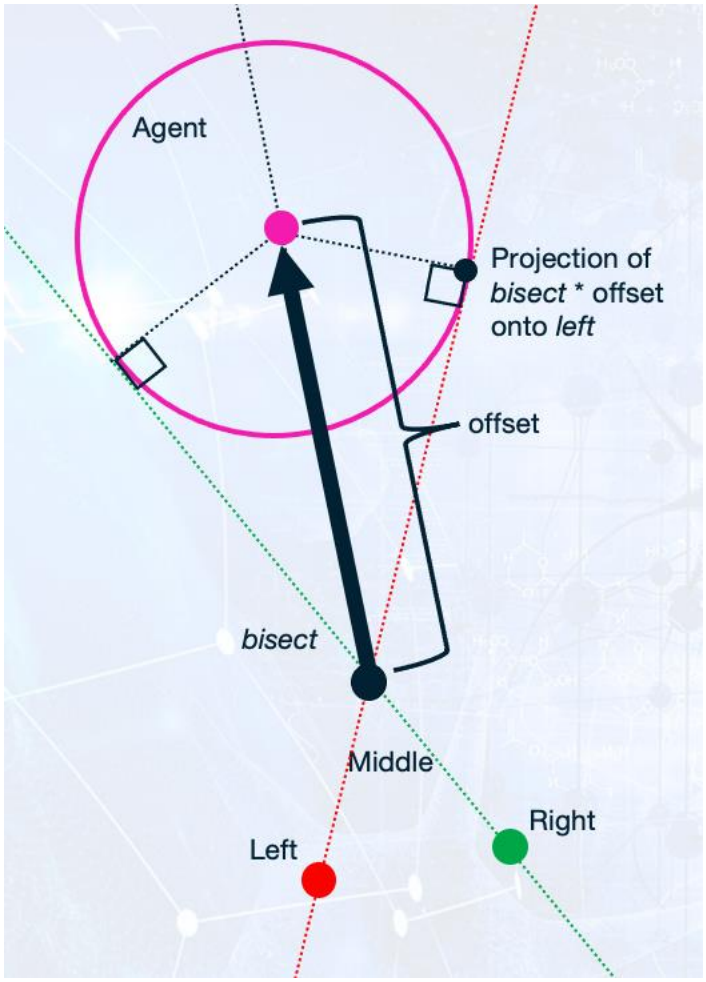
This is straightforward to achieve. The basic approach is:

- Iterate through the vertices of the polygon in counter-clockwise order, selecting each sequence of 3 vertices (*right, middle, left*).
- For each sequence, check if the angle at the *middle* of the three vertices is convex.
    - You can use Left() to see if the vertex *left* is to the left of the vector from *right* to *middle*.

- If not convex, skip to the next sequence of 3 (advance by 1 position).
- Now determine the vector that bisects the angle
    - Compute normalized vectors *left→middle* and *right→middle*
    - Find the sum of these two vectors and normalize the result to get the bisecting vector direction (vector that bisects the angle)



At this point, you can simply offset from vertex *middle* by the normalized bisecting vector multiplied by some distance scalar (like *agentRadius*). If you stop here, you will successfully create offset vertices that can be used as candidate path nodes. Unfortunately, there is a problem. You can easily end up with candidate edges that cannot connect in a circuit around the obstacle. This happens if the edge gets too close to the obstacle for the agent to have room. The problem is more likely with acute angles. To address the issue, the PoV can be placed further out. In the following figure, see how the PoV needs to be offset far enough along the bisecting vector so that there is clearance to go in a straight line along the obstacle either to the left or right. This is achieved when the agent can fit entirely within the bisected angle.



This offset can be determined be defining a system of equations to determine how big the offset needs to be in order for the distance between the end *of bisect*offset* vector and its projection onto *the left → middle* vector to be at least the minimum allowed PoV distance (*minPovDist*). (The projection could be made onto the *right → middle* vector as well.)

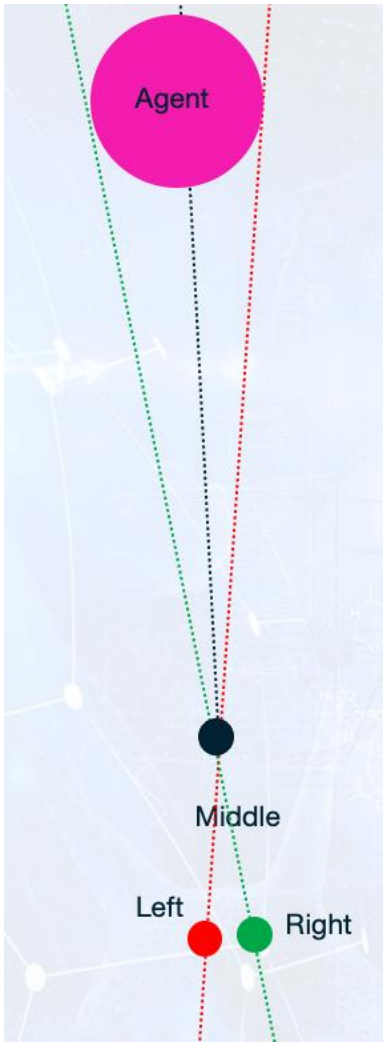
Vector Projection equation:

$$\boldsymbol{a}_1 = \frac{(\boldsymbol{a} \cdot \boldsymbol{b})\boldsymbol{b}}{\|\boldsymbol{b}\|^2}$$

Vector **a** is *bisect*offset*. Vector **b** is *left → middle*. (Since *left → middle* is normalized, the denominator is 1.0.) Now, we know that:

$$minPoVDist = \|\boldsymbol{a} - \boldsymbol{a}_1\|$$

Finally, solve for *offset*. Now you can figure out how far to place the PoV so that the agent can go in a circuit around the obstacle. However, there is still a problem. Consider an extremely acute angle.



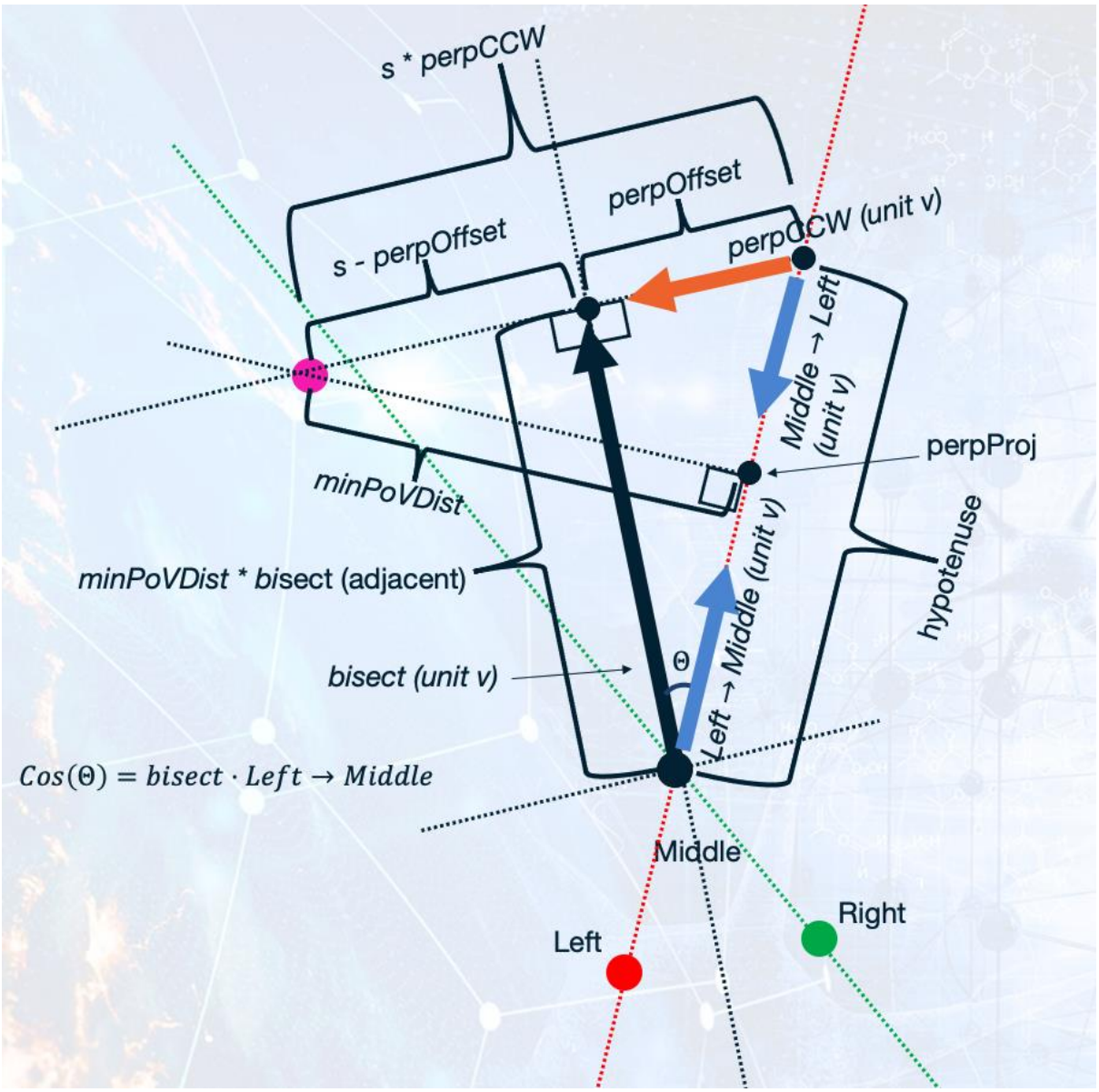
The PoV must be placed arbitrarily far away from the obstacle vertex depending on how acute the angle is. A reasonable solution is to determine if *offset* is beyond some threshold (*maxPoVDist*). If so, split the PoV into two separate PoVs that “square” off the end.

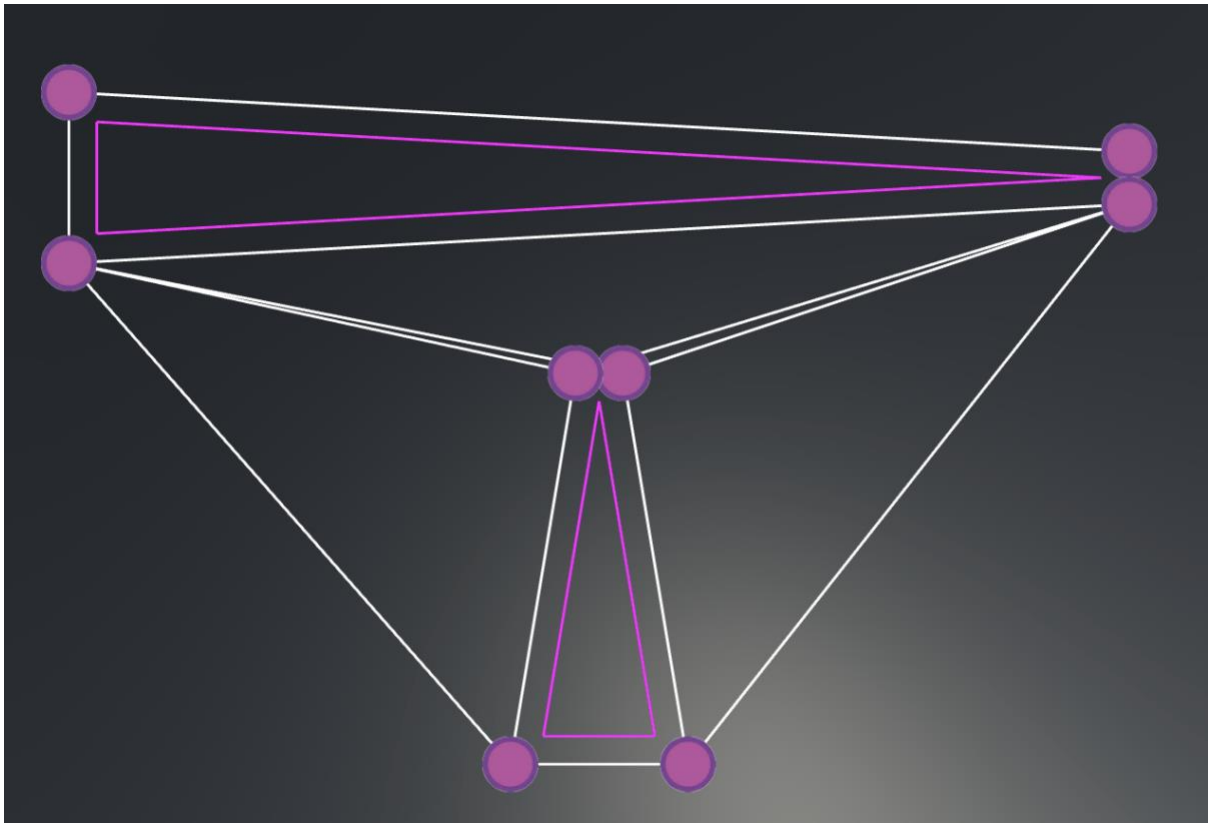
Once you have determined that two PoVs are needed, use the following strategy for placing the two PoVs.

- Find *minPoVDist * bisect*. This vector is just far enough away from *middle* vertex for the agent to have a clearance from the obstacle. You just need to offset some amount to the left and right perpendicularly to place the two points. The exact amount must be calculated.
- Find the dot product of *bisect* and *left → middle*. Recall that the dot product of two normalized vectors is the Cosine of the angle between the two vectors.
- Now you can use the trigonometry equation of right triangles, *Cos(theta)=adjacent/hypotenuse,* to solve for the distance down the length of vector *left->middle* (e.g., the hypotenuse)
- Next, use Pythagorean Theorem to determine the remaining leg of the triangle. This is the *perpOffset* distance from the end of vector *minPoVDist * bisect* perpendicularly

to the intersection point on *left → middle*. This offset will be used to help determine how far to place our points to left and right relative to *minPovDist * bisect*.

- Now, it's time to figure out how far to offset to the left and right. We'll use a similar projection technique as before.
- Find a perpendicular vector to *bisect* called *perpCCW*. E.g., PerpCCW(v) = (-v.y, v.x) and PerpCW(v) = (v.y, -v.x)
- Find the opposite vector to *Left → Middle,* called *Middle →* $Left$ (you can just use negation to flip the vector).
- Set up a system of equations by projecting vector *s*perpCCW* (where *s* is an unknown scalar) onto *Middle →* $Left.$ The projected point is called *perpProj*. Also, create an equation solving for distance between *s*perpCCW* and *perpProj* to be *minPoVDist*. Solve for *s.*
- Once you know *s,* you can finally place your points perpendicularly to the left and right of the end of *minPoVDist * bisect* by (*s – perpOffset*) multiplied by either perpCCW or -perpCCW.





This method for placing PoVs works well, especially for convex obstacle polygons, and many concave polygons. However, there are some cases of concave polygons with extreme concavity where the algorithm may not result in path nodes that can connect when they potentially could if placed better. More robust strategies based on expanded geometry may be necessary in those cases.