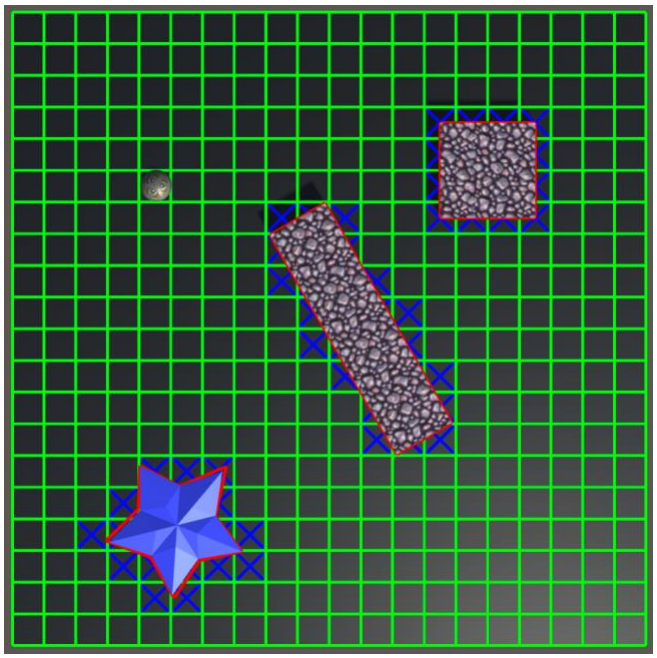
Homework 1

Homework 1: Grid Navigation

Before starting this assignment, watch the lectures regarding Grid Lattice and Computational Geometry intro.

In this assignment you will be implementing the creation of a grid lattice to support planned path agent movement in a game scene.

One of the main uses of artificial intelligence in games is to perform *path planning*. This is the determination of a sequence of movements through the environment that gets an agent from one location to another without running into any obstacles. For now, we will assume static obstacles. In order for an agent to engage in path planning, there must be a topography for the agent to traverse that is represented in a form that can be efficiently analysed (e.g., a *discretized space*). The simplest topography is a grid lattice. Think of an imaginary lattice of cells superimposed over an environment such that an agent can be in one cell at a time. Moving in a grid is relatively straightforward: from any cell, an agent can traverse to any of its four (or sometimes eight) neighboring cells.



In this assignment, you will write the code to superimpose a grid over any given terrain so that an agent can navigate according to connectivity (4-way or 8-way) of the cells. The code to generate the grid should work on any terrain such that an agent can never collide with an obstacle.
But first, you need to become familiar with the Unity game engine in which you will be working.

What you need to know

In Unity, we work with Game Objects to achieve the behavior we desire. You can see a list of these Game Objects in the Hierarchy tab. The terrain on which you'll be working is represented using the 'NavigationArea' game object (a plane geometry). Each Game Object can have multiple scriptable components attached to them. By 'Scriptable', we mean that the components behavior can be programmed. You can see a list of these components in the Inspector view when you click on the 'NavigationArea' Game object.

However, the preparation of this assignment is such that your efforts are somewhat isolated from the overall game event loop. Instead, you just need to implement algorithms according to the defined interfaces. If you do so correctly, the interactive elements will work. You may however be interested to explore the complete project. In particular, the code that calls the algorithms you implement.

For this assignment, you will be editing a file that the 'Game Grid' component calls. In the project view, you can expand Assets/Scripts/GameAIStudentWork/GridNavigation/ to find the 'CreateGrid' file. The ../../FrameWork/'GameGrid' file calls 'CreateGrid' in order to create the grid overlay data structures. If you double click the Script, this will open the C# Script in an editor.

There are other objects within the scene that are worth exploring including the agent and the obstacles.

Every iteration of the game loop, called a *tick*, the Update() method is called on all dynamic objects and their components and the scripts associated with them.

Below are the important bits of information about objects that you will be working with or need to know about for this assignment.

CreateGrid

Location: Assets/GameAIStudentWork/GridNavigation/CreateGrid.cs

This is the single file that contains your homework. Incomplete methods with placeholder scaffolding are provided for you to implement (see below).

Create()

This is a method that you need to implement. It is a public method called by other code and will be tested during grading.

It creates the grid lattice discretized space data structure of a scene.

It should pass back a grid, which is represented using a two-dimensional Boolean array. For each element in the grid, it is true if the agent can pass through it and false if it cannot. This is determined by whether the grid cell is blocked by an obstacle. Obstacles must overlap with the interior of the cell to count as blocked (just touching is still traversable).

It is recommended that you use Is*PointInsideAxisAlignedBoundingBox()* as a helper method that you also implement.

Parameters:

canvasOrigin: bottom left corner of navigable region in world coordinates

canvasWidth: width of navigable region in world dimensions

canvasHeight: height of navigable region in world dimensions

cellWidth: target cell width (of a grid cell) in world dimensions

obstacles: a list of collider obstacles (polygons)

grid (out): bool[,].

A cell at grid[i, j] is set to true if navigable, false otherwise. The first array dimension (i index) is associated/aligned with the positive x axis of the canvas. The second array dimension (j index) is associated/aligned with the positive y axis of the canvas. Note that the j index is in the major axis (memory contiguous axis) of the grid[] array for this application.

On the screen, the bottom left corner of the cell at grid[0, 0] is aligned with the bottom left corner of the canvas.

The returned grid's dimensions must be as follows: grid_size_x is the largest integer such that grid_size_x * cellWidth is less than or equal to the canvasWidth, and grid_size_y is the largest integer such that grid_size_y * cellWidth is less than or equal to the canvasHeight. If either grid_size_x or grid_size_y would be 0, then a value of 1 will be used instead.

Example of cache coherent array iteration:

```
for ( int i = 0 ; i < grid_size_x ; ++i ) {
    for ( int j = 0 ; j < grid_size_y ; ++j ) {
        // cell is grid[i, j]
    }
}
```

IsTraversable()

This is a method you need to implement. It is a public method called by other code and will be tested during grading.

Returns true if the grid is traversable from grid[x,y] in the direction dir, false otherwise. The grid boundaries are not traversable. If the grid position x,y is itself not traversable but the grid cell in direction dir is traversable, the function will return false. Returns false if the grid is null, or any dimension of grid is zero length. Returns false if x,y is out of range.

For this assignment, 8-way connectivity diagonals are considered traversable if the current cell is traversable and the cell in the diagonal direction dir is traversable (e.g., Up-Left). The diagonal direction is considered traversable even if the cells in horizontal and/or vertical direction are blocked (e.g., Up and/or Left cells). When considering diagonal connectivity, you don't need to consider the adjacent cells (e.g., Up or Left). However, that can make sense for many video game scenarios.

Return: a Boolean reflecting traversability of direction for grid coordinate

grid: the 2D Boolean grid to test a cell for traversability in a direction

x: grid index in x dimension (e.g. grid[x,y])

y: grid index in y dimension (e.g. grid[x,y])

dir: traversal direction enum. Eight ways are identified. Up/Down is in Y direction. Up is positive. Left/Right is in X direction. Right is positive.

NOTE: 4-way versus 8-way connectivity is determined external to IsTraversable(). So you don't need to worry about what the current setting is. Just evaluate whichever **dir** is passed.

The following methods internal to your code. It is recommended that you implement them and use them in your solution. However, you might choose to use a different approach.

IsPointInsideAxisAlignedBoundingBox()

This method tests containment of an integer vector point with the integer vector coordinates of a bounding box (defined by the min and max dimensions). The coordinates are world dimensions mapped to integer representation. The method returns true if the point is inside or on the edge of the bounding box. You might think this is in disagreement with the *Create()* method requirements. Please see the grid lattice lecture for discussion. Returns false if the point is outside the bounding box.

Return: A Boolean reflecting the point containment test

minCellBounds: Vector2Int of the minimum corner (x,y)

maxCellBounds: Vector2Int of the maximum corner (x,y)

p: Vector2Int test point to test whether it is inside the cell

IsRangeOverlapping()

Determines if the range (inclusive) from min1 to max1 overlaps the range (inclusive) from min2 to max2. The ranges are considered to overlap if one or more values is within the range of both.

Preconditions: min1 <= max1 AND min2 <= max2

Return: Returns true if overlap, false otherwise.

min1, max1: int of minimum/maximum of range 1

min2, max2: int of minimum/maximum of range 2

IsAxisAlignedBoundingBoxOverlapping()

IsAxisAlignedBouningBoxOverlapping(): Determines if the AABBs defined by min1,max1 and min2,max2 overlap or touch Returns true if overlap, false otherwise.

Preconditions: min1 <= max1, per dimension. min2 <= max2 per dimension

Return: Returns true if overlap, false otherwise.

min1, max1: Vector2Int of minimum/maximum of range 1

min2, max2: Vector2Int of minimum/maximum of range 2

Game Grid

Location: Assets/Scripts/Framework/GameGrid.cs

You don't actually modify this file but be aware that GameGrid calls CreateGrid.Create() and CreateGrid.IsTraversable() (your static functions) to actually create the grid, support path search, etc. It dictates what arguments are passed to your functions.

Presets

Location: Assets/Scripts/Config/CustomPresetConfig.cs

This is another file that you aren't required to modify but may choose to (regardless, you won't be turning it in). You may find it useful to add special presets for debugging your CreateGrid.Create() code. Certainly, you should test with all presets provided via the defined numerical keypresses!

GridTest (Unit/Integration Testing)

Location: Assets/Scripts/Tests/GridTest.cs

This is another file that you won't be submitting.

However, you are HIGHLY encouraged to write a battery of tests to make sure your submission works as expected.

Miscellaneous utility functions

Helper Methods you probably need to use:

- Mathf, See FloorToInt()
- Methods at the top of CreateGrid
- Polygon member methods. See getIntegerPoints()/getPoints(). The getIntegerPoints() returns integer-discretized Vector2Ints whereas getPoints() returns floating point Vector2s.
- Also, you might find Polygon.MaxIntBounds/MinIntBounds useful for Axis-Aligned Bounding Box (AABB) overlap tests as an optimization.

In addition, you might want to check the array and list methods of C# and Unity Engine's Vector2/Vector2Int class. Also, C# System.Tuple may be useful.

- https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1?view=netframework-4.8
- https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/arrays/
- https://docs.unity3d.com/ScriptReference/Vector2.html
- https://docs.unity3d.com/Manual/index.html
- https://docs.unity3d.com/ScriptReference/

Instructions

First, watch the lectures regarding Grid Lattice and Computational Geometry intro.

You must superimpose a grid over an arbitrary, given game world terrain consisting of obstacles. The grid is a 2D array of Booleans such that a *false* in any particular cell means the Agent cannot walk into the cell and a *true* in any particular cell means the Agent can walk into the cell.

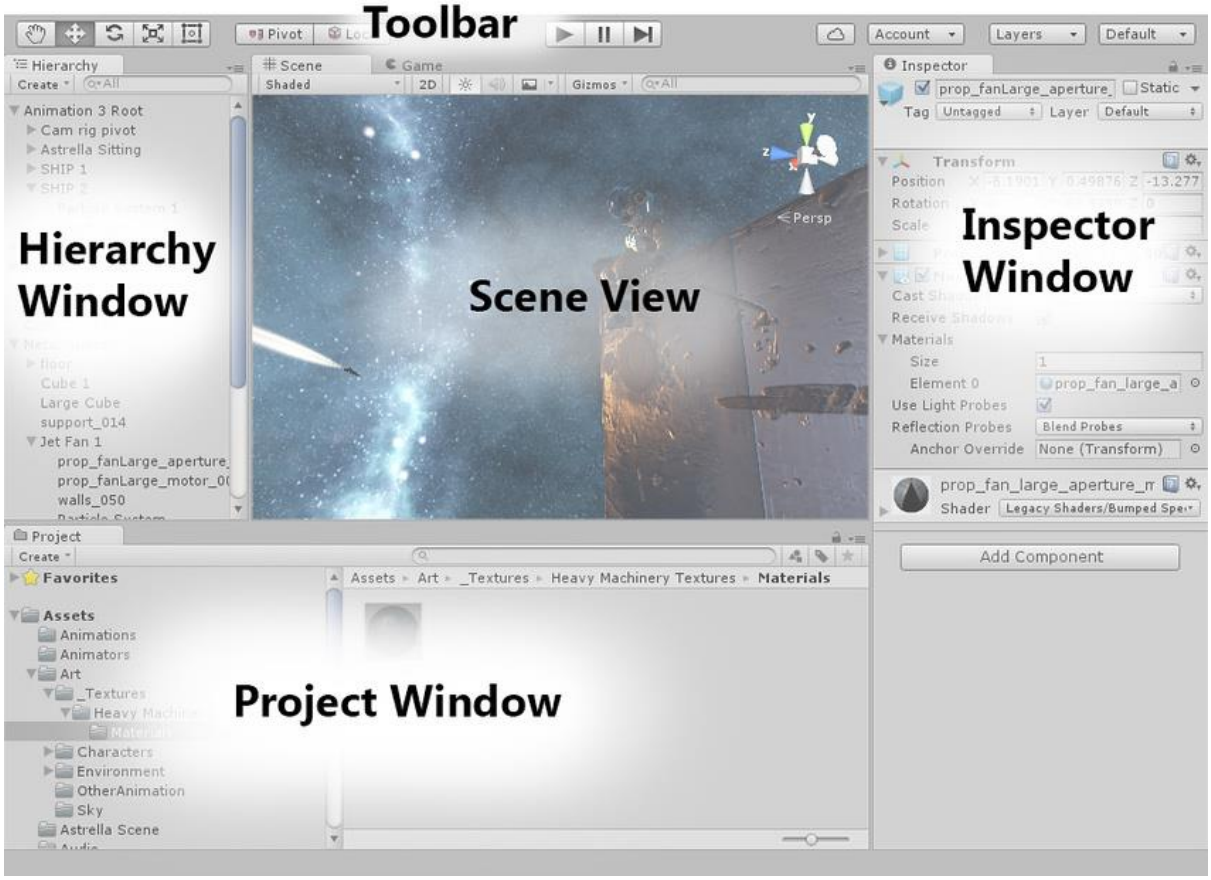
When you click on the screen, you indicate where you want the Agent to traverse.

Step 1: Download and install the Unity version specified for this course (see course introduction materials).

Step 2: Download the Unity project for this assignment
https://github.gatech.edu/IMTC/GameAIPathPlanning

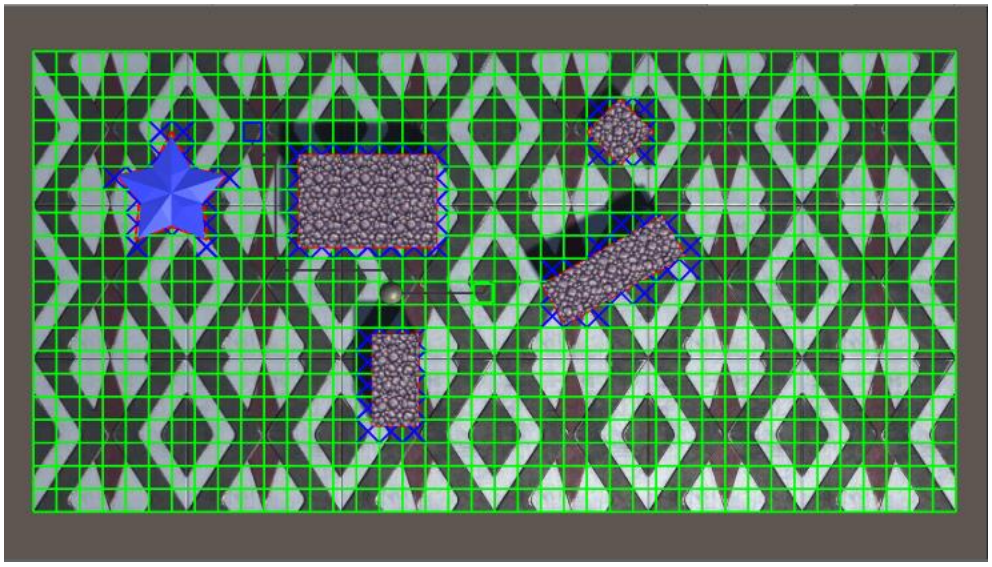
Step 3: Open Unity and load this project. Click on File -> Open Project and select the project folder for this assignment

Step 4: If not already open, open GridNavigation. Click on File -> Open Scene and select Scenes/GridNavigiation. These are the major components of the UI.



In the toolbar click the ‘play’ button. You should now be in the game view with a screen like the following:

You can click and drag the obstacles using the left mouse button. If you click on the plane, the Agent will attempt to navigate there via the graph. However, the limited placeholder code won't allow for much functionality until you complete implementation. You can press buttons 1,2,3,.. to select some preset configurations of obstacles.
Here is a screenshot of a working implementation:



Note that a properly populated grid will show boxes for navigable cells and X's for blocked cells. This rendering code is already provided and should work automatically for you.

Step 5: Navigate through the Project Window Assets and double-click on the CreateGrid.cs file. This should open up your IDE to edit the file. You have to now fill in the Create() method (and others mentioned previously) of this file such that:

1. It must create and return a 2D array of Booleans such that grid[column][row] indicates the traversability of the cell at (column, row).

Step 6: Test your implementation:
Click the play button at the top of your toolbar. Drag and move the obstacles around and try preset configuration. Click on a point on the grid to see that the sphere starts moving towards it by traversing the grid.

The numerical keys can select different presets.

Pathfinding search methods can be toggled with the Spacebar. Note that the Agent may not always be able to navigate to the goal if the Greedy-Simple search is utilized. Additionally, Dijkstra, Greedy Best-First, and A* won't work until you implement them in a future assignment.

You can visualize your path network by hitting "V". Nodes are indicated by white line intersections. The white lines themselves indicate graph edges. If you see a red edge then you don't have edges going in both directions.

You can peek under obstacles to see all your grid cells by either dragging an obstacle or toggling the "T" keypress.

<u>All interactions:</u>

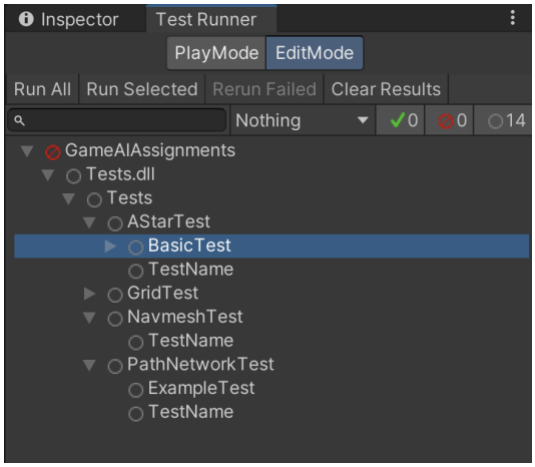
0-9 - choose different scenes
Tab - switch between grid, path network, and navmesh assignment scenes
Left click - path search to location
Right click - incremental path search to location
Shift Right click - Manual incremental search to location (see N below)
Left click drag - move obstacles around (or waypoints on path network scene)
N - Next manual incremental search step
T - Toggle to hide obstacle models
V - toggle to View graph adjacency on the Grid
. - (period) to toggle between 4-way (default) and 8-way connectivity
Spacebar - change search algorithm (see HUD top right)

Step 7: Automated Testing

It's *highly* recommended that you leverage the unit/integration testing feature as well! You should plan to spend considerable time writing your own unit/integration tests by extending Assets/Scripts/Tests/GridTest.cs. A couple example tests are provided with comments describing how to structure a basic test. The Unity tests are based on NUnit C# testing framework. These tests are in no way exhaustive and passing them should not be considered an indication of correctness (this applies to future assignments with provided tests as well). The example tests are only meant to serve as a starting point for writing your own tests. Your solutions are expected to do a good job of performing consistently and robustly, even with edge cases.

You can use Unity's build-in unit tester, called Test Runner, with this file. Access it from the Unity file Menu under Windows. You can then run your tests.

To run the tests, go to *Unity File Menu: Window->General->Test Runner.* The Test Runner looks like the following image. Select the test or test group you want to run and then choose “Run Selected”.



Grading

Your solution will be graded by autograder. The autograder will look for grid cells that intersect with various obstacles (arbitrary polygons that are convex, concave, etc.). The autograder will test your solution on several maps, some of which are provided as test maps in the presets.

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

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

Please do not submit anything with infinite loops.

Hints

Carefully consider all possible geometry situations you may encounter.

Debugging within the game engine can be hard. Print statements will be one possible way of figuring out what is going on (print() or Debug.Log()/Error()). Just make sure to remove the print statements afterwards! You can also use the Visual Studio debugger to debug your code if you’ve installed the Unity plugin for it.

Submission

To submit your solution, upload your modified CreateGrid.cs file **with correctly set student name string**. All work should be done within this file. Don’t forget to change the student name string!

You should not modify any other files in the game engine. You may modify the presets and unit test files, but don’t turn them in.

DO NOT upload the entire game engine.

Refer to Canvas assignment description for submission specifics, but most likely the assignment will be submitted to GradeScope.