INTRODUCTION

This is a Godot/GDScript project developed by O. Kaya under supervision of Alex Davies in 2023. It's meant to eventually be assembled into a drone control software to run on mobile phones and function similarly to an RTS game in terms of UX.

This documentation sheet will set forth all of the components and pieces that are there in the two existing scenes of the project, plus what's next, and extra notes. Additional, more specific minutiae are explained in the code itself, which has been cleaned up to remove most non-functional or deprecated logic, of which there was a lot during this process.

1. Pathing[DronePather.tscn]

1.1. **DroneMgr:** Manager script/object for spawning drones(default bound to middle mouse) and feeding them target positions(left mouse button) and then move orders.
This script programmatically determines an (adjustably) tight circle around the clicked point around which these positions will be, and sequentially tells the drones to path toward their points upon the completion of the last path.

1.2. **NavDrone:** This is the drone(representation) itself, when told to path by the manager, uses a navmesh(navigation mesh, sometimes referred to as navigation region in godot) to draw the shortest path there, as far as I know, with A*. Then covers that path with navmesh blocking cubes. Has custom move logic to follow that path. After the initial pathing, it'll trigger a navmesh rebake to cut its added obstacles into it, then trigger the manager to send the order to the next drone to path. Once all drones are done pathing, they're all given the order to clean up their obstacles and the navmesh is once again rebaked to prep for the next operation.
This style ensures no two drones' paths can ever intersect. We could try making drones live obstacles to each other(allowing more flexible pathing), but that'd make it potentially dangerous if the logic wasn't being run drone-side. This current system should(and seems to) make a collision impossible if the drone follows the given set of points, and reduced the total amount of data that's mission critical.

1.3. **Testbin:** This is the obstacle used by NavDrones to cover their paths for the navmesh. There's nothing complex here, it's just a cube. In the final version, these would be invisible to keep the UI sleek. Currently they also serve to mark out the path every drone will be taking, and could be colour coded to further show that clearly.

NOTES & TODO:

- In a finalized version, DroneMgr would not spawn drones but pull data as to where said drones are in real life and assign them to its internal list
- Ideally, DroneMgr wouldn't hand the circle points sequentially to the drones but instead minimize the total distance each drone travels by picking the most distance optimized set of points, or anything more optimized than the current fixed orientation and numbering points system
- The whole RTS thing implies we'd want behaviours beside 'circle one point', should be a simple thing to implement, but still worth noting. Similarly, it'd be nice to add the ability to select particular drones for a maneuver rather than all known ones at once.
- One issue with the quality of pathing is that if drone A and drone B haven't pathed yet and drone B is between A and its goal, if A paths first, B will just draw its path on top of A. This could be solved with smarter pathing iteration than just going through the set list of circle points and matching drone numbers in order, but a good thing to do would be to have a failsafe that if their paths do intersect, there'd be a reversal of pathing order between those to to sort it
- The movement and path iteration is done manually in navdrone as default godot will try to 'fix' the given path by redrawing it after rebake.
- The solution of rebaking the navmesh between every drone *does* work fast enough, around 2 seconds per drone to determine in the relatively large -though somewhat sparse- test scene. For a final version, the navmesh would likely need to be cut into different sections and be selectively rebaked to optimize performance, or dynamic obstacles would be used with the aforementioned downsides

2. 2D Mapping[cgtosm.tscn]

2.1. **Mapper:** This simple script takes a set of coordinates w/ zoom level up to 19 and using OSM's supplied formulas, finds the correct 2d map tile representing it, casts it onto a plane, and scales it to be 1 unit =1 meter

2.2. **MoveCam and crosshair:** Controls camera movement and rotation for a top down, 2d camera. The crosshair follows the camera exactly without capturing height, making for a constant 100x100 meter reticle on screen.

NOTES & TODO:

- We must integrate the current two scenes. It'd be very easy and of no use as of now, but better sooner than later.
- The crosshair randomly disappears based on position and zoom of the camera despite being in render distance and definitely above the map, this leads me to suspect it's float inaccuracy, but this is a non-issue as the crosshair is a temp measure to check scaling.
- The scaling is weirdly off for mapper, not by a significant margin, but enough to count. I may have been a little imprecise in the admittedly very compact math map I did, as geometric formulae isn't 100% accurate due to earth's irregular shape.

- 2D maps do not give us what we need to navigate real life space, and the 3d data is provided in a filterable JSON file format. Converting that into geometry is the path forward.
- The camera controls are set for 2D and top down, and probably would not work for 3d, certainly would not be intuitive, but a normal FPS camera with drag controls should fix that.

3. General Notes & Todo

3.1. An android port was attempted but due to android studio issues didn't really work on my phone. I just couldn't get the right api version loaded on my pc for a proper build.

3.2. UI is obviously PC focused right now and requires two buttons for testing in the pathing scene, and basically 6/a joystick for camera controls on the mapping scene.

3.3. Notes regarding exact functionality, methods, variables etc are in the scripts themselves.

3.4. 3D Maps, Integration of the two facets of the software, then connection to actual drones to send and receive data would be the core features missing, alongside the numerous other things mentioned above.

3.5. I tried to use all pre-existing solutions for loading 3d OSM data into godot as of November 2023 and found them to be all some combination of:

3.5.1. Non-functional

3.5.2. Non-complete

3.5.3. Non-documented

3.5.4. Non-applicable

3.6. Hence, It's suggested future programmers and engineers who may land on this project do not waste their time attempting to integrate those.

3.7. While documenting this I did find this writeup on getting 3d visualisation going in Unity that seems would be useful and relatively easy to convert to godot with the work I've already done on the project as a basis:
http://barankahyaoglu.com/dev/openstreetmap-in-unity3d/
While Geodot *may* be useful for actually pulling in the terrain shape, for less-urban environments where that is a significant factor.