Documentation

Table of contents

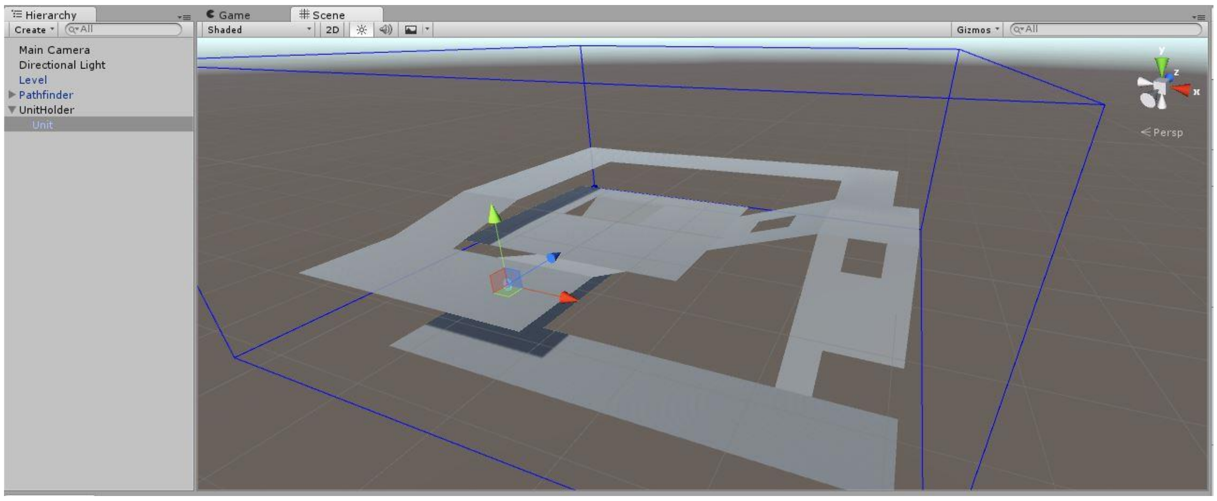
1. Setup guide
2. Pathfinder settings
3. Pathfinder cycle
4. When to use Quadtree or OcTree, and how.
5. Manual World Area Generation
6. Do's and don'ts
7. Info / thesis

Setup Guide

The tool of this assignment is made specifically for the Unity Engine; you will first need to install Unity 5.0 or higher before being able to use the tool.

We will go over the steps required to properly install and use the pathfinding tool by creating a level in which characters can move around the world by using our tool.

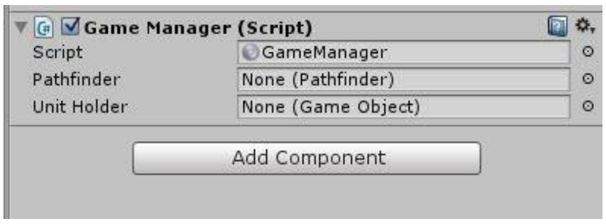
1. Open Unity and create a new project.
2. When inside your new project, import the pathfinding tool by going to Assets → Import Package → Custom Package, locate the FlowfieldPathfinder.unitypackage and open it.
3. In the Importing Package window, select “all” and press “import”.
4. To create our level, let’s start by placing the world. Go to the “FlowFieldPathfinder” folder→ Prefabs and drag the “Level” prefab directly into the Hierarchy window (located on the left side of the screen). You should now see a level mesh in your scene with a collider attached to it.
5. Now drag the “Pathfinder” prefab from the same folder directly into the Hierarchy, you will see a blue bounding box surrounding the level, and several buttons and values in the inspector on the right, when the pathfinder is selected.
6. We will go over all of the settings and features the pathfinder has later, for now let’s get our characters in. Create an empty game object by selecting “GameObject” → “Create empty”, call this game object “Unitholder”. Now drag the “Unit” prefab, from the “Prefabs folder”, into the “Unit holder”.
7. You now have a character in your level that is a child of the “Unit holder”, move it around the scene placing it on your level. Your Unity scene should look something like the figure below



Your scene should now look like this.

8. To work with the pathfinder we need a script that can access it and call its pathfinding functions, as an example there is a ”GameManager” script that we wil use. Create another empty game object, call it “Game”, and add the “GameManager” script as a component.

9. As can be seen in the figure below, the game manager needs to have a reference to the pathfinder and unitholder. Click and drag the pathfinder game object onto the pathfinder slot, and repeat this process for the unit holder.

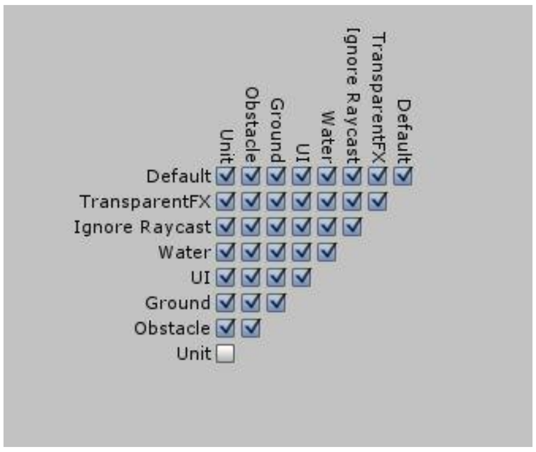


Game Manager (Script)

Script	GameManager
Pathfinder	None (Pathfinder)
Unit Holder	None (Game Object)

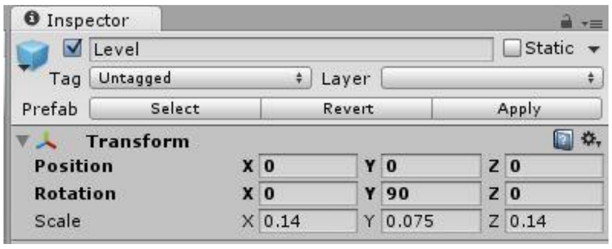
Assign the pathfinder and unitholder references to the game manager.

10. We will need to assign our game object to the proper physics layers. Go to “Edit”→ “Project Settings” → “Tags and Layers”. In the “Layers” segment we add 3 layers. We call them “Ground”, “Obstacle” and “Unit”.
11. We want to remove character collision through the Unity physics system for now. To do this: go to “Edit”→ “Project Settings” → “Physics”. Here you see a collisions matrix, make sure to disable unit to unit collisions as shown in the figure below.



	Unit	Obstacle	Ground	UI	Water	Ignore Raycast	TransparentFX	Default
Default	✓	✓	✓	✓	✓	✓	✓	✓
TransparentFX	✓	✓	✓	✓	✓	✓	✓	
Ignore Raycast	✓	✓	✓	✓	✓	✓		
Water	✓	✓	✓	✓	✓			
UI	✓	✓	✓	✓				
Ground	✓	✓	✓					
Obstacle	✓	✓						
Unit	☐							

12. Now to assign our game objects. Select the level and change its layer to “Ground” in the inspector on the right side of the screen as can be seen below. Now select our unit (not the UnitHolder) and set its layer to “Unit”.



Inspector

Level ☐ Static

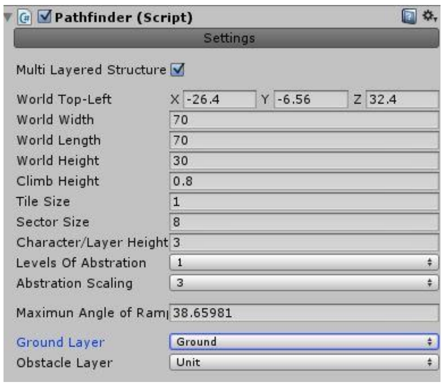
Tag Untagged Layer

Transform

Position	X 0	Y 0	Z 0
Rotation	X 0	Y 90	Z 0
Scale	X 0.14	Y 0.075	Z 0.14

You can assign the layer value in the top right.

13. Lastly select the pathfinder, in the setting you can see a ground and obstacle layer setting, make sure they are set to the correct “Ground” and “Obstacle” layer as seen below.



Multi Layered Structure	✓
World Top-Left	X -26.4 Y -6.56 Z 32.4
World Width	70
World Length	70
World Height	30
Climb Height	0.8
Tile Size	1
Sector Size	8
Character/Layer Height	3
Levels Of Abstration	1
Abstration Scaling	3
Maximun Angle of Ram	38.65981
Ground Layer	Ground
Obstacle Layer	Unit

14. We have successfully setup our system, Press the Unity “Play” button and you can let your character move around by left mouse clicking on the level, but make sure to adjust your camera first, so that it sees the whole level clearly.

    By right mouse clicking on the level you can set obstacles (right + b), remove obstacles (right + n) and adjust cost values (right + c), all of this functionality can be found in the “GameManager” script.

Pathfinder settings

We will go over the pathfinder's settings and features, to give you a better understanding of how to adjust the pathfinder to your project.

In the pathfinder's *setting menu* we see:

- **Multi layered structure:** Enable this when dealing with a level that has walkable surfaces going over each other, like we have right now.
- **World Top-Left**: This marks the origin point for the boundary box of the world, its position is shown in one of the corners of the bounding box, represented by a blue cube.
- **World Width/Length/Height**: Defines the width/length/height of the boundary box.
- **Climb Height:** This defines the maximum height difference 2 tiles can have to still be considered connected to each other. When this value is set to low, tiles on sloped areas cannot connect to each other, but don't set it higher than the "Character height".
- **Tile size:** Defines the size of tiles.
- **Sector size:** The world is segmented in sectors that are rectangles which are sized based on how many tiles fit in them. For example a sector size of 5 means a sector will be 5 tiles wide and 5 tiles long.
- **Character/Layer Height:** Defines the minimal height difference tiles must have, to not overlap. For example, if you have a bridge going over some land, the height difference between them must be bigger than the layer height. A good value to use is to take the height of your character + a little extra (i.e. 0.2).
- **2D Mode:** If you are making a 2D game on the X/Y axis, you can enable this mode to have the "Seekers" use their Y position instead of their Z. Be aware this mode is only designed for 2D games, grid(s) should be made with the custom grids (page 12) and no visual debugging tools were designed with this mode in mind. The original tool was meant for X/Z traversal on mesh geometry only, this is a workaround for those who want to use tool outside of its intended use.

- **Levels of abstraction:** Our world is divided into sectors, which forms the first layer of abstraction. Generally you will want to use 1 layer of abstraction.

  Why use 0 layers of abstraction: When dealing with a very small world (i.e. 10 by 10 tiles big) abstracting the world will not benefit the pathfinder. The abstraction of the world generates a graph over which A* searches are done to prune out unnecessary sectors. So if the world becomes too small, it will take longer to prune out sectors then to just fil the entire world with a single flow field.

  Another use of this could be for a tower defense game, in which all characters in the world need only 1 flow field path, which isn't changed very often.

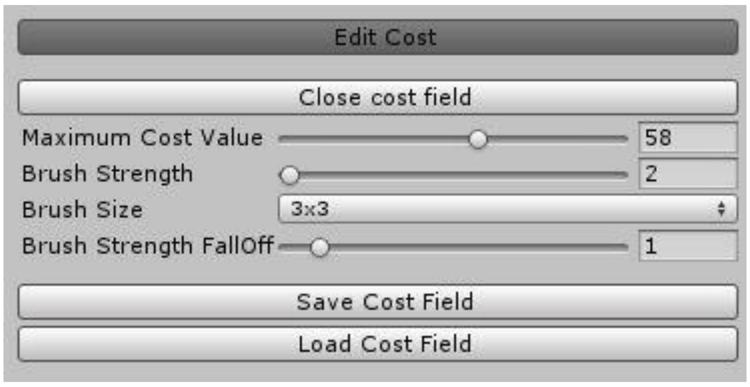
  Why use 2 layers of abstraction: When dealing with very large and/or complex worlds with many "U" shapes, a 2nd layer can speed up the HPA* search of the 1st layer significantly as it prunes out large portions of the world.
- **Abstract Scaling:** Defines how many sectors of a lower level are seen as a single sector on a higher level. For example: a scaling of 3 will result in sectors on the 2ndlayer to be as large as 3 by 3 1stlayer sectors.
- **Maximum Angle:** A non-editable value that shows the maximum angle a surface can be to be picked up properly by the tool, this value is determined by the "tile size" and "climb height".
- **Ground and Obstacle layer:** Defines if an object is seen as walkable ground, or a blocking obstacle.

While changing the settings you can press the yellow "Generate map" button, which will give you a preview of how the world will be generated.

In the *visual debugging* menu we can change what we want to draw in both the editor and during play mode.

Lastly we will talk about cost. As described in the thesis, cost makes tiles more expensive to cross over and therefore less desirable. You could apply this cost to let characters prefer to avoid certain areas, like a swamp or perhaps the edge of the world. These cost changes can be done by code, but you can also draw these values on the tiles directly in the editor. When done press the “Save Cost Field” button, so that the cost values are saved and are automatically loaded up when entering play mode.

To apply cost values in the editor we first need a preview of the world, press the “generate world” button. Next we go into the *Edit Cost* menu and open up a new cost field. You should now see a square brush on the level’s surface when hovering over it with the mouse. Hold the left mouse button while dragging over the surface to apply cost, based on how high the cost value is it will go from black to green.



Maximum Cost Value 58
Brush Strength 2
Brush Size 3x3
Brush Strength FallOff 1

We will now discuss the setting in the *Edit Cost* menu as can be seen above and how to properly use this feature.

In the pathfinder’s *setting menu* we see:

- **Open / close Cost field:** This button opens and closes the cost fields in the world.
- **Max Cost Value:** A limit set on the brush for the maximum value it can apply to a tile.
- **Brush strength:** Determines how much cost gets added to a tile every time a drag is registered.
- **Brush size:** Edit the size of the brush.
- **Brush falloff:** Each time cost (the brush strength) gets applied to a tile this cost minus the falloff will be applied to that tiles neighbor. For example: with a brush size of 3x3 and a falloff of 1 and a strength of 6, the tile in the middle will raise its cost by 6, while the 8 tiles around him will raise their cost by 5.
- **Save:** Save the cost field so that it can be loaded up when going into play mode.
- **Load:** Loads up the currently saved cost fields, allowing you to continue editing your cost fields the next time you open your project, its advised to load cost field data before opening them.

When you want to remove all your saved cost fields use the yellow “Remove Existing CostField”. Once you have opened cost fields, made your changes, saved them, and then close your cost fields, your data will be loaded up when the world is generated in play mode.

One important thing to note is that you will need to completely redo your cost fields every time you make adjustments to the world, like changing its position, adding/ removing geometry, changing the tile size, etc. These changes alter the world data structure, so the old structure used for the cost fields no longer matches and is incompatible. So keep in mind to input your cost changes when you have found the correct configuration for the project.

Scripting Reference

Here we outline 2 of the most important classes of the system, in order to give a better understanding of how to work with the tool through code.

Pathfinder

- **FindPath(Tile, List<Seeker>):** Starts a search and flow field generation for a group of seekers (characters) to a certain destination tile.
- **Vector3 GetMousePosition():** Returns a Vector3 of the position the mouse is aimed at on the world.
- **worldData:** Through the pathfinder you can access the worldData class, which holds references to many “manager” classes, which holds all the data of the world and the functions used to search for paths and generate flow fields.

WorldData

- **Tile Manager:** Used to find, access and get neighbors of tiles for several types of searches.
- **Hierarchal pathfinder:** Used for finding paths on the sector graph, returning only the sectors necessary to be traversed. These sectors can then be passed along for flow field generation.
- **Sector Manager:** Handles all operations for generating and altering sectors on any level of abstraction.
- **Integrationfield Manager:** Used for the generation of integration fields which are afterwards used for flow field generation.
- **Flowfield Manager:** Generates flow fields based on integration field data.
- **World Builder:** Contains all logic to generate the world.
- **World Manager:** Used to input world changes like cost changes and obstacle changes.

Pathfinder Cycle

Here we will describe the cycle of how the pathfinder works, how characters communicate with the pathfinder, and some of the behind the scenes processes that might not be clear.

Please take a look at the provided example script, called “Game Manager”, which shows how to work with the pathfinder.

Cycle of generation flow fields / getting your characters to move

To generate a flow field we call the “FindPath()” of the pathinder, which takes a destination tile and a list of “Seeker.cs”s that will follow it. This request is put in the “pathJobs” queue, and will be handled as soon as possible.

The path flow field will then be generated on a separate CPU thread, instead of the main thread, so that it will not lag the game.

The flow field generation will ensure a proper flow field from start to end, and is stored in the “FlowfieldManager.cs”, all characters that requested the “flowfieldpath” will get a reference to this “flowfieldpath”.

Characters will now follow the flow until they have been put in their “Idle state”, which happens as they move over their destination tile.

When characters request new flowfields, the tool keeps track of how many characters are still following each “flowfieldpath”, if a path has 0 followers, it will be deleted, preventing many flowfields from filling up memory. “pathfinder.KeepTrackOfUnitsInPaths(selectedUnits);”

Cycle of manipulating the world: cost and blocking

To edit a tile to manipulate its cost or blocked value you should use:
“pathfinder.worldData.worldManager.BlockTile(tile);”
“pathfinder.worldData.worldManager.UnBlockTile(tile);”
“pathfinder.worldData.worldManager.SetTileCost(tile, 10);”

These world changes are registered, and the pathfinder will check if a (or multiple) tile(s) is changed in its “Update” function. These world changes take priority over new flowfields being generated, as it is possible these changes will directly affect the flow of the next path request in the queue.

These changes will also affect the high level graph automatically and force flow fields to be recalculates if they go over a sector that has been changed**. In the future** I’m planning on making it possible to alter flowfieldpaths in a “smarter” way. As it currently recalculates it entirely, while this often would not be necessary.

Character movement

A character (Seeker.cs) will look at the flow value in its “currentTile” every frame and combine it with other forces to calculate its new “net force”, which will affect the velocity of the character.

The bigger the “netForce” is compared to the current velocity (which can be controlled with the “maxforce” and “maxMoveSpeed” values) the faster a character will adapt to its new desired force.

A character will follow the flow until it crosses over the destination tile and calls the “ReachedDestination()” function. Which will put it in Idle mode until a new flowfieldpath is assigned by the pathfinder.

Note: while in Idle mode, the character still calls the “FlowFieldFollow();” function, this is only because this function updates the character’s “currentTile”, which the tool needs, in order to determine if a character can move in a certain direction, this is to prevent the character from falling of the world.

However: if in your project your ensuring a character cannot fall of, due to collisions boxes around your world or your own system of tracking this, you will not need to call this function while in the Idle Mode.

QuadTree, seeker manager, Character movement

Seekers Tick() function makes the seeker move around, and gets called by the “SeekerMovementManager.cs”. The seeker manager steers all seekers and also assigns their neighbors, which affect the seeker’s steering forces.

The neighbors of a seeker are found with the help of a QuadTree, all seekers are first inserted in the quadtree, which will store the seekers in a way that it allows for fast neighbor finding.

A Quadtree is basically a square/quad that covers the world, and splits up in 4 equally sized quads whenever to many objects are inside it. Look at: https://en.wikipedia.org/wiki/Quadtree

This quadtree would then be rebuild every frame, however I don’t exactly do that. Instead I let the quadtree divide all the way to it max depth level when the game starts and don’t alter it.

This means more unnecessary memory is used, but it should be a very small footprint in your memory. The benefit is I don’t have to re-instantiate quadtrees every frame which means no garbage is generated from constantly throwing away old quadtrees. Each quadtree therefor has a “nodesAreUsed” Boolean value, which does get reset every frame, which shows whether or not you should explore its child nodes.

When to use QuadTree or OcTree, and how

The pathfinder uses either a QuadTree or OcTree which are used to store seekers in a way that it speeds up the neighbor finding process between seekers.

Please look up what these data structures are if you are unfamiliar with them, but the basics of them are that they divide the world in smaller and smaller rectangles, based on where seekers are in the world.

QuadTree: Use it when dealing with flat to flat-ish worlds, it doesn't take height differences in account, but is faster than an ocTree. I would recommend that a world similar to the one in the example scene is better of using a quadTree.

OcTree: Use it when dealing with square maps with a lot of height differences (floors), for example a 6-story high office building.

How use them?: In the "SeekerMovementManager.cs" the quad- or oc- tree is used, with a couple of function written for both of them, you can just remove/comment out based on which of the 2 is better for your project.

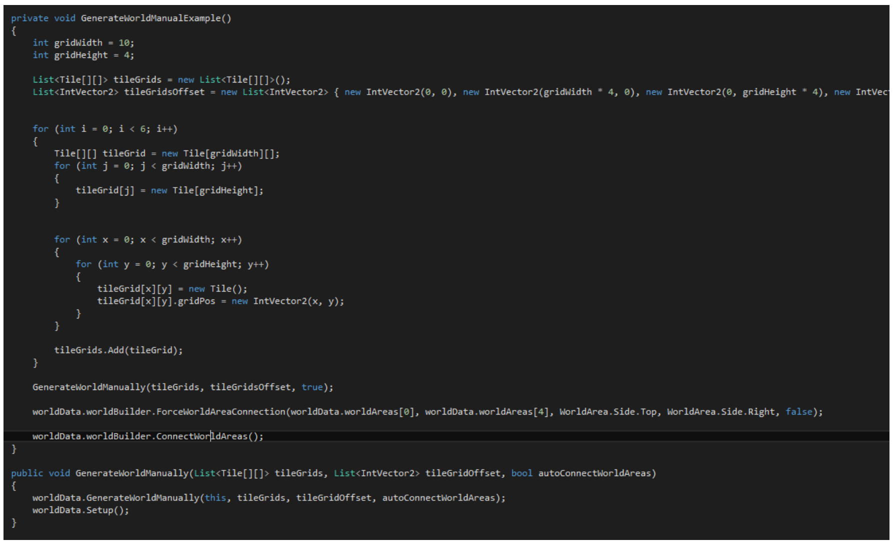
Important: make also sure to update which tree you are using in the **"ReachedDestination"** function of **"Seeker.cs".**

How do they work: Unlike true quad/oc trees, I do not complete remove all their child nodes every update and rebuild them based on seeker positions. Instead I let the trees split all the way down to their maximum depth level at the start. They each have a "nodesInUse" Boolean that states whether or not the child nodes are used, which makes sure I do not go into deeper levels that are unused. So just this Boolean and the "objects" list gets cleared every update.

After they have been cleared, seekers are inserted properly. Once all inserted, each seeker finds its neighbors with the help of the tree. And lastly the seeker moves with its Tick().

Manual World Area Generation

While the asset generates world areas with tile grids based on raycasting at the start. It is also possible to define worldAreas yourself in code and connect them the way you want. This can be used for parts of the world that come into existing later on in the game and also for 3D flow fielding.

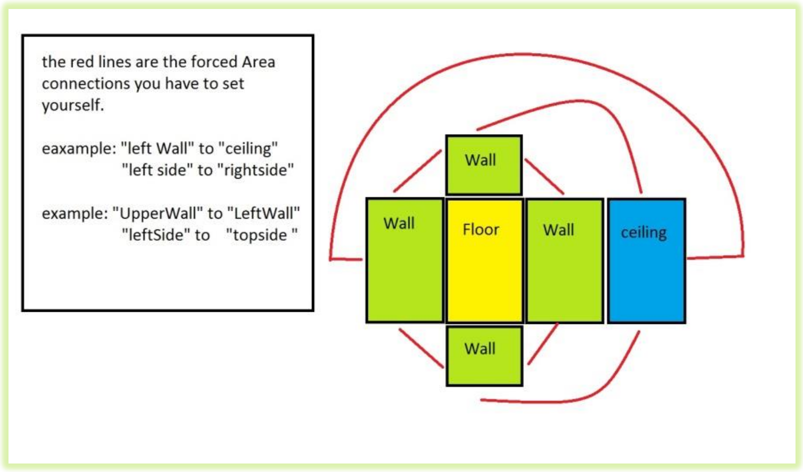


```
private void GenerateWorldManualExample()
{
    int gridWidth = 10;
    int gridHeight = 4;

    List<Tile[][]> tileGrids = new List<Tile[][]>();
    List<IntVector2> tileGridsOffset = new List<IntVector2> { new IntVector2(0, 0), new IntVector2(gridWidth * 4, 0), new IntVector2(0, gridHeight * 4), new IntVec


    for (int i = 0; i < 6; i++)
    {
        Tile[][] tileGrid = new Tile[gridWidth][];
        for (int j = 0; j < gridWidth; j++)
        {
            tileGrid[j] = new Tile[gridHeight];
        }


        for (int x = 0; x < gridWidth; x++)
        {
            for (int y = 0; y < gridHeight; y++)
            {
                tileGrid[x][y] = new Tile();
                tileGrid[x][y].gridPos = new IntVector2(x, y);
            }
        }

        tileGrids.Add(tileGrid);
    }

    GenerateWorldManually(tileGrids, tileGridsOffset, true);

    worldData.worldBuilder.ForceWorldAreaConnection(worldData.worldAreas[0], worldData.worldAreas[4], WorldArea.Side.Top, WorldArea.Side.Right, false);

    worldData.worldBuilder.ConnectWorldAreas();
}

public void GenerateWorldManually(List<Tile[][]> tileGrids, List<IntVector2> tileGridOffset, bool autoConnectWorldAreas)
{
    worldData.GenerateWorldManually(this, tileGrids, tileGridOffset, autoConnectWorldAreas);
    worldData.Setup();
}
```

In “Pathfinder.cs” “Awake()” is an example of how to define your own worldareas.

Please be aware of the following:

- Don’t connect the same WortldAreas more than 1 time; only connect them on a single edge.
- When manually connecting areas, its advised to keep your world areas rectangular
- Manual WorldAreas are by default placed at the bottom of the Pathfinding Bounding box.


the red lines are the forced Area connections you have to set yourself.
eaxample: "left Wall" to "ceiling"
"left side" to "rightside"
example: "UpperWall" to "LeftWall"
"leftSide" to "topside "
Wall
Wall
Floor
Wall
ceiling
Wall

Above is an example of how to layout worldAreas in a 2d grid, and connecting the sides so that it corresponds with the right Area. This allows for flow fields to match a 3D environment, of the flow going along walls.

Do’s and Don’ts

Here we will go over a couple of examples of how to use the tool in certain scenarios and things that I have seen users do incorrectly that was not explained in the documentation.

Don’t

- **Block a tile a character is currently standing on**, (changing cost is fine), as the tile gets changed to be blocked of, the character inside will be locked inside the tile, but more importantly flowfieldpaths will be recalculated, once calculated, the character inside has no flow data, which means he will request the flowfieldpath to be updated/changed, so that he can continue, but this will **never happen**, as the character is inside a blocked tile.

  As a result, every character inside a blocked tile, will request a new flowfieldpath every frame forever, this will overload the pathfinder queue and your project will suffer greatly from it.

- **Modify the “flowFieldPath” value of a seeker directly**, make sure to only manipulate this value through one of the provided functions of the asset, manipulating its value will affect all characters involved. Removing the reference, will result in the “pathfinder.KeepTrackOfUnitsInPaths()” being unable to detect when a flowfieldpath has 0 seekers, this means that **the path will never be deleted and will fill up memory.**

Do's / example scenarios

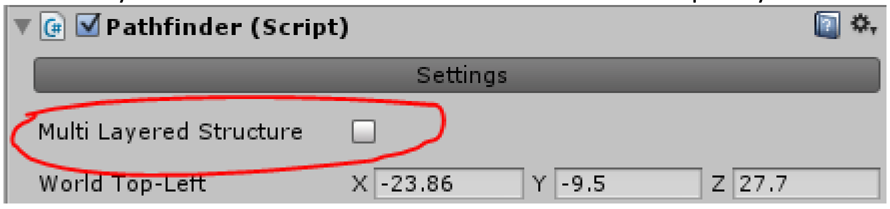
- **I want my character to stop when inside a certain radius**. To let a character stop use its "ReachedDestination()" function to stop and switch over to the Idle mode.

- **I want my characters to move around a character that has stopped**. I suggest manipulating the cost of the tile the stopped character is standing on. It will update the flow to go around the tile/character, usually an increase of just 5 should already be enough, unless you are dealing with a lot of manipulated terrain, the key is to make the cost higher than that of the tile's neighbors.

- **I want to use Unity's collision system to keep characters from clipping through each other.** You can enable collisions between your characters, but you will also have to put collision boxes around your world, holes, ramps, etc.

  Sadly it is impossible for me to ensure characters will not fall of the world when there are collisions, as the slight position changes the Unity physics system performs, are not detectable / reversible by me. That is why you will have to use collision boxes to prevent this from happening.

- **I want to combine this asset with Unity Terrain**. The tool works fine with Unity Terrain, but make sure to only use it in single layered mode, since the tool currently does not work well with many roundedoff surfaces in combination with multiple layers.


Pathfinder (Script)

Settings

Multi Layered Structure ☐

World Top-Left X -23.86 Y -9.5 Z 27.7

Info / Thesis

Introduction

Pathfinding is an essential component in many games. It is a system that returns a path for characters to follow to their goal in an efficient manner.

The data returned by our system is a flow field. Flow fields are 2-dimensional grids of vectors who each point to a neighboring vector, following the directions of the individual vectors will eventually lead to the destination. Flow fields are usually slower to generate than simple paths, but they can be used by many characters at once. This makes them more efficient when dealing with large crowds. Due to their long computation time, flow fields become inefficient on large maps. That is why we use hierarchical pathfinding in conjunction with flow fields to minimize this issue.

Hierarchical pathfinding A* (HPA*) searches a path with A*[1], [2] on an abstract representation of the world. This abstract representation is made by dividing the world into sectors, each containing a set of nodes. These sectors are then used to search a path on the less complex sector level which gives us a high-level sub-optimal path. This high level path is then used to identify which sectors must be traversed to reach a character's destination, only these sectors are then filled in with flow fields. This minimizes the amount of fields, making them better suitable for large worlds, while also ensuring a correct solution for the character's navigation.

We will go more in-depth into how the pathfinding system defines the game's world and explain the methods used throughout it to generate flow fields.

Word representation & generation

The system represents the world by using grids, which are easy to work with and are necessary to create flow fields. A grid is a two dimensional array of squares, like a chessboard, which stores the area our characters can move on [8].

However two dimensional grids cannot support 3D terrain that features overlapping, like a tunnel or overhang, which is why we use multiple grids, so that no grid can have a "missing" or double tile due to walkable surfaces being above one another. Each area of the world that contains its own tile grid and sector grid is called a "World Area, these can be seen in Fig1.

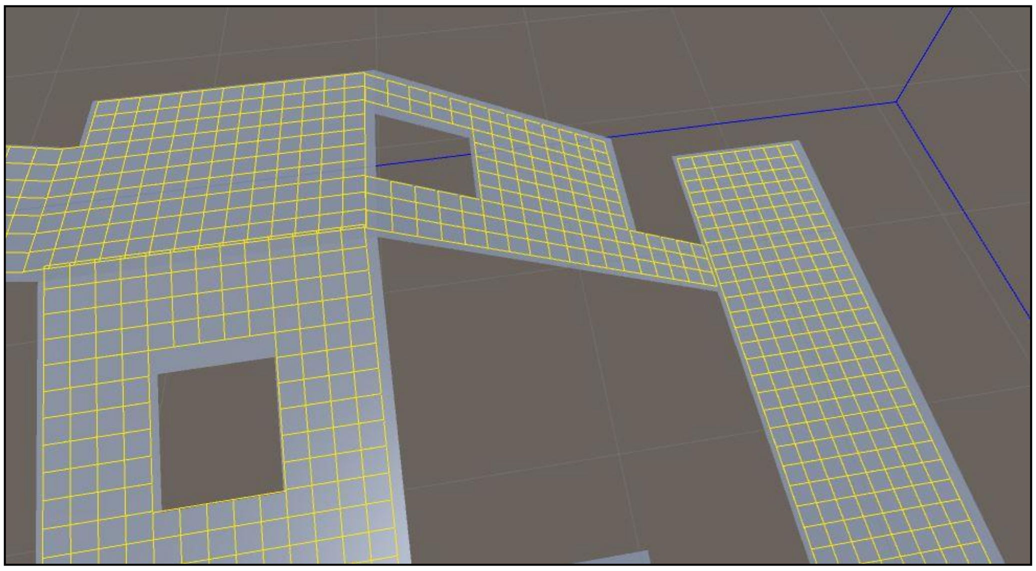


Fig 1. Five world areas with tile grids

Generation of the world starts off by cutting the world into temporary "height-layers". These layers each have a tile grid that covers the boundaries of the world in the X and Z axes, which are specified by the user.

Each possible tile position in the X and Z axes is checked by using ray-casts. The ray starts at the top of the world boundary box, shooting straight downwards. If the ray hits a collider that has been set to the correct Unity physics-layer, it will be seen as a valid tile position and a tile will be created in the appropriate height-layer.

After checking every possible position with ray-casts we convert the height layers into world areas. We iterate through all height-layers, select a tile from its grid, and do a wave expansion from the tile. This expansion collects all tiles that are of the same angle, and it can cross over from one height-layer to the next. The expansion stops when no more tiles at the same angle can be found. All tiles found by the expansion are then removed from their height-layers and are converted into a single world area. We repeat this process until all tiles are removed from the height-layers, when the height-layers are empty they are removed.

By following this method all tiles are grouped into world areas based on the angle between them. This way we end up with flat and sloped areas as can be seen in Fig1. These areas then store which tiles connect them to other areas, making it possible to move from one to the other.

Sectors & HPA*

Each world area not only has a grid of tiles it also contains a grid of sectors. A sector is a rectangle area that covers multiple tiles, so for example a sector could be 10x10 tiles. These sectors are connected to each other based on the tiles on their edges. On these connections (on both sides of the edge) abstraction nodes are placed, these nodes are then connected with each other and store their travel distance, in this case 1.

Now that we have nodes connecting different sectors to each other we want to connect these nodes to other nodes of the same sector. We do a wave expansion from a node to fill the sector and find the other accessible nodes in the sector. This gives us which nodes can connect to each other and also the distance between them. This process is repeated until we have expanded from each node, creating a high-level graph such as featured in Fig2. Lastly we do a similar node creating and connecting step between world areas. By linking all these abstraction nodes we can now do rough high level searches across the game world as seen in Fig3 [1].

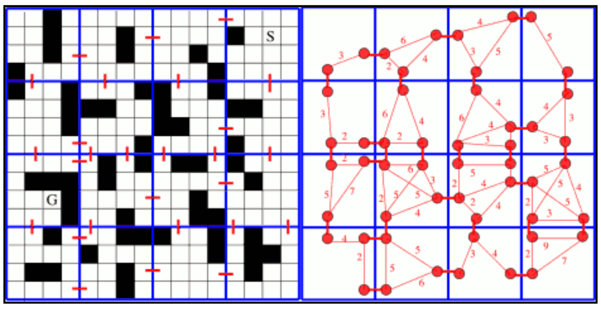


Fig 2. HPA* divides the map into sectors, these sectors are linked at their borders creating a higher-level network of nodes

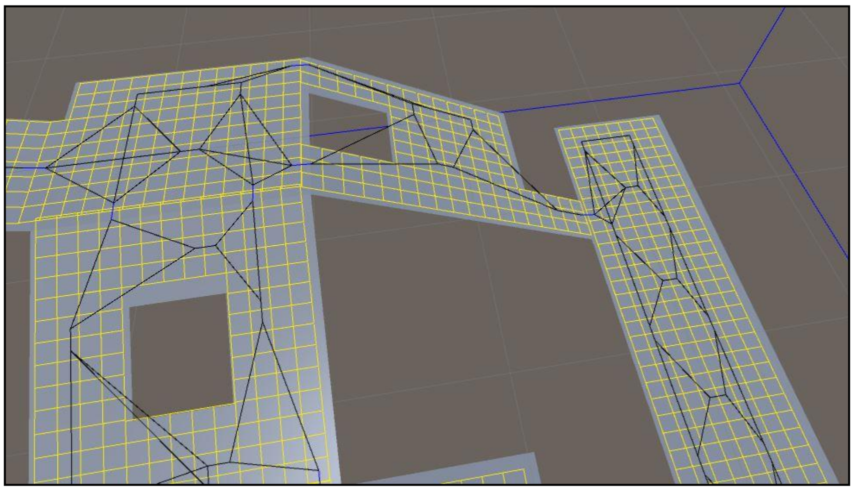


Fig 3. Node connections from several sectors in black, blue lines show the world area connections (Note that the lines only represent which nodes are connected, not the path or distance between them)

The nodes that we have created have also been linked to each other, creating a graph of connections on which we can do rough searches to find a path. This search starts off with the start and destination node being inserted into the graph, then A* is applied to the graph to find the shortest path [2]. This high level path gives us which sectors must be traversed in order to reach the destination.

Cost-, Integration- & Flow-fields

Now that we can find which sectors need to be traversed we can generate flow fields only for those sectors, so that we avoid spending time on generating flow fields for the entire world, most of which would not be used.

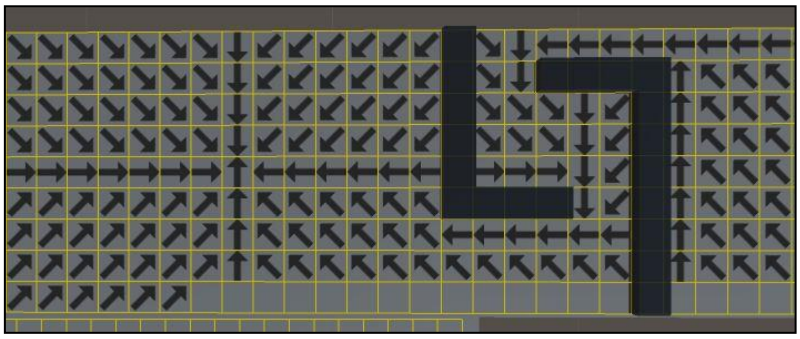


Fig 4. Flow Field: following the vectors leads to the destination

A flow field is a collection of vectors that characters can follow to move around the world as each vector corresponds with a tile as shown in Fig4. The direction of a flow vector is affected by its surroundings; a vector will not point directly at a wall. We can also assign a cost value to a tile, making it less desirable to cross. This could for example be used for a swamp [6].

Each tile in the world has a default a cost value of 1. By increasing the cost value we make it more “expensive” to travel over it, we try to get to our destination with as little effort as possible, in other words, we search for the fastest path not the shortest.

Manipulating cost values is an easy way for a user to influence the movement of their characters, as they will try to avoid tiles with high cost. Every time the wave expansion expands from one tile to the other, it does not simply increase the total distance by 1 but by that tile’s cost value. This cost also affects the connection distance between nodes on the higher level graph as talked about earlier.

These cost changes can also be done in the editor as shown in Fig5, users can save the cost values they have edited and load them in the moment the game starts. Being able to draw these cost values on the world surface, makes it easier to input the proper values around static objects and areas [5].

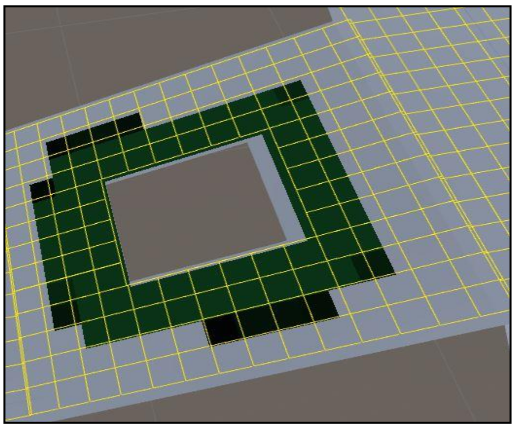


Fig 5. Cost drawn on the world’s surface

We talked about how character movement can be influenced by editing the cost values of tiles; we will now discuss how this is achieved.

To create a flow field you first need an integration field. This is similar to a flow field, but instead of vectors being stored for each tile, we store a number value. This number value tells you how costly it is to reach the destination. In the figure below this number or integration value is shown by coloring the tiles from (low) yellow to (high) green [6].

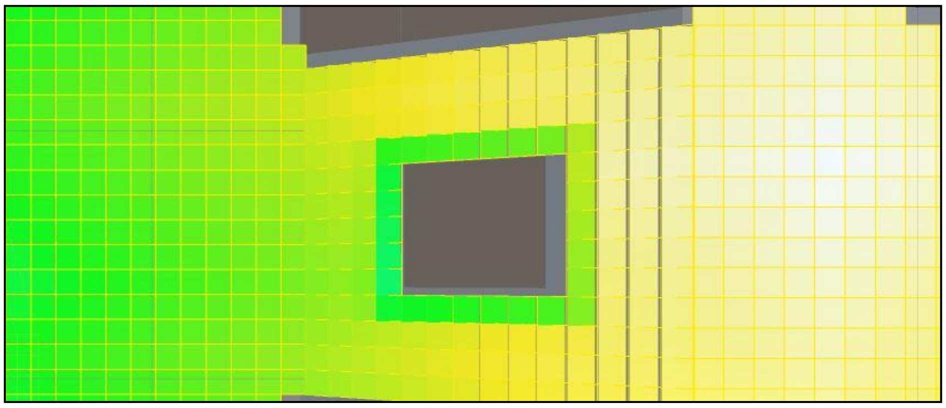


Fig6. Integration field: The search starts from the white tile outwards. The difference in cost from figure 5 can be seen here, as the integration value is higher than the surrounding tiles.

To create an integration field we do a wave expansion from the destination tile across the world. The expansion is limited to the sectors that have been found in the high-level search. This expansion goes through all tiles using their cost values. When we expand from a tile to its neighbors, we alter their cost value by adding the cost of the tile we started from. When the expansion is done all cost values are stored in a 2 dimensional array. This array is an integration field, in which each value reflects the travel cost between the matching tile and the destination node.

Finally we can create a flow field [7]. We go over each tile with an integration value and create a 2D vector that points to the tile's neighbor with the lowest integration value (out of 8 possible neighbors). We now have our flow field; each tile location has a matching flow vector that will lead you in the direction of the destination, which is reached by going from tile to tile. The figure below shows how the flow direction is determined by the integration values, as the flow vectors point towards the lowest integration value.

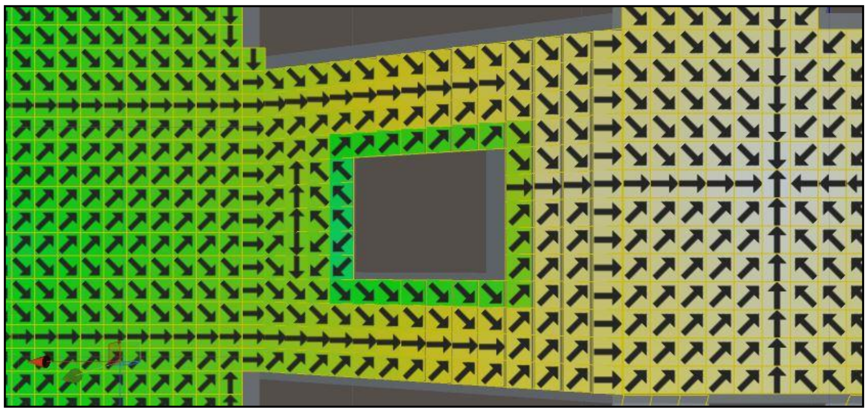


Fig 7. Flow Field: generated by making tiles point towards their neighbor with the lowest integration value.

Fig 8 and 9 show integration fields and flow fields on a larger scale, with the integration values being represented in color. From low to high value: yellow, green, blue, and purple.

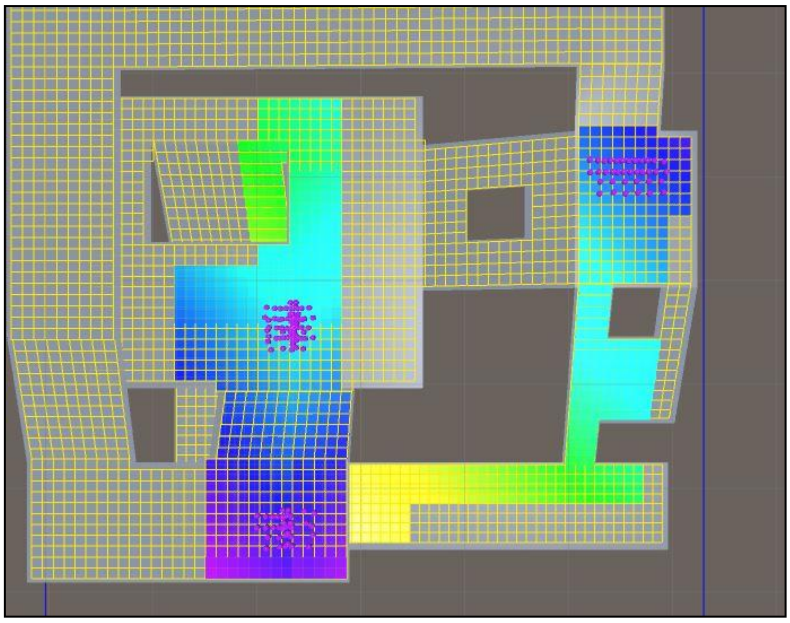


Fig 8. Integration field: search starts from the lower yellow region outwards to the 3 groups of purple characters

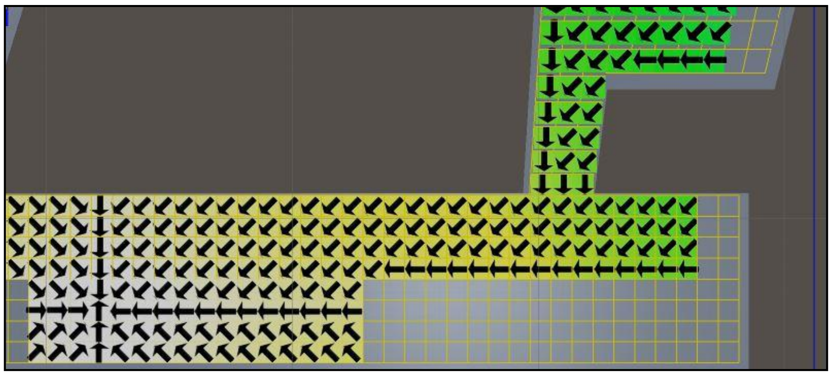


Fig 9. Flow field: generated by making tiles point towards their neighbor with the lowest integration value.

World changes

Our system supports changes to the world being made such as tiles being marked as walkable or blocked and their cost being editable, these changes can lead to parts of the world being cut-off from each other or at least slight changes in how characters should travel to their destination.

After one or multiple tiles have been changed, their sectors need to recalculate the distances between their edge abstract nodes and also re-build the sector nodes on specific edges, if these tiles match the edges. This ensures that future HPA* searches and flow fields will be correct, but fields created before the changes need to be updated [1].

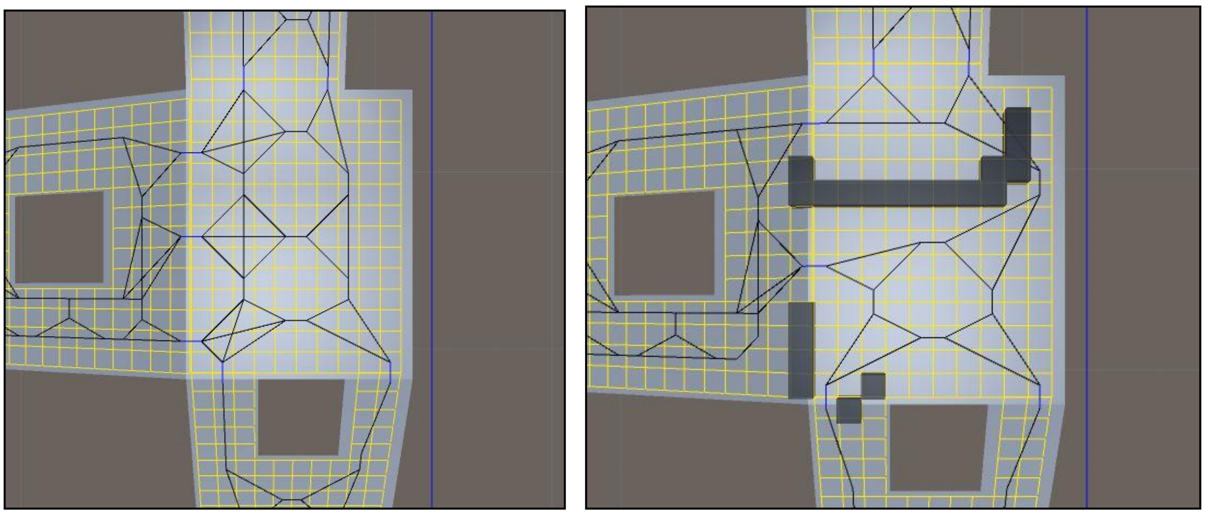


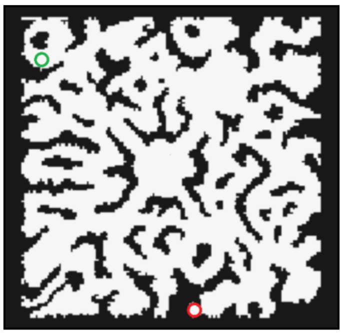
Fig 10. Sector graph before and after marking tiles as blocked

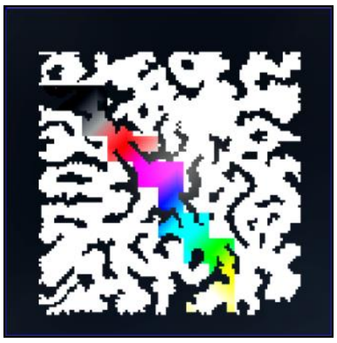
Performance

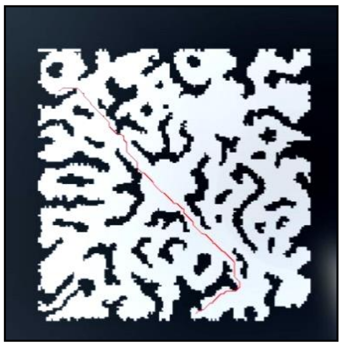
To better determine the suitability of the pathfinder in different scenarios, we examine the average execution time on two game maps. We also examine the execution times of an A* implementation, which uses the same data structure and narrows down its search with HPA* just like our tool.

In the examples the start location is highlighted in green and the goal in red. The execution times below are the average time of running the pathfinding solutions 200 times to find a path from start to end with a single layer of abstraction.

Map 1, *Garden of War,* Warcraft 3







Flow field A*

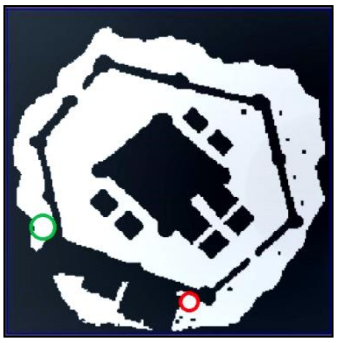
Table 1. Flow field times

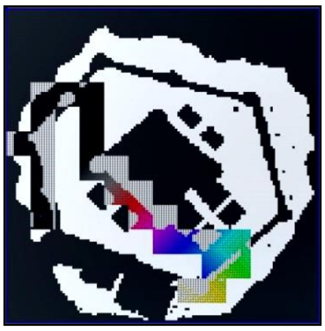
Pathfinder vs world size	64x64	128x128	256x256
Sector size 5x5	1.77ms	4.2ms	11.4ms
Sector size 10x10	2.55ms	4.76ms	10.7ms
Sector size 15x15	2.9ms	6.3ms	13ms
Sector size 20x20	4.02ms	9.6ms	18.0ms

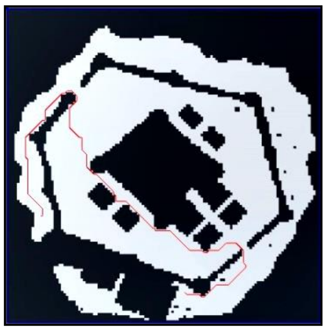
Table 2. A* times

Pathfinder vs world size	64x64	128x128	256x256
Sector size 5x5	1.3ms	2.3ms	3.7ms
Sector size 10x10	1.7ms	1.7ms	3.7ms
Sector size 15x15	1.5ms	1.72ms	2.83ms
Sector size 20x20	0.74ms	1.97ms	3.45ms

Map 2, Baldur's Gate 2







Flow field A*

Table 3. Flow field times

Pathfinder vs world size	64x64	128x128	256x256
Sector size 5x5	4.5ms	10.5ms	17.4ms
Sector size 10x10	4.6ms	6.3ms	15.1ms
Sector size 15x15	5.9ms	8.2ms	19.8ms
Sector size 20x20	7.0ms	10.2ms	23ms

Table 4. A* times

Pathfinder vs world size	64x64	128x128	256x256
Sector size 5x5	1.7ms	3.5ms	12ms
Sector size 10x10	1.5ms	2.15ms	5.3ms
Sector size 15x15	2.16ms	2.5ms	4.6ms
Sector size 20x20	1.8ms	1.78ms	4.1ms

The results show that A* is always faster than flow fields, which is what we expected since flow field generation is more complex. When looking at the results in table 1 and 2 we can see that generating a flow field path is between 2-5 times slower than generating an A* path with the same world size and sector size. The same applies to the ratios between table 3 and 4.

While a flow field takes longer to generate, these differences also show the strength of flow fields for group pathfinding. A single flow field path could be used by dozens of characters, but when using A* we would have to generate a path for each character individually. Our results show that flow field pathfinding is the faster solution when dealing with crowds above a certain size, in our case about 6 characters.

Using the right sector size significantly improves execution times for both methods; this can be seen clearly in the last column of table 4. A 5x5 sector size has become so small compared to the world size that the HPA* search takes considerably longer, when compared to the other execution times in the same column.

References

1. Alex J. Champandard, *Near-Optimal Hierarchical Pathfinding (HPA*)*, http://aigamedev.com/open/review/near-optimal-hierarchical-pathfinding/ (October 11, 2007, accessed 29 May 2015)
2. Adi Botea. Martin Muller. Jonathan Schaeffer, *Near Optimal Hierarchical Path-Finding*, https://webdocs.cs.ualberta.ca/~mmueller/ps/hpastar.pdf (2006, accessed 29 May 2015)
3. Andrew Fray and Graham Pentheny, *Improvements in AI steering behaviors*, http://gdcvault.com/play/1018262/The-Next-Vector-Improvements-in (2013, accessed 29 May 2015)
4. Craig Reynolds, *Steering Behaviors For Autonomous Characters*, http://www.red3d.com/cwr/steer/ (1999, accessed 29 May 2015)
5. Uber Entertainment, https://youtu.be/5Qyl7h7D1Q8?t=1469 (March 2013, accessed 29 May 2015)
6. Sidney Durant, *Understanding Goal-Based Vector Field Pathfinding*, http://gamedevelopment.tutsplus.com/tutorials/understanding-goal-based-vector-field-pathfinding--gamedev-9007 (July 2013, accessed 29 May 2015)
7. Elijah Emerson, *Game AI Pro*, (New York, CRC Press, September 2013), pg. 307 – 316
8. Amit Patel, *Map Representations*, http://theory.stanford.edu/~amitp/GameProgramming/MapRepresentations.html (2011, accessed 29 May 2015)
9. Amit Patel, http://theory.stanford.edu/~amitp/GameProgramming/ (2011, accessed 29 May 2015)
10. Anonymous, *Binary heap*, http://en.wikipedia.org/wiki/Binary_heap (May 2015, accessed 28 May 2015)