0 A.D. Pathfinder Design

Wildfire Games – http://wildfiregames.com/

June 27, 2015

Disclaimer: This document mixes elements of the current implementation and TBI features. You're invited to take a look yourself at the code, or ask questions in #0ad-dev on QuakeNet.

Contents

1 Basic concepts

Our world consists of:

- **Units** – Small, mobile objects. Units move along the ground (or water, if they're ships). They have instant acceleration and deceleration, and zero turning circles.
- **Structures** – Possibly-large, stationary objects. This includes buildings, walls, trees, rocks, etc.
- **Terrain** – 2D grid with a heightmap and other per-terrain-tile data (e.g. texture), plus a water plane.

Units and structures are not restricted to terrain tiles in any way – they can have high-precision positions and can face in any direction. Both are represented as *obstruction squares* – we simplify their shapes to a single square[1]. Units are typically humanoids and should move in at least roughly human-like ways (not e.g. like hover-tanks). Units should never move through structures, and in general should never move through other units (except in special cases like formations).

The world might have something like 1000 units (all moving at once), 256×256 terrain tiles, 100 buildings, 1000 trees. The target platform is PCs (in particular x86 or x86_64 with 512+ MB RAM; we can use a lot of memory for pathfinding if necessary).

Pathfinding is the operation of picking a route for a unit to move along from its current location to any other location in the world, so that it does not collide with other units or with structures or with impassable terrain. The picked route should be the shortest (or an equal shortest) of all possible routes.

Pathfinding is split into two levels. The *long-range pathfinder* is responsible for finding approximate routes that can span from one corner of the map to the other. It only needs to care about structures and terrain; any units in the way will probably have moved by the time the pathing unit reaches them, so there's no need for the long-range pathfinder to plan around them. It can approximate the world as a 2D grid. The *short-range pathfinder* is responsible for computing precise paths that follow a segment of the long-range path – these should not be quantised to a grid (since that would look ugly), and should avoid other units, but only need to work over a short distance (to the next waypoint on the long-range path).

1.1 Coordinates

Units have an (x, z) position, measured in 'meters' (the generic world-space unit of distance). The coordinates are implemented as fixed-point numbers (with a 16-bit fraction, i.e. a resolution of 2^{-16}m $\approx 15\mu$m), but we can treat them as real numbers. The positive x direction is interpreted as right or east; the positive z direction is up or north.

Navcells (the tile-like concept used for navigation) are squares identified as (i, j), corresponding to world-space coordinates

$$i \times \text{NavcellSize} \leq x < (i+1) \times \text{NavcellSize}$$
$$j \times \text{NavcellSize} \leq z < (j+1) \times \text{NavcellSize}$$

where NavcellSize is typically 1.0 in the current implementation.

A unit is always located on a single navcell:

$$\text{PointToNavcell}(x, z) = (\lfloor x/\text{NavcellSize} \rfloor, \lfloor z/\text{NavcellSize} \rfloor)$$

[1] For "square", read "rectangle". Or "affine transformation of a square" if you prefer.

1.2 Units

Units have a *unit obstruction* shape, with a defined radius. The shape is usually interpreted as an axis-aligned square (with width and height equal to 2 × UnitRadius), but sometimes interpreted more like a circle. Infantry units might have a radius of around 0.5, while siege units might have a radius of around 4.

Units also have a passability class, which defines various details about what kinds of terrain they can cross. Most of the data structures used by the pathfinder are duplicated once per passability class. For this document, we'll mostly just describe algorithms in terms of a single passability class, and the implementation implicitly picks the appropriate data for the current unit's class.

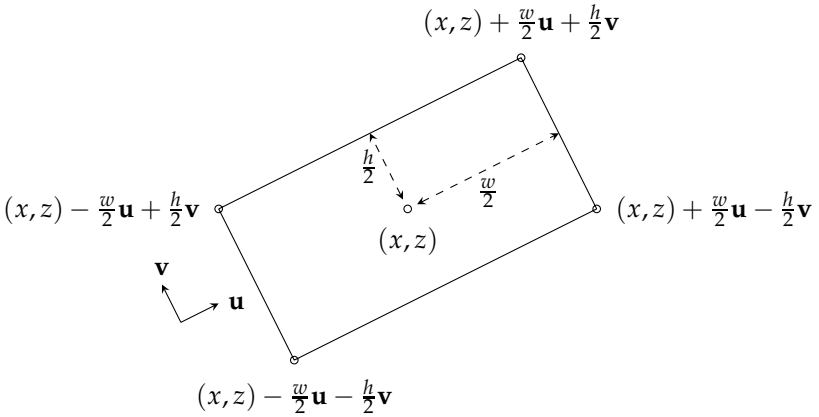
There might be something like 8–16 distinct passability classes in the game.

Passability classes also define a *clearance* value, which is typically similar to their obstruction radius, defining how close they can get to structures or impassable terrain. Clearance should be a multiple of NavcellSize.

TODO - Constraints on clearance vs radius: Currently, the radius of units is ignored, only their clearance is taken into account when computing a path, and when the move actually happens, their real obstruction size is not tested.

1.3 Structures

Structures are represented by *static obstruction squares*, which are non-axis-aligned parallelograms defined by $(x, z, \frac{w}{2}, \frac{h}{2}, \mathbf{u}, \mathbf{v})$:



The $\mathbf{u}$ and $\mathbf{v}$ are unit vectors. They should be perpendicular; that's not theoretically required, but some of the implementation assumes that.

We actually store the half-width $\frac{w}{2}$ (as "`hw`" in the code) instead of w, because that seems more convenient for the implementation. (Same for h.)

We store $\mathbf{u}$ and $\frac{w}{2}$ as two values, instead of pre-multiplying them and using a single (non-unit) vector everywhere, because occasionally we really need the unit vectors (e.g. to compute world-space distance from a point to a square).

Some of the game code uses a representation (x, z, w, h, θ) instead (where θ is anticlockwise from the x axis), which is more convenient for editing. That is transformed into the vector form (which is more convenient for geometric computation) for the purposes of the algorithms described in this document.

2 Implementation and code organization

Pathfinding happens in two typical situations:

- When a unit is requested to move;

- When the AI wants to plan a move (which will be eventually processed by the unit pathfinder) or a building construction.

The first case is handled by two components: `CCmpPathfinder`, which is a system component implementing pathfinding algorithms, and `CCmpUnitMotion`, which is a component each unit entity has. `CCmpUnitMotion` calls `CCmpPathfinder` methods to compute unit moves.

The second case must be handled inside each *AI worker*, which is an object defined in `CCmpAIManager` that should eventually allow AI threading. As a consequence, it is not possible for AI workers to ask `CCmpPathfinder` for paths in a synchronous way.

Instead of implementing an asynchronous path request system for AI pathfinding, both AI workers and `CCmpPathfinder` have a `LongPathfinder` object capable of computing long paths based on a passability grid computed from terrain and static obstructions. This grid is computed in `CCmpPathfinder::UpdateGrid` and passed to the AI manager when an update is needed.

As of the short-range pathfinder, short-range algorithms remain plain `CCmpPathfinder` methods, only used for unit motion.

3 Navcell grid

The long-range pathfinder can be simpler and more efficient if it has a grid representation of the map. (Since our maps are large and open (i.e. a high degree of connectivity between areas), and must support dynamic generation (random map scripts) and dynamic modification (new buildings, walls, etc), a grid is likely better than a nav mesh.)

Terrain is already based on a grid representation of the map, but it is pretty low resolution (tiles have size 4.0×4.0; you could fit 16 units onto a single terrain tile without much trouble) so it will be poor at approximating narrow gaps which a single unit could fit through. Increasing the terrain resolution would mean more heightmap and texture-pointer data (10 bytes per tile), more graphics data (at least 16 bytes per tile, more when blending between textures), and more triangles to render per frame. It's better if we can use a higher-resolution grid for pathfinding than for terrain.

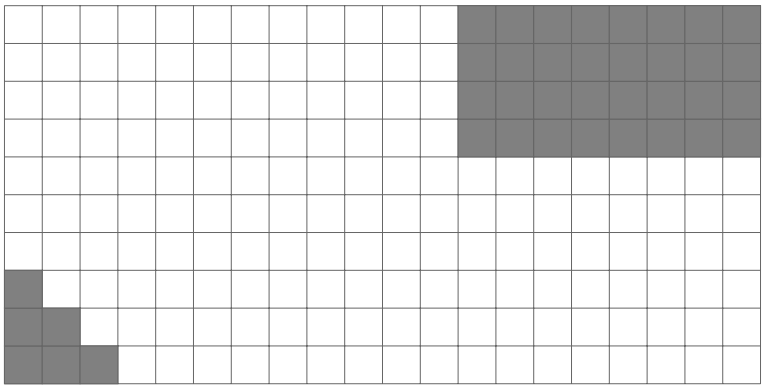
Pathfinding therefore uses a navcell grid, independent of the terrain grid. Navcell passability is encoded as a 2D array covering the map, with one bit per passability class. Passability is computed from terrain and from static obstructions.

3.1 Terrain

(Implemented by `CCmpPathfinder::UpdateGrid`.)

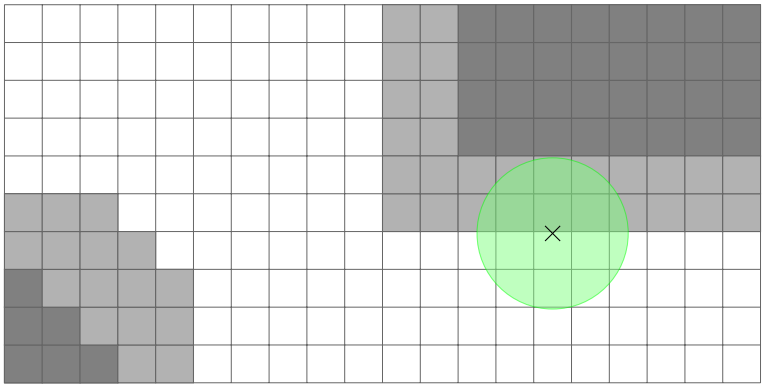
A passability class defines maximum terrain slope, min/max water depth, etc. For each navcell, we compute slopes and depths from the terrain heightmap. When navcells are higher resolution than the heightmap, we do some interpolation to get smoother output.

That will give a binary *terrain passability grid* for this passability class:



We also force the navcells around the outside of the map to be impassable. (This means we usually don't have to worry about units walking off the edge of the map. It also means we can use the shroud-of-darkness effect to make the map smoothly fade out around its edges, and units won't walk so far out that they disappear into that SoD.) This is either a circular or square pattern, and we currently have at least 3 impassable terrain tiles (i.e. currently 12 navcells) around each edge.

The grid is then expanded by the passability class's clearance value c (effectively convolution with a square filter of size $(2c+1) \times (2c+1)$) to get the *expanded terrain passability grid*:



(It'd be nicer to do a more circular convolution, so that the sharp corner in the top-right gets smoothed to a more rounded corner, especially with large clearance values. That would be a self-contained enhancement and won't have any further consequences.)

A unit may be located anywhere on any passable navcell in this new grid. If a unit is visualised as a circle with radius r, then the circle won't overlap any impassable navcell in the original terrain passability grid as long as $c \geq r$. The green circle in the previous figure indicates a unit with $r = 2$ in a valid location.

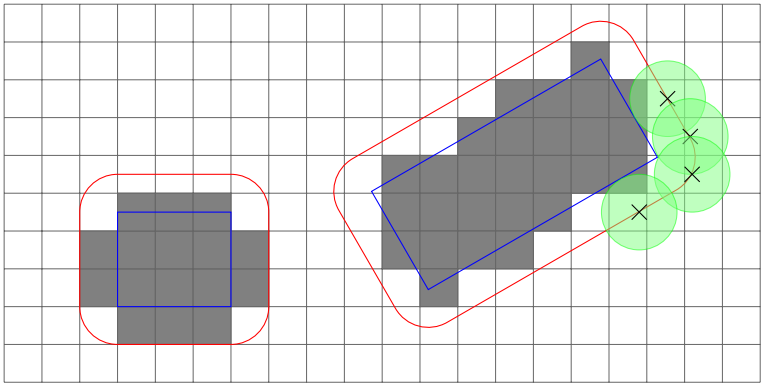
3.2 Static obstructions

(Implemented by `CCmpObstructionManager::Rasterize`.)

Rasterization is the process of converting a vector shape (in this case an obstruction square) into a pixel grid (in this case actually a navcell grid). We do this so that the grid-based pathfinders can handle static obstructions (like walls) in the same way as they handle terrain-based obstructions (like rivers).

For a given clearance value c, a navcell is blocked by an obstruction shape if every corner of the navcell is either inside the shape or is a distance $d \leq c$ away from the shape.

In the following figure, blue is obstruction squares, red is the points at distance $c = 1$ outside of the obstruction square, green circles are some units with radius 1, and grey squares are blocked by the obstructions.



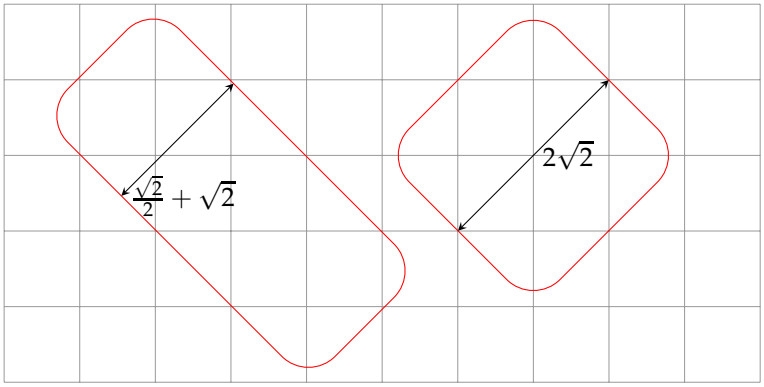
It is important that partially-obstructed navcells are still considered passable: when a unit is trying to move to a building (e.g. to attack it), it can always succesfully get within any distance $d > c$ of the obstruction shape, without having to move illegally onto an impassable navcell.

TODO: In the implementation, we need to be sure units never try to move within a distance $d \leq c$, else they might be blocked by navcells. Maybe add c to all (positive) attack/gather/etc ranges?

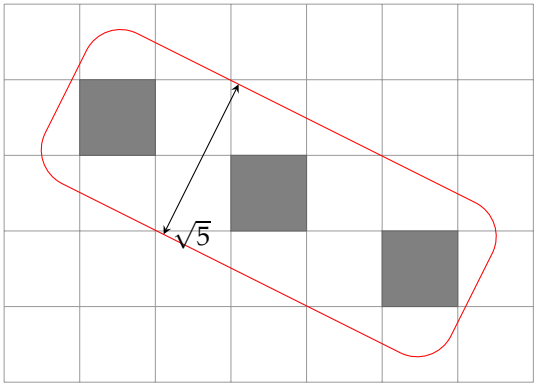
3.3 Narrow obstructions

If an obstruction is especially narrow (e.g. a fence), it might not block any navcells and units could walk straight through it, which would surprise map designers.

The worst case is when rotated 45 degrees, in which a long obstruction must be at least $\frac{3}{\sqrt{2}}$ (about 2.12) navcells wide (including the clearance expansion), and a square obstruction must be at least $2\sqrt{2}$ (about 2.83) wide:



But there are still problems when a long obstruction is $\sqrt{5}$ (about 2.24) wide, since it may contain gaps that units can walk through:



To avoid these problems, when rasterizing a rectangle of size (w, h) with clearance c, we will instead rasterize a rectangle of size (w, h) with clearance $\max(c,\ (3 - \min(w, h))/2)$ (where 3 is a nice round number that's definitely bigger than the limits demonstrated here). When the smallest unit has $c = 1$, that limit will only take effect for obstructions with $w < 1$ or $h < 1$.

3.4 Grid updates

Each turn, the simulation calls CCmpPathfinder's UpdateGrid. This function calls the obstruction manager to know which part of the grid should be updated. This data is then stored for a clever update of the long-range pathfinder internal state. It is also sent to the AI long pathfinder.

If the terrain is modified, everything is recomputed. Else, the zone of the map where modified obstructions are is reset to its terrain-only passability value, and all obstructions overlapping modified obstructions are rasterized again on the map.

4 Path goals

When a path is requested, there is a source unit and a goal. The goal might be:

- Point (x, z) – e.g. when telling a unit to walk to some specific location.

- Square $(x, z, \frac{w}{2}, \frac{h}{2}, \mathbf{u}, \mathbf{v})$ – e.g. when telling a melee unit to attack a square building, they will move to positions close to the building (depending on their maximum attack range) in a square pattern.
- Circle (x, z, r) – e.g. when telling a ranged unit to attack a square building, they will move to positions within maximum attack range of the building in a circular pattern.
- TODO: Inverted squares/circles for minimum attack range?

TODO: Need some constraints on squares/circles that are based on structures, so they're always far enough out.

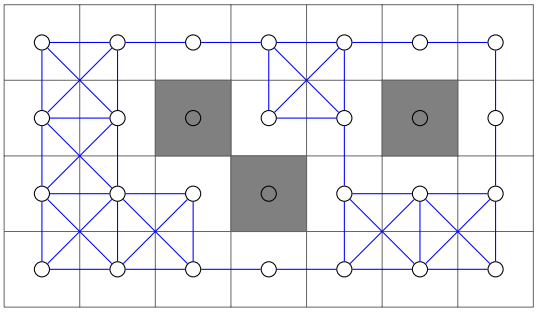
A navcell can be considered to be *inside a goal*. If the goal is a point (x, z), then only the navcell PointToNavcell(x, z) is inside the goal. Otherwise, a navcell is inside the goal if any point inside (or on the edge of) that navcell is inside (or on the edge of) the goal shape.

TODO: Think about edges cases in relation to vertex pathfinder.

5 Long-range pathfinder

5.1 Navcell-based A* pathfinder with JPS optimization

The basic pathfinder is a pretty standard A* over a uniform-cost grid of navcells. Connectivity between navcells is as illustrated:



From any passable navcell, you can move to any horizontally/vertically adjacent passable navcell with cost 1. Also, you can move diagonally from any navcell (i, j) to $(i \pm 1, j \pm 1)$ if those navcells, plus the adjacent navcells $(i \pm 1, j \mp 1)$ and $(i \mp 1, j \pm 1)$, are all passable. Diagonal moves have cost $\sqrt{2}$.

Note that this definition of diagonal movement means that connectivity between any two navcells can be determined solely from the horizontal/vertical adjacencies; the only effect of the diagonal moves is to allow smoother shorter paths.

This algorithm is opimized using a JPS algorithm (jump-point search), which is especially efficient on sparse grids, i.e. when a lot of similar possible paths exist.

5.2 Hierarchical pathfinder

The long pathfinder has a hierarchical pathfinder that provide efficient connectivity algorithms. It is used directly by the AI and it is called to improve long paths computed with the A* algorithms.

It can do the following:

- Determine if there is at least one path through the navcell grid between passable navcells a and b (i.e. determine if b is *reachable* from a).
- Given a goal shape and a passable source navcell a, determine if there is any navcell that is in the goal and is reachable from a. If not, find one of the reachable navcells that is nearest to the center of the goal.
- Given an impassable navcell a, find one of the passable navcells that is nearest to a.

Note that it doesn't actually find paths. We might want to extend it to do that in the future, if JPS is too slow.

TODO: Details.

6 Vector pathfinder

In addition to the navcell-based pathfinders (which are fine for approximate paths over a fairly static grid with large obstructions, but not good at fine detail or at very frequent changing or small obstructions), we have a pathfinder based on a vector representation of the world.

The current implementation is a points-of-visibility pathfinder. The world is represented as a set of edges (impassable straight lines for the outlines of obstructions) and a set of vertexes (for the corners of obstructions). Edges are single-sided (they block movement from outside an obstruction to the inside, but not vice versa). A path can go from any vertex to any other vertex, if the straight line between those vertexes does not intersect any obstruction edges. Given those vertexes and connectivity data, a basic implementation of A* can find optimal paths.

The vector pathfinder has to understand terrain-based impassability (which is fundamentally defined on a navcell grid), static obstructions, and dynamic obstructions. (Dynamic obstructions are other units – they are smallish, and can be treated as circles or axis-aligned squares or similar.)

Inconsistencies between the navcell pathfinder and vector pathfinder are problematic. If the navcell pathfinder thinks a route is blocked, but the vector pathfinder thinks there is enough space, then units will sometimes use that route and sometimes refuse to. In the opposite case, the navcell pathfinder will send a unit along a route that it thinks is valid, but the vector pathfinder will think there's not actually enough space and the unit will be stuck and won't know where to go. This is inevitable when a route is blocked by dynamic obstructions, but we should aim to avoid such problems in at least the cases where there are no dynamic obstructions.

To minimise those inconsistencies, the vector pathfinder uses the navcell-grid rasterized versions of static obstructions, instead of using the original rotated-rectangle vector versions.

The pathfinder algorithm does not explicitly take account of the unit's size – it assumes the unit is a point and can move arbitrarily close to an obstruction edge. Because the navcell grid has already encoded the unit's clearance value, we can convert the navcell outlines directly into the vector representation for this pathfinder, and units will not get too close to static obstructions.

Note that since navcells intersecting the edge of a static obstruction are not made impassable, the vector pathfinder will let units move slightly inside the geometric shape of the obstruction (by up to $\sqrt{2} \times$ NavcellSize). (TODO: Be more specific/correct.)

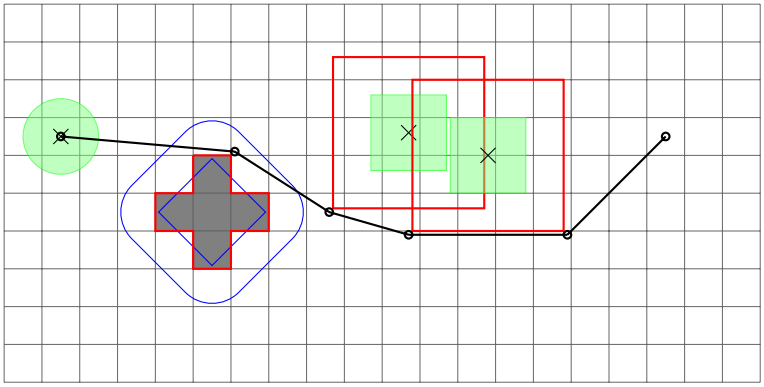
Dynamic obstructions (units etc) are converted directly into vectors, without quantising to the navcell grid. Their obstruction shapes are axis-aligned squares with a known radius. To stop units intersecting each other, each square is expanded by the pathing unit's own obstruction radius.

Note that all edges (from dynamic obstructions and from the navcell grid) are axis-aligned. This allows some simplifications and optimisations, and avoids precision issues when deciding whether a point is precisely on a line.

TODO: Inconsistency between clearance and obstruction radius?

The pathfinder can then find paths between corners that don't intersect edges.

TODO: How does it handle edge cases like units starting precisely on an edge?



7 Specific situations

7.1 Stuck units

(Implemented by `LongPathfinder::ComputePathOffImpassable`.)

A unit might find itself on an impassable navcell. (We should prevent that whenever possible, but we should be robust to error cases where the prevention fails, and it can easily happen in Atlas.) The JPS pathfinder and hierarchical goal reachability system inherently require that units start on passable navcells. We therefore need a way to safely get units off impassable navcells as quickly as possible.

The hierarchical pathfinder can find the nearest passable navcell to the unit. We can then construct a straight-line path to that navcell, and let the unit recompute a proper path once it's got out.

Normally the short-range pathfinder will let the unit get quite widely diverted from the long-range path. That's bad when the unit is starting inside a building or a region of impassable terrain – it might decide to go a very long way through the impassable region, instead of getting out as quickly as possible.

TODO: How can we handle that? (Add a flag to disable the short-range pathfinder? What if another unit is in the way?)

7.2 Minimum ranges

Some units have a minimum attack range (e.g. a ballista can't be aimed at someone standing right in front of it). If they are too close to their target, they need to move away from it before being able to attack.

Minimum ranges will usually be quite small – maybe 16 navcells at most. That's short enough that the short-range pathfinder will always suffice; we don't need to use the long-range pathfinder to work out how to move away from the target. This means the long-range pathfinder only has to care about how to move towards a target (which avoids some minor complexities in the implementation).

The short-range pathfinder already moves to the edge of a goal shape, and doesn't care whether the unit is starting inside or outside that shape. That means it'll work for minimum-range movement with no further modifications.

7.3 Unit movement

The long-range pathfinder will give the unit a waypoint to walk towards. The unit will move a short distance in that direction each turn. To cope with dynamic changes to the world, the unit needs to verify that its movement won't collide with anything. This requires testing the movement line against the navcell grid, and against unit obstructions (expanded by the moving unit's radius). If there is a collision then the unit will compute a short path towards the next waypoint, to move around obstacles.

To check collisions with other units, we expand every unit obstruction shape by the moving unit's radius, and test the movement line against those squares. TODO: How is the testing done, precisely? If

the start point is inside (or on the edge of) the square, then it will never collide; otherwise, if the end point is inside (or on the edge) then it collides; otherwise, it tests whether the line passes through the square (TODO: edge cases).

This means a unit must never be placed precisely on the edge of another unit obstruction – it will be able to move inside the second unit. If a unit is not inside (or on the edge), it will never be able to move so that is inside (or on the edge), so we just need to be careful when spawning new units to avoid starting too close.

TODO: Describe the navcell passability testing.

7.4 Unit spawning

(Implemented by `CCmpFootprint::PickSpawnPoint` and `CCmpPathfinder::CheckUnitPlacement`.)

When a building trains a unit, it needs to pick positions nicely located around the perimeter for the units to appear. (We don't have units walking out of a door, they just appear instantly.) A position is valid if it is on a passable navcell, and if the unit doesn't overlap with any existing units.

TODO: The implementation tests against static obstructions unnecessarily.

7.5 Foundation unit dispersal

(Implemented by `CCmpObstruction::GetConstructionCollisions` and `CCmpObstructionManager::GetUnitsOnObstru`

When a builder unit starts building a foundation, there might be other units standing in the way. (No other buildings etc can be in the way – the foundation couldn't be placed in that case.) Those units need to move out of the way, else they will be colliding with the newly-constructed building. But the builder itself shouldn't have to move (it would get stuck in an infinite loop of trying to build and then moving away and then returning and trying again).

The important thing here is the rasterized building obstruction – for each nearby unit, the unit should not be on a navcell that is blocked by the rasterized obstruction after expanding by the unit's clearance.

Since the rasterization only includes navcells that are entirely inside the obstruction expanded into a rounded-rectangle, we could use an expanded (non-rounded) rectangle as a conservative approximation to find units that might collide with the rasterization. To be certain, we should add a small tolerance value (perhaps 0.5 navcells) onto the sizes.

So: Given the building's obstruction square, loop over every unit in the world. Expand the square by unit's clearance plus tolerance. If the unit is inside that square, tell it to move outwards to a square based on the obstruction square plus clearance plus a larger tolerance.

The builder will have moved to a goal shape of the building's obstruction square plus the max build range. If the build range is strictly greater than the clearance plus tolerance, then the builder won't block the building it's building.

TODO: Implement that range restriction somewhere.

8 Building placement

(Implemented by `CCmpObstruction::CheckFoundation` and `CCmpPathfinder::CheckBuildingPlacement`.)

Buildings have various terrain restrictions (maximum slope, min/max water depth, distance from shore, etc) defined as a passability class. Players should only be permitted to place a building if every navcell under the building is valid, so that they can't make them hang over the edges of cliffs etc. Unlike the previously-described rasterization, this should be an over-estimate of the building's shape rather than an underestimate.

Overlapping structures are not a problem for the pathfinding system, except that it's weird and ugly to let players build overlapping buildings, and it's bad if players pack buildings so tightly that units get trapped or if newly trained units don't have any space to spawn into.

We can therefore do pretty much anything to determine placeability, but should expand the building's obstruction square somewhat to ensure it's not too near any other buildings or obstructed navcells.

TODO: What specifically should we choose to do?

9 Summary of constraints

- Units have (x, z) position, plus static UnitClearance and UnitRadius.
- UnitClearance should be an integer number of navcells, to ensure consistent behaviour when the terrain grid is expanded by clearance.
- UnitRadius can be any non-negative number.
- For any unit, PointToNavcell(x, z) should be a passable navcell.
  If not, the pathfinders will always try to move the unit onto the nearest passable navcell.
- For any two units, we should have

$$|x_1 - x_2| > \text{UnitRadius}_1 + \text{UnitRadius}_2$$
$$|z_1 - z_2| > \text{UnitRadius}_1 + \text{UnitRadius}_2$$

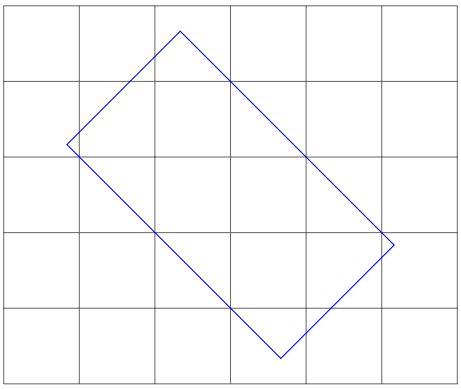
  If not, one unit might walk straight through the other.
- When a unit is targeting a building, we need

$$\text{MaxRange} \geq \text{UnitClearance} + \varepsilon$$

  to ensure the goal shape is fully on passable navcells, and is fully reachable by the vector pathfinder.
- When a unit is targeting another unit, we need

$$\text{MaxRange} \geq \text{UnitRadius}_1 + \text{UnitRadius}_2 + \varepsilon$$

  to ensure the goal shape is fully reachable by the vector pathfinder.
- To guarantee those two range constraints, we will compute MaxRange separately in each case, as UnitMaxRange + UnitClearance + ε or as UnitMaxRange + UnitRadius_1 + UnitRadius_2 + ε, with $\varepsilon = \frac{1}{8}$ (arbitrarily), where UnitMaxRange is the non-negative value specified in the unit definition.
- When units are spawned, they must be on a passable navcell. They must not collide with any unit obstruction shapes, expanded by UnitRadius + ε, with $\varepsilon = \frac{1}{8}$ (arbitrarily).
- Static obstructions must be at least $\frac{3}{2}\sqrt{2}$ (~ 2.12) in each dimension, else they might not block any navcells and units could walk through them:



10 TODO

Things to fix in the implementation:

- Enforce range constraints.
- Remove (or specify) support for `Unit` shapes.
- Fix vector pathfinder to not do quadrant stuff.
- Set up passability classes for the current units.
- Testing.
- Fix the navcell grid vectorisation.
- Make impassable-navcell-escaping work properly.
- A* heuristic overestimate with large goals.
- ...