Automatic Navigation Mesh Generation in Configuration Space

Stuart Golodetz

1 Introduction

The representation of the walkable area of a 3D environment in such a way as to facilitate successful navigation by intelligent agents is an important problem in the computer games and artificial intelligence fields, and it has been extensively studied. As surveyed by Tozour [15], there are a variety of common ways to represent such an environment, including:

- *Regular Grids.* These support random-access lookup but do not translate easily into a 3D context. They also use a lot of memory and can yield aesthetically unpleasing paths for navigating agents.
- *Waypoint Graphs.* These connect large numbers of nodes (often manually placed) using edges that imply walkability in the game world. They were previously popular in games but are costly to build and tend to constrain agents to walking 'on rails' between connected waypoints.
- *Navigation Meshes.* These represent the walkable surface of a world explicitly using a polygonal mesh. Polygons within a navigation mesh are connected using links that imply the ability of the agent to walk/step/jump/etc. between them (see Figure 1).

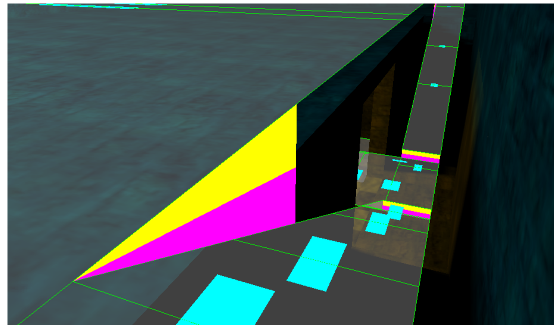
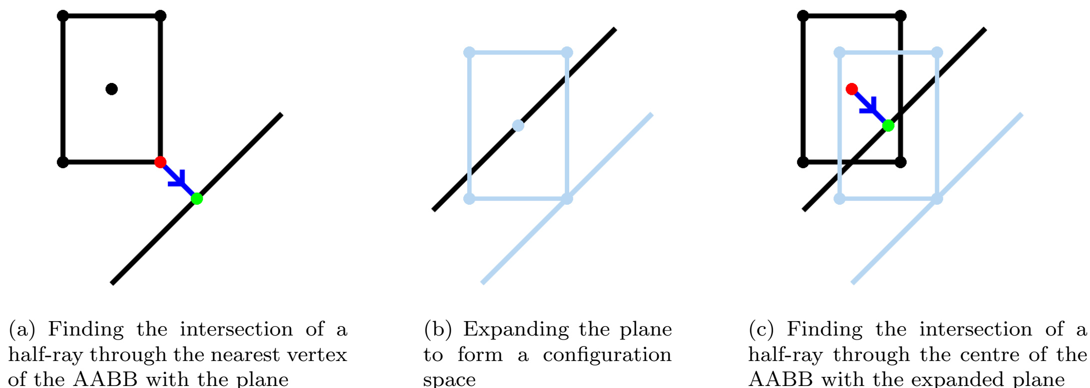


Figure 1: An example navigation mesh and its links: cyan = walk link; magenta = step up link; yellow = step down link.

Since their introduction by Greg Snook [13], navigation meshes have proved to be a particularly successful approach due to their ability to represent the free space available around paths through the world (this is extremely useful because it provides the pathfinder with the information it needs to successfully avoid local obstacles). As a result, they have seen widespread use in both games themselves, and popular games engines such as Source and Unreal, and many games authors have contributed to their theoretical development (most notably in the *Game Programming Gems* and *AI Game Programming Wisdom* book series). There has also been significant interest from researchers in academia (e.g. see [4, 7, 10, 16]).

One facet of using navigation meshes is how to build them in the first place, and numerous methods have been described in the literature. An early approach due to Tozour [14] works by first determining the walkable polygons in a 3D environment by comparing their normals with the up vector, and then iteratively merging together as many polygons as possible using the Hertel-Mehlhorn algorithm [6, 9] and a $3 \rightarrow 2$ merging technique. Hamm [5] generates a navigation mesh using an empirical method that involves sampling the environment to create a grid of points, identifying a subset of points both on the boundary of and within the environment, and connecting these points to form a mesh. Ratcliff [11] creates a navigation mesh by tessellating all walkable surfaces in the world, merging the results together to form suitable



(a) Finding the intersection of a half-ray through the nearest vertex of the AABB with the plane

(b) Expanding the plane to form a configuration space

(c) Finding the intersection of a half-ray through the centre of the AABB with the expanded plane

Figure 2: Detecting the first collision point between a translating AABB and a plane.

nodes and then computing links between neighbouring nodes. Van Toll *et al.* [16] build a navigation mesh for a multi-layer environment by constructing a mesh based on the medial axis for each layer and then connecting the medial axes by 'opening' the connections between the layers. The same authors also demonstrate how such a mesh can be dynamically modified [17]. Mononen's open-source Recast library [8] first voxelizes the 3D environment before running a watershed transform [1, 3] on the walkable voxels and creating a mesh from the resulting partition of the walkable surface.

In this article, I describe the implementation of navigation mesh construction in my homemade *hesperus* engine [2], based heavily on the techniques of van Waveren in [18]. The goal is to provide a helpful, implementation-focused introduction for those with no prior experience in the area. At a high-level, the method is as follows:

1. Firstly, given a 3D environment made up of brushes (simple convex polyhedra, each consisting of a set of polygonal faces), and a set of axis-aligned bounding boxes (AABBs) used to represent the possible sizes of the agents that will navigate the environment, expand the brushes by appropriate amounts (see §2) to create a set of expanded brushes for each AABB.

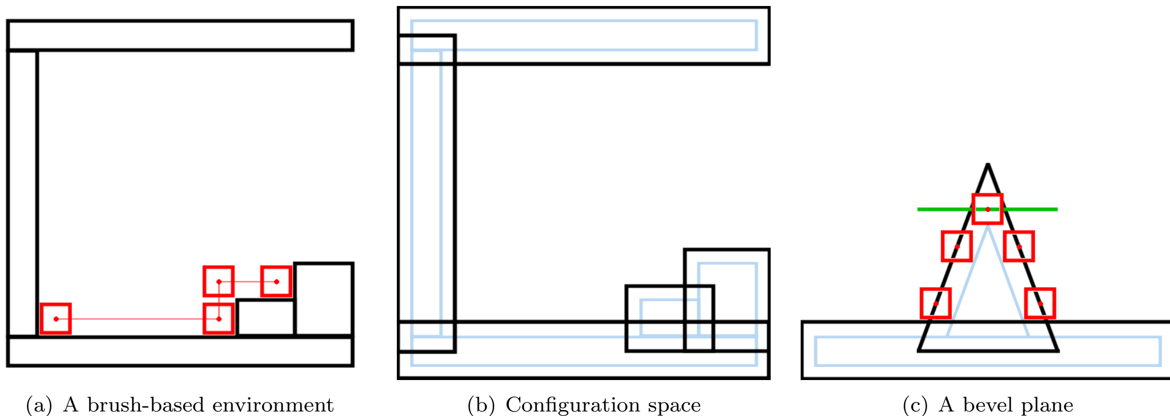
2. Next, using constructive solid geometry (CSG) techniques, union the expanded brushes for each AABB together to form a polygonal environment. Filter the polygons of the environment to find those that are *walkable* (judged by comparing their face normals with the up vector). This gives us the polygons of a navigation mesh for each AABB, but without any links to indicate how agents should navigate between them. See §3.

3. Finally, generate walk and step links between the polygons of each navigation mesh (see §4) – these indicate, respectively, that an agent can walk from one polygon of the mesh to another, or step up/down from one polygon to another. These links can be used to generate a graph for the purposes of path planning.

The following sections look at each of these steps in more detail.

2 Configuration Space

When planning the movement of intelligent agents (e.g. robots), a *configuration space* is the space of possible configurations in which an agent can validly exist. As a first example of what this concept means and when it can be useful, consider detecting the first point of collision between a plane and an AABB



(a) A brush-based environment (b) Configuration space (c) A bevel plane

Figure 3: A configuration space can be generated for the entirety of a brush-based environment by expanding all of the brushes by an appropriate amount: (a) shows a brush-based environment, together with the range of movement of a simple agent; (b) shows the configuration space that would be generated for the centre of that agent; (c) shows that expanding non-axis-aligned brushes may require *bevel* planes (shown in green) in order to correctly determine the range of movement.

that is moving by translation only – this might normally involve determining which vertex of the AABB is nearest to the plane and finding the point at which a half-ray oriented in the AABB's direction of movement and starting at that vertex would intersect the plane (see Figure 2(a)). The configuration space alternative is to initially expand the plane in the direction of its normal by a fixed amount so that the centre of the AABB touches the expanded plane precisely when the nearest vertex of the AABB touches the non-expanded plane (see Figure 2(b)). The first point of intersection can then be calculated by finding the point at which a half-ray starting at the centre of the AABB would touch the expanded plane (see Figure 2(c)) – there is no longer a need to first determine which vertex is nearest to the plane. Put another way: by expanding the plane, we have created the space of possible configurations for the centre of the AABB, and thereby restricted our testing to making sure that the AABB's centre is always in a valid location.

As explained in [18], the concept of configuration space can be extended to an entire 3D environment, allowing us to test an agent represented as an AABB against such an environment using only point-based (rather than AABB-based) intersection tests: this was the approach taken in the popular *Quake III Arena* game. Starting with a *brush-based* 3D environment (i.e. one that is built up by combining instances of simple convex polyhedra such as cuboids, cylinders and cones – a common approach in 3D world editors), a configuration space for agents with a specific AABB can be constructed by expanding each brush by an appropriate amount (see Figures 3(a) and 3(b)). Note that expanding each brush correctly can require the introduction of additional *bevel* planes as described in [18] (see Figure 3(c)).

3 Basic Mesh Generation

3.1 Brush Unioning

Having expanded the brushes of a brush-based environment to construct a configuration space for an AABB in the manner described, the next step is to union the expanded brushes together to generate a set of polygons that represent the expanded environment as a whole. These polygons can then be pro-

Listing 1 Brush Unioning

```
function union_all
:   (brushes: Vector<Brush>) → List<Polygon>

var result: List<Polygon>;

// Build a tree for each brush.
var trees: Vector<BSPTree> :=
  map(build_tree, brushes);

// Determine which brushes can interact.
var brushesInteract: Vector<Vector<bool>>;
for each b_i, b_j ∈ brushes
  if j == i then
    brushesInteract(i, j) := false;
  else
    brushesInteract(i, j) := in_range(b_i, b_j);

// Clip each polygon to the tree of each brush
// within range of its own brush.
for each b_i ∈ brushes
  for each f ∈ faces(b_i)
    var fs: List<Polygon> := [f];
    for each b_j ∈ brushes
      if brushesInteract(i, j) then
        fs := clip_polygons(fs, trees(j), i < j);
    result.splice(result.end(), fs);

return result;
```

cessed further to construct a navigation mesh.

In conceptual terms, the process of brush unioning is relatively simple: given an input set of brushes, each of which consists of a set of (outward-facing) polygonal faces, it suffices to clip each brush face to all the other brushes within range of its own brush in the environment. From an implementation perspective, a convenient way to do this is to build a binary space partitioning (BSP) tree for each brush and clip each face to the trees of the other brushes. The high-level implementation of this process can be found in Listing 1 and full source code is available online [2]. The end result is a set of polygons that represent the expanded environment.

3.2 Finding Walkable Polygons

Given the set of polygons generated in the previous section, finding those polygons that are *walkable* is straightforward: it suffices to compare the angle between each polygon's normal, $\hat{\mathbf{n}}$, and the up vector ($\hat{\mathbf{u}} = (0,0,1)^T$) to some predefined threshold. The

Listing 2 Edge Plane Table Construction

```
function build_edge_plane_table
:   (walkablePolygons: List<Polygon>) →
    Map<Plane,EdgeRefsPair,PlaneOrdering>

var ept: Map<Plane,EdgeRefsPair,PlaneOrdering>;

for each poly ∈ walkablePolygons
  for each p_1 ∈ vertices(poly)
    var p_2: Vec3 := next_vertex(poly,p_1);
    var plane: Plane := make_vertical_plane(p_1,p_2);
    var canon: Plane := plane.make_canonical();
    var sameFacing: bool :=
      plane.normal().dot(canon.normal()) > 0;
    if sameFacing then
      ept(canon).sameFacing.add(EdgeRef(poly,p_1));
    else
      ept(canon).oppFacing.add(EdgeRef(poly,p_1));

return ept;
```

angle can be computed using the dot product:

$$\theta = \cos^{-1}(\hat{\mathbf{n}} \cdot \hat{\mathbf{u}})$$

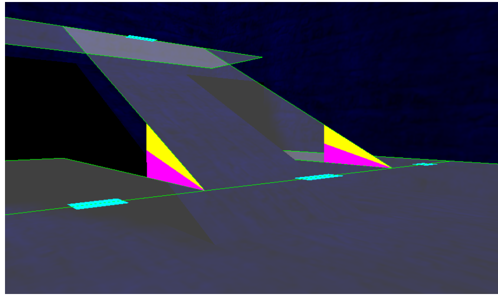
We then keep precisely those polygons whose angle is less than or equal to the threshold. In the *hesperus* engine, a suitable threshold for human characters was found to be $\pi/4$ (i.e. 45 degrees to the horizontal).

4 Walk and Step Links

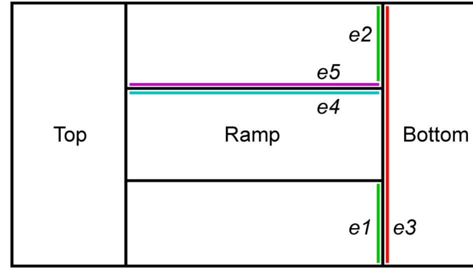
To generate simple links between walkable polygons in a navigation mesh, the general strategy is as follows:

1. We first create an *edge plane table* that maps each vertical plane through one or more walkable polygon edges to two sets of edges that lie in the plane (edges in one set are oriented in the same direction as a 'canonical' plane; edges in the other have the opposite orientation).

2. For each plane in the table and for each ordered pair of opposing edges for that plane, we check to see whether any links need to be created. This is done by transforming the opposing edges into a 2D coordinate system in the plane and calculating the intervals (if any) in which the various types of link need to be created.



(a) The navigation mesh for *Ramp*



(b) A top-down view, with some edges highlighted

Plane	**Same-Facing Edges**	**Opposite-Facing Edges**
1	$\{e_1, e_2\}$	$\{e_3\}$
2	$\{e_4\}$	$\{e_5\}$

(c) The part of the edge plane table corresponding to the highlighted edges

Figure 4: An illustration of (part of) the edge plane table for the *hesperus* test level called *Ramp*: the ordered pairs of opposing edges are (e_1, e_3), (e_2, e_3), (e_3, e_1), (e_3, e_2), (e_4, e_5) and (e_5, e_4). The first four pairs will cause walk links to be created; the last two pairs will cause step links to be created.

4.1 Edge Plane Table Construction

The edge plane table is a map of type Plane $\rightarrow$ ({Edge}, {Edge}). To construct it, we proceed as shown in Listing 2. For each edge of a walkable polygon, we first build the vertical plane passing through it and then add it to the table based on its facing with regard to the 'canonical' vertical plane through the edge. This has the effect of separating the edges of walkable polygons into two sets on each vertical plane. These can then be checked against each other in a pairwise manner to create navigation links – see Figure 4 for an example. A few details are needed to make this work:

1. *Vertical Plane Construction.* Each edge is necessarily non-vertical (because it belongs to a walkable polygon), so the normal vector of a vertical plane through it can be calculated straightforwardly using the cross-product. If the endpoints of the edge are $\mathbf{p_1}$ and $\mathbf{p_2}$, and $\hat{\mathbf{u}}$ is once again the up vector, then the desired normal can be calculated as:

$$\mathbf{n} = (\mathbf{p_2} - \mathbf{p_1}) \times \hat{\mathbf{u}}$$

   Normalising this to give $\hat{\mathbf{n}} = \mathbf{n}/|\mathbf{n}|$, the equation of the desired plane is then:

$$\hat{\mathbf{n}} \cdot \mathbf{x} = \hat{\mathbf{n}} \cdot \mathbf{p_1}$$

2. *Canonical Plane Determination.* Given a plane with equation $ax + by + cz - d = 0$, we define the 'canonical' form of this plane to be the one where the first of a, b or c to be non-zero is positive. Thus, the canonical form of both $0x + 1y - 1z - 23 = 0$ and $0x - 1y + 1z + 23 = 0$ would be $0x + 1y - 1z - 23 = 0$. Note that a plane is either already in canonical form, or can be canonicalised straightforwardly by negating all of its coefficients.

3. *Plane Ordering.* In order to use planes as the key for the edge plane table, we need to define a suitable way of ordering them. This can be done using a variant of the approach described in [12]. In practice, the ordering was found to be easier to debug (although somewhat less efficient) if implemented as shown in Listing 3.

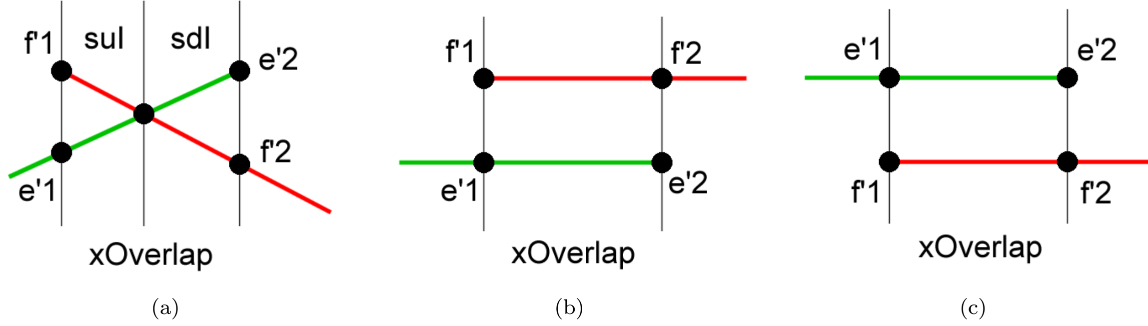


Figure 5: Creating links between edges: in (a), the gradients differ and a step down link is created from e to f in the region labelled sdI, and a step up link is created in the region labelled suI; in (b), the gradients are the same and a step up link is created from e to f in the region labelled xOverlap; in (c), a step down link is created from e to f in the region labelled xOverlap. The remaining case, of parallel edges leading to a walk link, is not shown.

4.2 Link Creation

To generate walk and step links, we transform each ordered pair[1] of opposing edges that lie in the same (canonical) plane into a 2D (orthonormal) coordinate system in the plane, and then determine the intervals (if any) in which links need to be created. A suitable 2D coordinate system for a plane $\hat{\mathbf{n}} \cdot \mathbf{x} - d = 0$ can be generated as follows. To determine a suitable origin $\mathbf{o}'$ for our coordinate system, we find the point on the plane that lies nearest to the world origin $\mathbf{0}$: this is simply $d\hat{\mathbf{n}}$. Given that the plane is vertical, suitable axis vectors for our coordinate system can be defined as:

$$\hat{\mathbf{i}}' = \frac{\hat{\mathbf{n}} \times \hat{\mathbf{u}}}{|\hat{\mathbf{n}} \times \hat{\mathbf{u}}|}$$

$$\hat{\mathbf{j}}' = \hat{\mathbf{u}}$$

This gives us a coordinate system in which $\hat{\mathbf{i}}'$ is horizontal (in terms of the surrounding world) and $\hat{\mathbf{j}}'$ is vertical. To transform an edge e on the plane into this new coordinate system, we transform each of its endpoints $\mathbf{e_1}$ and $\mathbf{e_2}$ as follows:

$$\mathbf{e_n} \mapsto ((\mathbf{e_n} - \mathbf{o}') \cdot \hat{\mathbf{i}}', (\mathbf{e_n} - \mathbf{o}') \cdot \hat{\mathbf{j}}') = \mathbf{e}'_{\mathbf{n}}$$

Having transformed a pair of opposing edges e and f into the plane's coordinate system, we next calculate the horizontal interval in which each edge lies; e.g. for e this would be:

$$[\min\{e'_{1x}, e'_{2x}\}, \max\{e'_{1x}, e'_{2x}\}]$$

If the horizontal intervals for the opposing edges do not overlap, then there can be no links between them. Otherwise, we compute the gradients m and cut points c of the lines $y = mx + c$ through the two edges (still in the plane's coordinate system) using basic mathematics; e.g. for e these would be:

$$m_e = \frac{e'_{2y} - e'_{1y}}{e'_{2x} - e'_{1x}}$$

$$c_e = e'_{1y} - m_e * e'_{1x}$$

Based on a comparison of the gradients, we then create the link segments in the plane as shown in Listing 4 and Figure 5. The endpoints $\boldsymbol{\ell_1}$ and $\boldsymbol{\ell_2}$ of each link segment can be straightforwardly transformed back into world space to create the actual links as follows:

$$\boldsymbol{\ell_n} \mapsto \mathbf{o}' + \ell_{nx} * \hat{\mathbf{i}}' + \ell_{ny} * \hat{\mathbf{j}}'$$

[1]Note that because the pairs are ordered, we consider each unordered pair of edges twice when creating links, once in each direction.

Listing 3 Plane Ordering

```
function less: (lhs: Plane; rhs: Plane) → bool

var nL, nR: Vec3 := lhs.normal(), rhs.normal();
var dL, dR: double := lhs.dist(), rhs.dist();

// If these planes are nearly the same (in terms
// of normal direction and distance value), then
// neither plane is "less" than the other.
var dotProd: double := nL.dot(nR);

// cos^-1(x) is only defined for x ≤ 1, so clamp
// dotProd to avoid floating-point problems.
if dotProd > 1.0 then dotProd := 1.0;

var angle: double := cos^-1(dotProd);
var dist: double := dL - dR;
if |angle| < ε_angle and |dist| < ε_dist then
  return false;

var aL, bL, cL: double := nL.x, nL.y, nL.z;
var aR, bR, cR: double := nR.x, nR.y, nR.z;

// Otherwise, compare the two planes
// "lexicographically".
return (aL<aR) or
       (aL=aR and bL<bR) or
       (aL=aR and bL=bR and cL<cR) or
       (aL=aR and bL=bR and cL=cR and dL<dR);
```

5 Potential Extensions

At present, only walk and step links are implemented in *hesperus*, but there are various additional links that it would be helpful to add.

5.1 Crouch Links

One obvious extension is to add *crouch* links – these are links that tell agents when they need to crouch in order to traverse low areas (e.g. a low archway or a pipe). As the example in Figure 6 illustrates, these are inter-mesh links; they should be created so as to link the standing and crouching meshes for an agent at the boundary of areas that can be traversed on the crouching mesh but not on the standing one. Assuming that the AABBs for the two meshes differ only in their heights (as is the case in the example), one way of automatically detecting crouch links would be to match edges on the standing mesh that do not cause any walk or step links to be created with edges in the same plane on the crouching mesh that do.

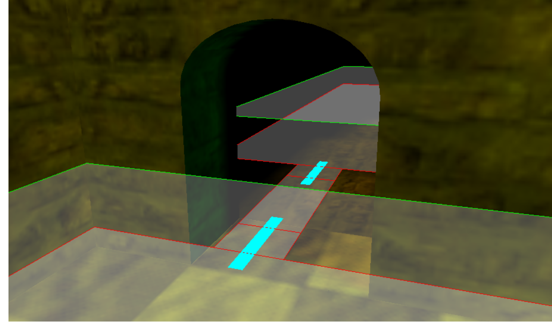


Figure 6: Creating crouch links between navigation meshes can allow tall characters to pass through low areas. Here, crouch links should be created between the standing (green) and crouching (red) meshes to allow agents to traverse this low archway.

5.2 Ladder Links

The addition of ladder links (indicating that an agent can travel from one floor to another using a ladder) is not conceptually very difficult, but it requires tool support. In *hesperus*, the map editor would need to be augmented to handle ladders and other static entities; when placing a ladder, it would then be a simple matter to create a link at either end of the ladder to represent travel in each direction. It should be noted that *traversing* ladder links is significantly more complicated than traversing walk or step links, because it takes time to climb or descend a ladder and someone may be coming the other way, but traversal is beyond the scope of this article.

5.3 Jump Links

A third type of extremely useful link would be jump links – these are used to indicate places at which an agent can jump to reach another part of the navigation mesh. Calculating jump links can be somewhat costly because it involves simulating the agent making jumps to determine whether or not they are possible. In our case, the situation is made slightly easier because we are working in configuration space and can avoid worrying about clearance, but general-purpose jump links are still non-trivial to generate

automatically. One easy type of jump link that could be generated immediately would be vertical jumps – these can be generated in the same way as step up links, but using a larger height threshold.

6 Conclusions

In this article, I have illustrated how to generate navigation meshes at an implementation level using an approach based on the work of van Waveren in [18]. Whilst there are many alternative techniques for navigation mesh construction, as surveyed in the introduction, this configuration space approach is useful because it allows us to avoid the difficulties regarding clearance height that have to be dealt with by other approaches; it also means that each agent occupies a single point on the mesh, completely avoiding the problems caused by an agent straddling multiple mesh polygons.

Navigation mesh *generation*, however, is only part of the picture – in a future article, I hope to write more about using navigation meshes for localisation, movement and path planning.

7 Acknowledgements

As always, I would like to thank the Overload team for reviewing this article and suggesting ways in which to improve it.

Listing 4 Link Segment Calculation

```
function calculate_link_segments
:   (e'1: Vec2; e'2: Vec2; f'1: Vec2; f'2: Vec2;
     xOverlap: Interval) → LinkSegments

xO ≡ xOverlap;
var result: LinkSegments;
var m_e: double := (e'_2y − e'_1y) / (e'_2x − e'_1x);
var m_f: double := (f'_2y − f'_1y) / (f'_2x − f'_1x);
var c_e: double := e'_1y − m_e * e'_1x;
var c_f: double := f'_1y − m_f * f'_1x;
var Δm, Δc: double := m_f − m_e, c_f − c_e;

if |Δm| > ε then
  // If the gradients of the source and destination
  // edges are different, then we get a combination
  // of step up/step down links. We want to find:
  // (a) The point walkX where y_f = y_e
  // (b) The point stepUpX where y_f − y_e = STEPTOL
  // (c) The point stepDownX where y_e − y_f = STEPTOL
  var walkX: double := −Δc/Δm;
  var stepUpX: double := (STEPTOL − Δc)/Δm;
  var stepDownX: double := (−STEPTOL − Δc)/Δm;

  // Construct the step down and step up intervals
  // and clip them to the known x overlap interval.
  var sdI, suI: Interval
    := [min{walkX, stepDownX}, max{walkX, stepDownX}],
       [min{walkX, stepUpX}, max{walkX, stepUpX}];
  sdI := sdI ∩ xO;
  suI := suI ∩ xO;

  // Construct the link segments.
  if not sdI.empty then
    result.downToF :=
      [(sdI.low, m_e * sdI.low + c_e) ,
       (sdI.high, m_e * sdI.high + c_e)];
    result.upToE :=
      [(sdI.low, m_f * sdI.low + c_f) ,
       (sdI.high, m_f * sdI.high + c_f)];
  if not suI.empty then <analogously>
else if |Δc| < STEPTOL then
  // If the gradients of the source and destination
  // edges are the same (i.e. parallel edges), then
  // we either get a step up/step down combination,
  // or a walk link in either direction.
  var s1: Vec2 := (xO.low, m_e * xO.low + c_e);
  var s2: Vec2 := (xO.high, m_e * xO.high + c_e);
  var d1: Vec2 := (xO.low, m_f * xO.low + c_f);
  var d2: Vec2 := (xO.high, m_f * xO.high + c_f);

  if Δc > ε then
    // The destination is higher than the source.
    result.upToF := [s1, s2];
    result.downToE := [d1, d2];
  else if Δc < −ε then
    // The destination is lower than the source.
    result.downToF := [s1, s2];
    result.upToE := [d1, d2];
  else
    // The destination and source are level.
    result.walk := [s1, s2];
```

References

[1] Serge Beucher. *Segmentation d'Images et Morphologie Mathématique (Image Segmentation and Mathematical Morphology)*. PhD thesis, E.N.S. des Mines de Paris, 1990.

[2] Stuart Golodetz. The *hesperus* 3D game engine. Source code available online at: `https://github.com/sgolodetz/hesperus2`.

[3] Rafael C Gonzalez and Richard E Woods. *Digital Image Processing*. Pearson Education, 2nd edition, 2002.

[4] D Hunter Hale and G Michael Youngblood. Full 3D Spatial Decomposition for the Generation of Navigation Meshes. In *Proceedings of the Fifth Artificial Intelligence for Interactive Digital Entertainment Conference*, pages 142–147, 2009.

[5] David Hamm. Navigation Mesh Generation: An Empirical Approach. In Steve Rabin, editor, *AI Game Programming Wisdom 4*, pages 113–123. Charles River Media, 2008.

[6] S Hertel and K Mehlhorn. Fast triangulation of simple polygons. In *Proceedings of the 4th International Conference on the Foundations of Computation Theory*, volume 158 of *Lecture Notes in Computer Science*, pages 207–218. Springer-Verlag Berlin, 1983.

[7] Marcelo Kallmann. Navigation Queries from Triangular Meshes. In *Proceedings of the Third International Conference on Motion in Games (MIG)*, pages 230–241. Springer-Verlag Berlin, 2010.

[8] Mikko Mononen. Navigation Mesh Generation via Voxelization and Watershed Partitioning. *AiGameDev.com*, March 2009. Slides available online (as of 30th July 2013) at `https://sites.google.com/site/recastnavigation/MikkoMononen_RecastSlides.pdf`.

[9] Joseph O'Rourke. *Computational Geometry in C*, pages 60–61. Cambridge University Press, 2nd edition, 1994.

[10] Julien Pettré, Jean-Paul Laumond, and Daniel Thalmann. A navigation graph for real-time crowd animation on multilayered and uneven terrain. In *Proceedings of the 1st International Workshop on Crowd Simulation*, Lausanne, Switzerland, 2005.

[11] John W Ratcliff. Automatic Path Node Generation for Arbitrary 3D Environments. In Steve Rabin, editor, *AI Game Programming Wisdom 4*, pages 159–172. Charles River Media, 2008.

[12] David Salesin and Filippo Tampieri. Grouping Nearly Coplanar Polygons into Coplanar Sets. In David Kirk, editor, *Graphics Gems III*, pages 225–230. Morgan Kaufmann, 1992.

[13] Greg Snook. Simplified 3D Movement and Pathfinding Using Navigation Meshes. In Mark DeLoura, editor, *Game Programming Gems*, pages 288–304. Charles River Media, 2000.

[14] Paul Tozour. Building a Near-Optimal Navigation Mesh. In Steve Rabin, editor, *AI Game Programming Wisdom*, pages 171–185. Charles River Media, 2002.

[15] Paul Tozour. Search Space Representations. In Steve Rabin, editor, *AI Game Programming Wisdom 2*, pages 85–102. Charles River Media, 2004.

[16] Wouter G van Toll, Atlas F Cook IV, and Roland Geraerts. Navigation Meshes for Realistic Multi-Layered Environments. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3526–3532, San Francisco, California, USA, 2011.

[17] Wouter G van Toll, Atlas F Cook IV, and Roland Geraerts. A navigation mesh for dynamic environments. *Computer Animation and Virtual Worlds*, 23:535–546, 2012.

[18] Jean Paul van Waveren. The Quake III Arena Bot. Master's thesis, Delft University of Technology, 2001.