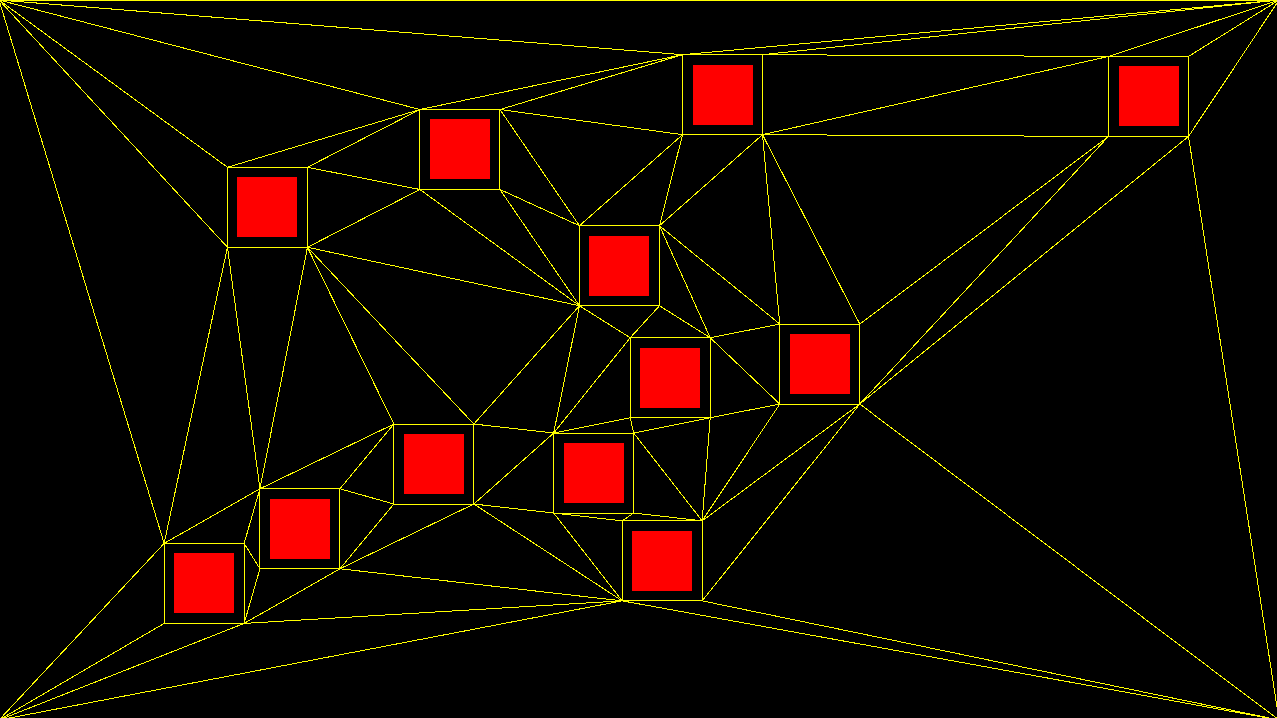
Tutorial – Navigation Mesh

Navigational Mesh, aka Nav Meshes, are used in many games and game engines.

For this task you will be provided with a starting application. This application makes use of one external library and a funnelling algorithm that aims to smooth out paths generated by A\* or a similar pathfinding algorithm over a graph made up of triangles.

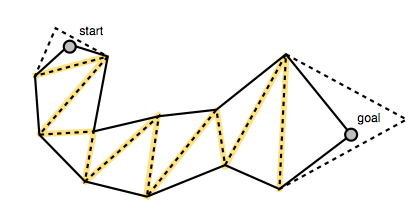
The external library used is a simple library called **Poly2Tri**, available at <https://github.com/greenm01/poly2tri>, which can create a triangle mesh from input polygons and can cut holes in the polygons based on other polygons. The application provided includes this library and generates a triangle mesh with square holes cut into the mesh. The squares must not overlap. We also cut the holes slightly larger than the obstacle so that the nav mesh leaves enough space for the size of our game objects to pass the obstacle safely, so we add some padding when the holes are cut.

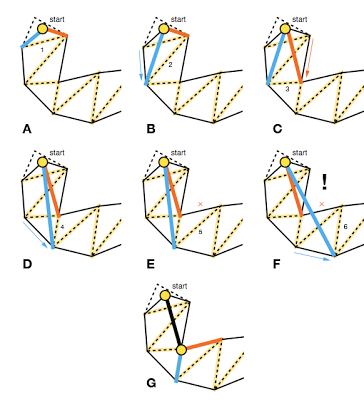


There are three tasks that need to be completed for this exercise:

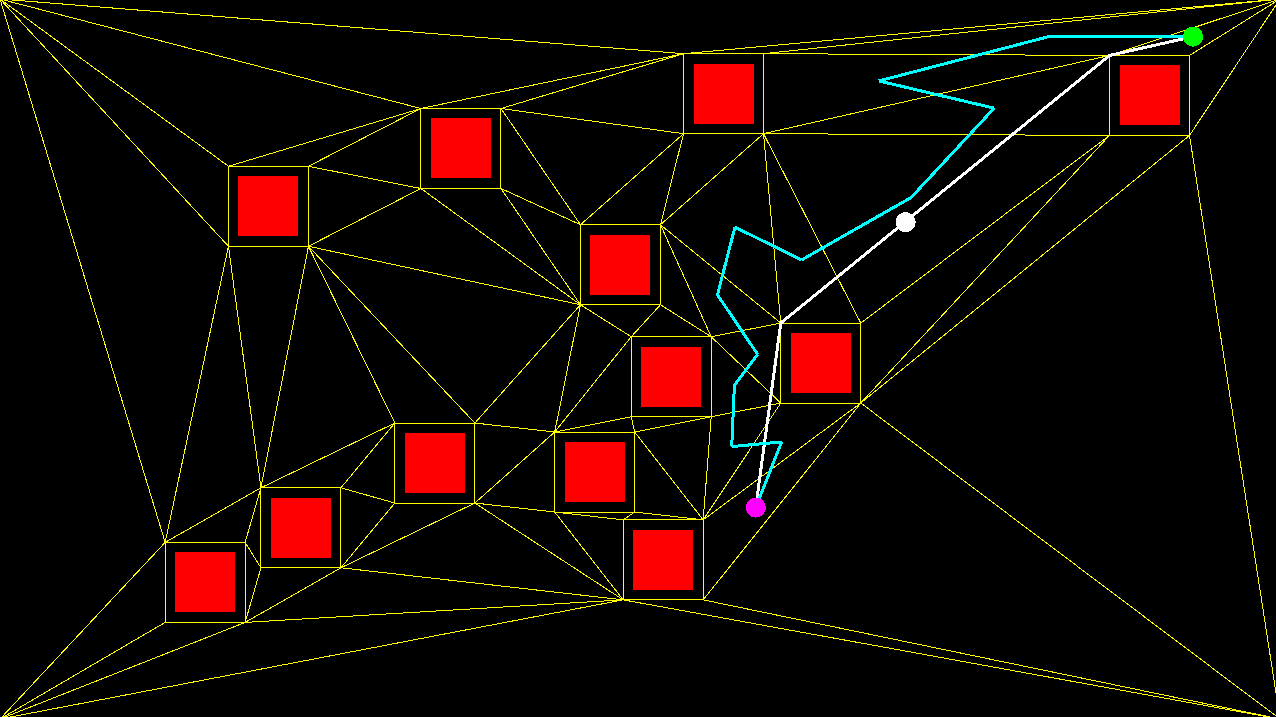
1. Convert the triangles into navigation nodes for pathfinding.
2. After an A\* path through the triangles has been found, implement path smoothing using a funnelling algorithm provided.
3. Add a game object that finds a path through the nav mesh, smooths it, then follows the smoothed path.

All nodes are triangle, so connecting nodes will always share a triangle edge of two vertices. The path through the mesh is then a triangle strip.

To create the shortest line through the strip of triangles we can use a funnelling algorithm that recursively traverses through the shared triangle edges to find bends. When a bend is found a line from the start to the bend is added to the shortest path list, then the algorithm restarts from the bend location traversing through remaining edges to find more bends. It finds bends by comparing vectors from the current bend point, or start point, to the vertices made by the shared edge. If the vectors cross over then a bend is found and becomes the new point to test from. The following image shows the algorithm in action:

This continues until the shortest path is found. The algorithm we will use is taken from <http://digestingduck.blogspot.com.au/2010/03/simple-stupid-funnel-algorithm.html>.

The following image shows the A\* path through the triangles as a light blue line. A short optimal path is found using the funnelling algorithm, displayed as a white line that a white game object is following:



But before we can do that we must turn the triangles into nodes.

Converting Triangles to Nodes:

The provided **NavMesh** project contains a **NavMesh.h** and **NavMesh.cpp** that implements a **NavMesh** and related classes, including a **NavMesh::Node** that derives from the **Pathfinding::Node** from a previous tutorial.

There are two parts within the **cpp** that need to be implemented; the **NavMesh::build()** method and the **NavMesh::smoothPath()** method, but we’ll get to **smoothPath()** later.

*Look through the entire project first and ensure that it builds and runs for you. This may mean that you have to modify the project slightly.*

Once you are ready, look at the **NavMesh::build()** method:

We need to first create a **NavMesh::Node** for each triangle, then find triangles that share an edge and create a link between them. Since **NavMesh::Node** inherits from **Pathfinding::Node** we create standard links between the nodes with a cost that is the distance between the triangle centers.

void NavMesh::build() {

m\_cdt->Triangulate();

#pragma message("TODO: Convert triangle mesh to path nodes!")

// first convert triangles to NavMesh::Node's

// then link nodes that share triangle edges

// cleanup polygons

for (auto& p : m\_polygons)

for (auto ptr : p)

delete ptr;

m\_polygons.clear();

// close up Poly2Tri

delete m\_cdt;

m\_cdt = nullptr;

}

We can access the triangles made with **Poly2Tri** through the variable **CDT m\_cdt**. It has a method **GetTriangles()** that returns a **std::vector<Triangle\*>** that we can then loop over. All triangles have a **GetPoint()** method that takes in the index of the triangle vertex, from 0 to 2. Using these we can create **NavMesh::Node’s** and set their position to be the average of the triangle corners.

We also need to store the triangle corners within the node for funnelling later.

The following code does just that, and should go within the **build()** method after the first comment:

// first convert triangles to NavMesh::Node's

std::vector<p2t::Triangle\*> triangles = m\_cdt->GetTriangles();

for (auto tri : triangles) {

NavMesh::Node\* n = new NavMesh::Node();

n->vertices.push\_back({ (float)tri->GetPoint(0)->x,

(float)tri->GetPoint(0)->y });

n->vertices.push\_back({ (float)tri->GetPoint(1)->x,

(float)tri->GetPoint(1)->y });

n->vertices.push\_back({ (float)tri->GetPoint(2)->x,

(float)tri->GetPoint(2)->y });

n->position.x = (n->vertices[0].x + n->vertices[1].x +

n->vertices[2].x) / 3;

n->position.y = (n->vertices[0].y + n->vertices[1].y +

n->vertices[2].y) / 3;

m\_nodes.push\_back(n);

}

Looking at the code you can see that we store the triangle corner points within a **std::vector** called **vertices**, and that we average the three points to get the position of the node.

Next we need to link connected triangles.

To do this we simply loop over all of the nodes and for each node, loop over all of the other nodes and test if they share a triangle edge. They share an edge if two of their vertices are shared. Triangles share either 1, 2 or 0 vertices, but they only share an edge if it is 2. If they share an edge we simply make a link between them with a cost of the distance between the node positions.

Note there are other ways to do this. Not all navigational meshes use triangles, and there are different ways to link them and assign cost.

The following code goes within **build()** after the second comment:

// then link nodes that share triangle edges

for (auto node : m\_nodes) {

for (auto node2 : m\_nodes) {

// ignore same node

if (node == node2)

continue;

// share verts?

int sharedVerts = 0;

for (auto& v : node->vertices) {

for (auto& v2 : node2->vertices) {

if (v.x == v2.x &&

v.y == v2.y)

sharedVerts++;

}

}

// link if two verts shared (should only ever be 0, 1 or 2)

if (sharedVerts == 2) {

float mag = (node2->position.x - node->position.x) \*

(node2->position.x - node->position.x) +

(node2->position.y - node->position.y) \*

(node2->position.y - node->position.y);

// add links to both nodes

node->edges.push\_back(Pathfinding::Edge(node2, mag));

node2->edges.push\_back(Pathfinding::Edge(node, mag));

}

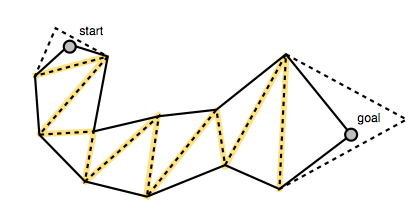
}

}

If you were to run the application you should now be able to see the triangle navigational nodes drawn with yellow lines, and you would be able to run A\* or Dijkstra’s over the graph of nodes to get a path through it.

Now we need to implement the funnelling algorithm for path smoothing.

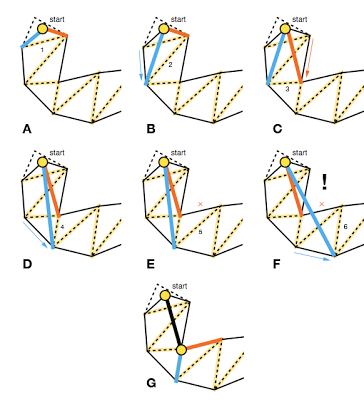
Path Smoothing:

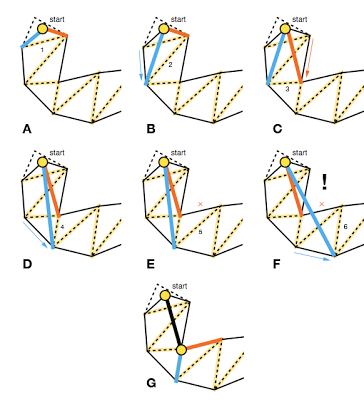
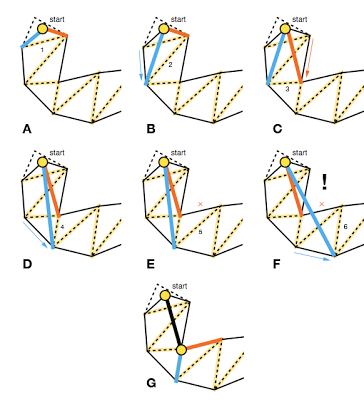
If you run a pathfinding algorithm on the graph of nodes you will receive a list of nodes to travel through, but the triangles may cause the path to be very jagged.

We could simply follow the path and move a game object along it, heading towards each triangle’s centre and then when we reach it, or enter the triangle, we head towards the next, but this will move the game object along an awkward and unrealistic path. What we need is to find the shortest path.

The shortest path through the navigation mesh would be one that draws straight lines from the start point to the first bend, then the next, and next, so on and so forth until we can see the goal point from a bend.

An algorithm that achieves this is a funnelling technique that detects these bends and adds them to a new path our game objects can follow. The funnel is the strip of triangles, and the result of the algorithm is the shortest path through the strip.

The actual funnelling algorithm has been provided for you, **NavMesh::stringPull().** We give the algorithm a list of “**portals**” which are the shared triangle edges containing 2 vertices. We need to ensure that the vertices used for all edges are arranged in a similar way so that all vertices on the right of the funnel are first in each edge, and the vertices on the left are second. The algorithm then iterates through these vertices until they cross each other, which means a bend was found.



The task we need to complete is that the list of edges for the “portals” hasn’t been built.

The function **NavMesh::smoothPath()** needs to be implemented:

int NavMesh::smoothPath(const std::list<Pathfinding::Node\*>& path,

std::list<Vector2>& smoothPath) {

if (path.size() == 0)

return 0;

smoothPath.clear();

#pragma message("TODO: Build list of edges along the path to smooth!")

return smoothPath.size();

}

The job of this method is to create the portal list based off of a path through the nav mesh that has been found with an algorithm like A\*, then pass the list to the **stringPull()** method to fill in a smoothed path that is stored within the reference parameter **std::list<Vector2>& smoothPath**.

Getting the list of shared edges is easy enough, since we can query a **NavMesh::Node** against another node to find the vertices they share with a call to **NavMesh::Node::getAdjacentVertices(),** but it isn’t quite that simple.

We need to ensure the vertices are in the correct order.

To begin with though the funnelling algorithm requires that the first portal be made up of two points that are both the position of the starting point, and it requires the last portal to be made up of two points that are both the position of the goal point. The most amount of portals that will exist is 1 more than how many nodes there are to traverse.

We can remove the #pragma message and start to replace it with the following code.

We start by allocating an array of **Vector2** to represent the portals, 2 Vector2 for each portal.

We then add the starting node’s position as the first two Vector2, then we will loop over all nodes to gather the other portals, and finally add the goal node’s position as the last portal.

We then pass the portal list into the **stringPull()** method. The following is the updated **smoothPath()** without the loop filled in yet:

Now we will look at implementing the loop.

int NavMesh::smoothPath(const std::list<Pathfinding::Node\*>& path,

std::list<Vector2>& smoothPath) {

if (path.size() == 0)

return 0;

smoothPath.clear();

// build portal list

int index = 0;

Vector2\* portals = new Vector2[(path.size() + 1) \* 2];

// add start node as first portal

portals[index++] = ((NavMesh::Node\*)path.front())->position;

portals[index++] = ((NavMesh::Node\*)path.front())->position;

// LOOP TO GO HERE!

// add last node as end portal

portals[index++] = ((NavMesh::Node\*)path.back())->position;

portals[index++] = ((NavMesh::Node\*)path.back())->position;

// run funnelling algorithm

Vector2 out[100];

int count = stringPull(portals, index / 2, out, 100);

// gather up shortest path

for (int i = 0; i < count; i++) {

smoothPath.push\_back(out[i]);

}

// cleanup and return length of path

delete[] portals;

return smoothPath.size();

}

The loop requires us to keep track of the current node and the previous node. This means we skip the first node but tag it as previous, and always update the previous pointer after each loop.

The following replaces the comment within the previous code snippet, **// LOOP TO GO HERE!:**

NavMesh::Node\* prev = nullptr;

for (auto it = path.begin(); it != path.end(); ++it) {

if (it != path.begin()) {

NavMesh::Node\* node = (NavMesh::Node\*)\*it;

// MORE TO GO HERE

prev = (NavMesh::Node\*)\*it;

}

}

And now we need to determine which way the edge is aligned, so that we can ensure the funnel is aligned correctly.

To do this we can create a vector that goes from the previous node’s position to the current node, then another vector that goes from the previous node’s position to the first of the shared vertices between the triangles. Both vectors must be normalised.

With these two vectors we can perform a cross product. A cross product is typically a calculation performed on two 3D vectors, but in this case we can assume our 2D vectors are 3D and both have a Z component of 0. A cross product between these two vectors will return a new vector that only has a valid Z component, which we can use to determine which way the triangle is facing based on if it is positive or negative, and thus determine if the edge is clockwise or counter-clockwise.

The cross product between the two 2D vectors is calculated as so:

So, using the cross product of the two vectors, we can add portals to the list with the correct order. The following code replaces the previous snippet’s **// MORE TO GO HERE**:

// find vertices they share to make a portal from

Vector2 adjacent[2];

prev->getAdjacentVertices(node, adjacent);

// get a vector going from previous node to this one

float mag = (node->position.x - prev->position.x) \*

(node->position.x - prev->position.x) +

(node->position.y - prev->position.y) \*

(node->position.y - prev->position.y);

Vector2 fromPrev = {};

if (mag > 0) {

mag = sqrt(mag);

fromPrev.x = (node->position.x - prev->position.x) / mag;

fromPrev.y = (node->position.y - prev->position.y) / mag;

}

// now get a vector going to the first adjacent vertex on the edge

mag = (adjacent[0].x - prev->position.x) \*

(adjacent[0].x - prev->position.x) +

(adjacent[0].y - prev->position.y) \*

(adjacent[0].y - prev->position.y);

Vector2 toAdj0 = {};

if (mag > 0) {

mag = sqrt(mag);

toAdj0.x = (adjacent[0].x - prev->position.x) / mag;

toAdj0.y = (adjacent[0].y - prev->position.y) / mag;

}

if ((fromPrev.x \* toAdj0.y - toAdj0.x \* fromPrev.y) > 0) {

portals[index++] = adjacent[0];

portals[index++] = adjacent[1];

}

else {

portals[index++] = adjacent[1];

portals[index++] = adjacent[0];

}

And with that the **smoothPath()** method would be complete.

Implement a Behaviour to Follow the Path:

To properly make use of this technique you will need to implement a game object to follow a smooth path.

The provided application contains **Behaviour Tree** behaviours within the **NavMesh** class that you may wish to make use of, or you may implement your own, but you must now implement a behaviour for a game object to follow a path through the nav mesh.

It is suggested that the game object start at a random node and then path to a target random node, and once it arrives there it should path to a new random node.

Alternatively you may want to use the mouser to select target locations for the game object to navigate to.