Metapopulation dynamics for ecological simulations

Isaac Manzi-Rickaby

Aston University, Birmingham, United Kingdom

ABSTRACT

Metapopulation dynamics play an important role in the continuation of a species survival. This software package aims to provide an easy way for an entry level ecological audience to visualize this and see the potential impact of human interference across a whole metapopulation.

CONTENTS

1.0   Introduction

Metapopulation dynamics can found in many fields, including genetics and ecology; a metapopulation is a group of separate populations of a singular species which have some sort of influence on another, usually in ecology this will relate to animals in a habitat competing for resources or territory. Metapopulation dynamics can also be used to see how species behave within a habitat which has external effectors.

With the current global concerns increasing over our collective influence on the climate and its effects on wildlife, the potential for interest in some of the underlying dynamics is also greater. However, metapopulations are quite daunting to someone with no knowledge in the ecological or field.

This software package aims to provide an intuitive interface to create simulations of a habitat, the species and resources in the habitat, which requires little to no research to use. The target audience is people little ecological or technological knowledge, but an interest in the area. And aims to show how different effectors on a population can impact the metapopulation. The software package also aims to provide a high level of abstraction, so users can model a variety of potential scenarios.

2.0   Background Research

2.1. ***Frederico Mestre, University of Évora*** *[1]*: created an R package (MetaLandSim) created "to simulate metapopulational dynamics on a habitat network. It also computes a lot of landscape connectivity metrics and simulates range expansion"[2]. This package is very detailed and powerful, there is also a related research paper which was very important when developing my own solution. However, this package is also quite complex and difficult to operate without a background in both ecology and the R programming language, which would be a hindrance to my target audience.

2.2. ***Amrita Vishwa Vidyapeetham, Metapopulation dynamics simulation*** *[3]*: probably the most accessible, yet detailed solution available. Has customisable options for the simulation and the ability to change the metapopulation model. This solution is provided by an educational website and has a lot of information on the solution and the dynamics themselves. However, it is not very clear to see what is going on in the simulation, and while there is more user input, it is strictly about the model and doesn't provide much else for the user to change.

2.3. **Unnamed web simulation** [4]: this software package is very accessible to users with little knowledge of ecology or any programming, which is great for my target audience; however, there is very little information on the origin of the simulation, and while the simulation is accessible and visualizes metapopulation dynamics along with background information on the topic, it is a very constrained system with little user control or input.

3.0   Materials and Methods

3.1.   **Unity Engine** [5]: is a cross-platform game engine, while its main use is for game development, it is not limited to this. It has great tools for simulation and 3D or 2D graphics. Using Unity Engine means that there is no need to develop all the supporting tools for the simulation such as rendering and a physics engine to manipulate the graphics for the simulation. And has extensive support and documentation.

3.2.   **Json.NET** [6]: is a JSON utility library for unity, with better support for serialization operations of objects in C# than what is offered by the core Unity Engine libraries. This is used in the saving and loading of data to JSON files, which is the ideal solution for allowing users to create custom data for the simulation which can be shared and used by other users.

3.3.   **Kenny.nl interface-sounds** [7]: is a creative commons licenced audio pack created by Kenny.nl for the public to use in projects, there are many different packs available for a range of uses, they are provided in .ogg format, which is supported by the Unity engine.

3.4.   **Blender** [8]: Blender is a free software package for 3D modelling with many export options including .fbx which is supported by Unity engine. This software package was used to create the tile prefab model.

4.0   Requirement Analysis

The software packages simulation is visual and interactive by allow users to control the simulation before and during operation. It is usable without any prior knowledge of any programming or ecological sciences.

When creating assets for the simulation the same should be said for having no background and should be intuitive and easy to use. Assets should be serialized to portable files and be possible to be shared between users or created in external programs.

The simulation is composed of a grid of interactable squares which represent a patch of land. This patch of land can contain resources and species. As time progresses within the simulation through tick increments the resources should increase based on a defined growth rate and be consumed in accordance with the population of species and the defined requirements for their survival.

Species instances should die off according to their age, failed migrations and starvation. Species instances should spawn new instances, and or migrate based user defined values. Creating new species and resources for the simulations are possible from the menus. The creation menus are intuitive, and resources can be created from the species resource requirements section without losing the information entered in the species creation menu.

5.0   Architectural Design

The architectural design of the software package is the rough outline of how the requirements can be achieved through the given materials and resources, this design will be passed to the next phase of the development cycle where the design is implemented.

5.1. Menu System Design

The menu systems provide the users with the ability to create new simulations, load simulations and the various resources and species necessary for these simulations. The main menu system is designed so that navigation between the various pages is simple.

Figure 1 details a map of the potential routes between pages within the main menu system. And what pages will need to be developed.

Unity has simple to use user interface options, where objects can be dragged onto a canvas and positioned and anchored to work on a range of devices and screen orientations through positioning of these UI elements and their settings.

There are various buttons on the home screen which direct to different pages including the 'new simulation' and 'load simulation' pages.

Figure 1

These pages are a collection of UI elements parented to a GameObject. These GameObjects can be toggled active or not at runtime through code. Using button presses to call relevant functions within the MenuController.cs and AssetCreationManager.cs scripts to operate the menu system and the asset creation system respectively.

The 'new animal, 'new resource', and 'new simulation' are pages which are used by users to create assets and interact with the AssetCreationManager.cs and FileUtility.cs scripts which handle the messy user data and the data-serialization of created assets respectively. The 'animal list', and 'resource list' also interact with the FileUtility.cs script in order to deserialize the JSON files into workable data types to reconstruct the instances of species, simulation and resource assets. All user inputs are parsed from string inputs, so data is validated after parsing before storage.

The 'animal list', 'resource list' and 'new simulation' pages use the Unity Engines grid components to populate with the currently loaded custom assets found in the /customassets/* directories into selectable button presses, returning the chosen elements asset to the previous page. This is used for adding Resource and Species assets into a new simulations' arguments, and for adding resource assets to a new species resource requirements.

As the simulation and menu systems should be separated into different Unity scenes; to carry data from the menu scene into the simulation scene, a persistent object is required. This persistent object should also hold the sound effects and logic for playing these sounds, as sound effects are used in both, the menu and the simulation scenes.
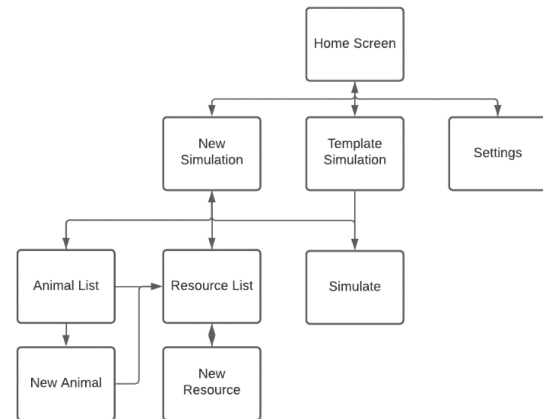
5.2. Simulation System Design

The simulation takes in a list of Resource assets, a list of Species assets, and the simulation dimensions provided by the user as the initial starting arguments and are the arguments used to setup the simulation environment. The dimensions are used to create a multidimensional array containing individual cells which use the resource and species assets properties provided to calculate initial starting values.

Users can advance time in the simulation by using the user-interface; advancing time of the simulation will loop through the array of cells and process the changes that have occurred within each cell, such as resource growth, resource consumption, population growth and falls, and species migration.

The state of each cell can be determined by the carrying capacity (K) of the cell for each species (S) based on the resource population (RP) and the species resource requirements (key) and the count of the species population (C).

$$K = \frac{RP[key]}{S[key] \cdot C}$$

If the carrying capacity is higher than the current population then the current Simulation Cell will just remove the resource requirements, and the Cell State will be set to habitable (or occupied if the population is greater than 0), otherwise, the simulation will need to starve some of the effected species and remove them from the population.

When species migrate, cells with the state occupied have a higher priority than cells with the state habitable and will always give birth on arrival at the chosen destination; unless there is no habitable or occupied cell in the species migration range, in which case the instance will be removed from the simulation and be considered a dead or lost species instance.

Migrations create an indication for the user to visualise where the species has come from and where it is going. Instantiating a capsule to represent a migration and translating its position over a set period from the birth cell position to the migration target cell position.

During the setup phase of the simulation, each cell will populate itself based on the values from the species assets and the resource assets, the resource assets have a spawn probability and an initial range for the quantity to be spawned if applicable to the cell; and the species assets have a value for the initial coverage percentage, which applies to cells which are calculated to be habitable after the initial resource dispersal, the initial amount of a species added to a cell is a random range from 1 to the max child count associated with that species.

6.0   Software Development
6.1.   Menu System Development
The menu system is simple in operation and development. Within the unity scene hierarchy, creating an empty *GameObject* which will work as a page within the menu, and having a collection of UI elements parented to this *GameObject*, means if the object is active, then all the child objects will also be active. Having multiple of these page *GameObjects* and toggling their active state through button presses creates a menu system and its navigation.

Using an *ENUM* to store the current and previous pages, as well as the content route provides all the information needed to navigate these pages; using a simple switch case on the current page to deactivate all pages, and active the current page is used for navigation, previous page is used to go back a page, and content route is used to add selected grid entries to the corresponding page; for example, *ResourceAsset.cs* instances can be added to both *'new simulation'* and *'new species'* pages.

For any page that has user selections such as *'load simulation'*, *'resource list'* and *'species list'* pages, there is a scrollable grid UI element which each selectable entry is entered. These grids rely on the *FileUtility.cs* script, which provides access to the user created assets. All these grids work in similar ways; however, they display different amounts of information to the user; when on the *'resource list'* page, there will be a populated grid of all the found *ResourceAssets.cs* instances, with a colour splash and the name of the resource. This is done by fetching the list of resource assets from the *FileUtility.cs* script, and for each element of the list a grid entry prefab will be instantiated; each grid entry will have a hook script (in this case *ResourceListHook.cs*). Using dependency injection, this script takes the *ResourceAsset.cs* instance and uses the values to detail the prefabs name and colour.

Grids are scrollable due to the content size filter component which changes the size of the panel containing the entries based on the content; using the minimum vertical size, the grid will only enlarge when more elements are added.

UI elements within unity have a position, size and anchoring type; these are used to position and anchor the elements within the screen so that resolution changes do not move objects into wrong areas of screen space.

Another *GameObject* which is not really related to the menu system but is found in the menu scene is the *PersistentSceneManager.cs*. The loading of a new scene will clear all current GameObjects from memory and load in the new scenes *GameObjects*. This causes issues as when the simulation scene is loaded, we still need the values defined by the user on the simulation settings. *PersistentSceneManager.cs* is a singleton, and in the *Start()* function, the *DontDestroyOnLoad(this)* function is called. This is signalling to Unity that this *GameObject* should not be destroyed on scene changes. This *GameObject* and script is used to carry over any necessary data and functionality such as simulation environment settings and audio control. For audio control, there are four effects that can be triggered by calling a function, *ConfirmEffect()*, *BackEffect(), LaunchEffect()*, and *MigrateEffect()*; these functions call the current Audio Source to play the sound effect.

6.2.   Asset Creation System Development

Unity has many ways to take user input, it can all be done through Unity's Input class; however, this is not ideal when taking user input in the form of text inputs rather than gaming inputs such as single button presses.

Using Unity's UI components to build forms to create user assets isn't easy either. Unity does not provide tools to change *InputFields* acceptable data types, therefore all *InputFields* work with string types. This causes problems when wanting to only accept integers or floats.

Figure 2

Figure 2 displays the 'new species' page, and the form created using the UI elements provided by Unity, which has InputFields that expect floats and integer values and not string.

To accept only a certain datatype this software uses the *TryParse()* method to return a true Boolean value if successful. If successful, then the entered value is stored within AssetCreationManager.cs, if an asset type has all potential values stored, then it is saveable an the save asset button is set to interactable; otherwise, the button is not interactable and users cannot save the created asset with missing information.

Created assets which are saveable, are saved by passing the asset variable to the FileUtility.cs scripts corresponding function, for example; a created *ResourceAsset.cs* instance is passed to the *SaveResource(ResourceAsset asset)* function when the save button is pressed. The purpose of this function is to serialize the variable to a JSON format string using *json.NET*, these JSON strings are then stored as JSON files in the corresponding directory nested in the */AppData/LocalLow/Aston University/Metapopulation Simulator/CustomAssets/* directory for the given type. If these directories do not exist, then they are created at the first run time.

Demo assets are created on the first execution of the software package. The length of the *FileUtility.cs* asset lists is used to check if the program has been executed before. If the length of the lists is 0, then the demo assets are created and saved. This is needed as users may want a way to see how the software runs before modelling their own assets and simulations.

6.3. Simulation Development

When the simulation scene is loaded through the Unity Engines Scene Manager, the *PersistentSceneManager.cs* will hold all the simulation environment values which are needed to set up the simulation before execution.

The initial starting arguments are the simulation dimensions, and the lists of assets present in the simulation. For the list of *SpeciesAsset.cs* instances, they need to be converted to *SpeciesAgent.cs* instances which contain the functionality as well as the values; this is not necessary for *ResourceAsset.cs* instances, as they have no functionality. The simulation dimensions are used to create the visual array and the simulation array, and the lookup dictionary. The *visualArray* holds the instances of the cell tile prefab, which is visible and interactable to the user, whereas the *simulationArray* holds the *SimulationCell.cs* instances, which are not visible to the user as this is the functionality and value of each cell. The lookup dictionary is used when the software needs to find the *SimulationCell.cs* instance which corresponds to the given *visalArray* instance, this is used in the *CellDetailManager.cs*.

To setup the simulation, a nested for loop iterates over the simulation dimensions (iterates x and z values due to Unity engines y value representing the vertical axis, and not depth), and for each iteration the *SimulationManager.cs* calculates the new position for the *visualArray* entry using the current x and z values multiplied by (*cellTileSize* + *cellTileborder*) values, which gives a small gap between each visual cell, this instance of the prefab is then added to the *visualArray* at position [*x,z*]; during the same iteration, a new *SimulationCell.cs* instance is created, and dependencies are injected, this created instance is added to the *simulationArray* at

Figure 3

position [*x,z*] and the *cellLookupDictionary* adds the key of the instantiated prefab, and a value of *SimulationCell.cs* instance, the visuals are then updated for the current iterated cell. Once the visual array is all instantiated, the camera needs to be centred; this is achieved by halving each of the simulations dimensions and multiplying this by the cell tile size add to the cell border size to create new x and z positions for the camera. Figure 1 shows an example simulation once the setup is complete.

When creating a new *SimulationCell.cs* instance, through dependency injection, the list of resources and species for the simulation, the current coordinates and the simulation dimensions. For every resource in the possible resource list, using the probability values of the asset, it will either be added to the resource population of the cell or not; this is calculated by generating a random float between 0 and 1, if the value is greater than 1 – assets spawn probability, then the *resourcePopulation* dictionary has the resource assets id added as a key, and a random value between minimum and maximum quantity is added as the corresponding value.

After looping through the resource list, the species list is iterated to check if the current *resourcePopulations* carrying capacity is greater than 1, meaning the cell is habitable. If so, then the *SpeciesAgent.cs* instances name is added to the *speciesPopulation* dictionary as the key, and a new list of instances is created for the *speciesPopulation* as the value. For each key in the *speciesPopulation* the initial spawn probability is checked the same way as resources on whether the cell should be occupied by the current species, a random value between 0 and 1 is checked against the *spawnProbability*. If so, then a random value between species' minimum child count and maximum child count is generated, this is used as an iteration range to create the right number of *SpeciesAgent.cs* instances for the given species key, each *SpeciesAgent.cs* instance created is added to the list corresponding to the *speciesPopulation* keys instance list. When instantiating a new instance of *SpeciesAgent.cs*, the *Init()* function is used to pass data used to setup each agent. Such as the probability of migrating, carrying a child and the other values, the age is set to 0 as it is a new instance.

The *SimulationTick()* function found in *SimulationManager.cs* is used to increment the tick value, which represents time flow in within the simulation. During the *SimulationTick()* execution, the *simulationArray* is iterated over and each *SimulationCell.cs* has the *CellTick()* function execute; this returns a dictionary, keys being *SpeciesAgent.cs* instances which are due for migration, and the values are lists of possible destination coordinates. The *totalPopulation* dictionary is also reset at the start of the *SimulationTick()*, and for each *SimulationCell* iterated, the species populations are counted and added to the dictionaries corresponding value; once the *SimulationCell.cs* instance has had the *CellTick()* function executed, and population counted, the *visualArray* entry is updated based on the *SimulationCell.cs* instances cellState value.

The *CellTick()* function is used to update each individual *SimulationCell* instance. Calculating the resource consumption for the current tick is done by looping through the *speciesPopulation* dictionary, and if the given list of species' instances is greater than 0, the species' demand is calculated; for each *resourceRequirement* entry for the given species, the required value is multiplied by the count in the *speciesPopulation* list for the given species, this demand is added to the *resourceConsumption* dictionary. The current *resourcePopulation* dictionary is iterated and growth is calculated by multiplying the current quantity by the growth rate of the resource asset and adding to the current quantity; then the corresponding *resourceConsumption* value is subtracted from the *resourcePopulation*. If the new quantity after subtracting the *resourceConsumption* value is less than 1, then the affected species should start to die off due to starvation as they are not receiving their required resources. Based on the calculated carrying capacity of the resource, it then calculates the number of instances of the affected species' that should be killed off; removing instances is done by generating a random number between 0 and the *speciesPopulation* list count and issuing the generated number as the index to remove an element from the *speciesPopulation* list.

During *CellTick()* execution, the *speciesPopulation* keys are iterated, and then the corresponding instance list is also iterated; for each iteration, the species instance has the *Tick()* function executed which increments the instances age. The current age of the instance is checked against the *migrateAge*, and *maxAge* and *birthAge* and processed accordingly. If older or equal to the *maxAge*, then the instance is added to the list of instances to be removed; if older or equal to *migrateAge*, and the instance *willMigrate*, and not already migrated, then it is added to the list of instances to migrate; if older or equal to *birthAge*, and is *carryingChild*, then generate a random number between species minimum child count and maximum child count, and create that amount of new instances of the given species. Once all these steps are completed, the *SimulationCell.cs* instances *cellState* is updated using the *UpdateState()* function, it checks if the current *resourcePopulation* has the carrying capacity for any species in the simulation, and if it can support at least one instance of any species, it is set to habitable, if the *speciesPopulation* for any species is greater than 0, then the *cellState* is set to occupied; otherwise if it cannot support any species instances, then it is set to clear.

A dictionary is created during *CellTick()* which contains the instances that will migrate, and the possible destinations related to the current cell and the maximum migration range. This dictionary is returned to the *SimulationManager.cs* instances *MigrateAgents()* function; this function checks all the potential destinations and checks if there is any currently habitable or occupied *SimulationCell.cs* instances, if there is an occupied cell, this is prioritised over habitable cells; however, if there are no habitable or occupied cells in the potential destinations list, then the instance is killed, otherwise the instance is passed onto the *SimulationCell.cs* instances *AcceptMigration()* function, which simply sets the migrated Boolean variable to true, so it doesn't re-migrate, and adds to the *speciesPopulation* dictionary accordingly.

The visuals of the migration are created by instantiating the migration prefab, which is a GameObject containing a capsule, the hook script takes in the current cell coordinates, and the destination coordinates and translates the capsule from the starting coordinates to the destination over a period.

6.4. User-Interface Development

The user-interface for the simulation scene was developed in a very similar way to the menu system. There is a simple bar along the bottom of the screen which holds the current tick increment value, and the tick button, which calls *SimulationManager.cs* instances *SimulationTick()* function. There is also a button to open or close the cell detail panel; this is the panel which shows the current cells details such as the currently selected cells coordinates, the resource population and species population, the cell detail panel also has a button to open or close the cell options panel.

The cell detail panel can be opened with the button on the bottom-bars far right. If no cell is currently selected, then the cell detail panel will be blank, and have 'x, y' as the current coordinates. A cell is selected by clicking on the tile prefab on screen, this is completed by using the Unity engines Input class to check for mouse button presses, and casting a ray from the camera, through the mouse position onto the scene's world position, if the ray collides with a collider, we can access the clicked *GameObject*. If the tag of the selected *GameObject* is "Cell", then we can use this *GameObject* in *SimulationManager.cs* instances *cellLookupDictionary* to find the *SimulationCell.cs* instance relating to the selected *GameObject*.

The cell detail panel uses the grid system found in the menu system in the menu scene; instantiating a grid entry prefab and customizing the visuals of the entry through a hook script attached to the instance. In this
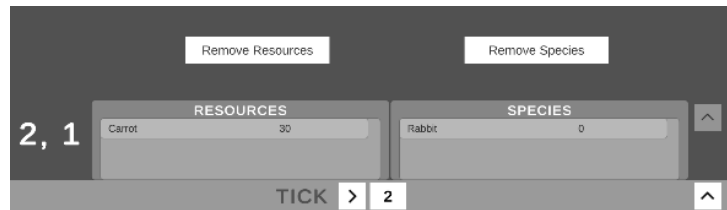

Figure 4

case, the name is updated to the resource or species name, and the count value is updated to the quantity of the instance in the selected cells *SimulationCell.cs* lists. The cell options panel has a button to clear the currently selected *SimulationCell.cs* instances current species, or resources.

Users can also interact with the simulation by navigating the region using the camera. The *CameraController.cs* is attached to the main camera in the simulation scene, this script uses Unity engines Input class to map arrow key presses to camera position translations, and mouse scroll-wheel movement to camera height from the simulation.

Figure 4 displays the cell detail panel and the cell options panel.

7.0   Testing

Once the software packages development is complete, testing is required to make sure all the requirements formulated in the requirement analysis are achieved, and that all areas of execution perform as expected.

7.1.   User Input and Data Validation

Users interacting with the software through the user interface will need the relevant UI elements to be interactable, and the Unity Input system to be configured correctly for their system.

Unity has options to install the new Input System released for Unity versions 2019.1+ [9] is more efficient for multiple platform development, however as this software package is not designed for mobile or console deployment, this is not necessary. The built in Input system is sufficient as only mouse clicks, and arrow keys are required. The new input system is designed better for console and mobile development as multiple device inputs can be mapped to singular functions in the editor and requires no additional code to set up.

UI elements which are interactable such as Buttons and *InputFields* all need to be tested as the hierarchical structure of these elements decide on the 3D layering of these elements. If a *TextMeshPro* UI element is above a Button UI element, the user may click on the Button element but the bounding box of the *TextMeshPro* UI will block interaction with the desired element.

As discussed earlier in … all the *InputFields* return only string types, and this cannot be limited within Unity; Therefore, all the string values need to validate once parsed into the expected datatype. For example, when entering the 'maximum age' of a species in the 'new species' page of the menu scene, a user can enter "fifteen" which is not the expected datatype, this will cause issues when attempting to simulate this species; using the *TryParse()* function, the software can validate that the parsing of the data is successful and only allow the user to save the asset once a value has been deduced.

7.2.   Cell State and Migrations

Some assumptions made in order to simplify the problems of simulating a metapopulation also cause some small issues. The current solution to visualising the cell state, is to colour the visual cells prefab based on whether it is clear, habitable or occupied; this is applied based on the *cellState* of the *SimulationCell.cs* instance. However, this causes some issues when simulating multiple species. A cell will be marked habitable if it is habitable to any of the species present in the current simulation, which can result in unexpected behaviours; migrating instances of a species will prioritise cells in range which are occupied, or habitable; however, a cell may be tagged habitable based on a different species resource requirements, and not be habitable to the currently migrating species, this can result in a species instance migrating to a cell which is not a habitable environment.

This could be fixed by creating a dictionary containing the species as the key, and the *cellState* as the value, and checking accordingly. It is also slightly unclear to the user which cells are habitable to which species, as there is a single colour for the habitable status, however this seems to be the best current solution for visualising the cell state.

Another assumption is that a migrating species instance should give birth upon arrival, so that the cell is not occupied for a single instance's lifetime. This can cause issues, as this should only be done if it is not occupied already. If a cell only has one occupied cell in migration range, it is possible that all migrations will target the occupied cell and give birth each migration causing an unrealistic population boom in the targeted cell, this can have ripple effects within the cell, including resource consumption increases. This is fixed by checking the targeted locations population count for the same species and setting the birth data accordingly.

8.0   Deployment

Deploying the software package had two stages, building the Unity project for the targeted platforms, and pushing the Unity files and build files to the GitHub repository.

The Unity project build creates a compiled version of the project along with a launcher; this is a portable piece of software that can be shared easily through these build files; however, this provides its own issues where the developmental editor version operates differently to the compiled build version.

When using the asset creation options within the build, none of the assets where loading or saving. This was due to the API compatibility settings within unity being for .NET 2.X and not .NET 4.X, which was required for json.NET from newtonsoft.com, Also changing Appliction.datapath, to Application.persistantdatapath within FileUtility.cs fixed this problem, and assets can be loaded and saved during build version runtime. The custom user assets can be found at the Application.persistentdatapath for the current operating system, in the case of windows this will be C:/Users/~/AppData/LocalLow/Aston University/Metapopulation Simulator/.

The build version also allows for Fullscreen operation of the software, which changes the resolution. These resolution changes didn't work well with the first build, causing UI Elements to be positioned wrong and overlap. Unity has simple options to change without having to change any code to fix these, within the project settings in Unity you can change the software to run in windowed mode and change the resolution. Changing the anchoring settings of some of the UI elements also fixed their positioning with different resolutions.

The completed Unity project files, including build files are available to download from GitHub [10] and should be opened using Unity version 2021.1.21f [11].

9.0   Improvements
While this software package achieves all the requirements and operates as expected, it can still be improved upon for different use cases.

❖ Art Assets

A more complete and varied set of Tile meshes could improve the visuals of the software package. Having an artist create a set of nature, and industrial tiles to represent occupied, habitable cells, and clear cells respectively, would better display what the coloured tiles represent.

❖ Resource Spread

Currently the resources are spread using a very simple random placement algorithm which does what is intended; however, using a better method to more realistically spread resources around the simulated region could be implemented, such as poisson disc sampling or a Perlin noise function.

❖ Species behaviour and Asset Design

Species resource requirements are ids of resource assets; however, it would have been possible to design it so the *SpeciesAsset.cs* extends the *ResourceAsset.cs* so that other species can also be required resources to develop predatory species. Extended behaviours could also be developed for species to further increase species interaction within the simulation, such as species which migrate as a group.

10.0   Evaluation
Researching this field and developing a solution for simulating metapopulation dynamics has been very enjoyable and educational in both computer science and ecological fields. While working with strict time constraints the software package has been completed to achieve all the requirements devised in section 4 to a high level.

The users can create their own assets within the software package, and the created assets can be shared with other users, so long as they are placed into the correct system folder and have a unique entry in the id field.

The simulations are vague and allow users to develop their own complex simulations through the modelling of these assets, and users can interact with the simulation through individual cells to imitate human interaction in a geographical region.

The software package is user friendly and easy to operate without any background knowledge of computer science or ecology; it provides a great entry point for learning about metapopulations and how human interaction can impact the dynamics at play.

As the software is created in Unity, the package is available to build for many platforms and devices, and a range of screen sizes and resolutions.

11.0   References

[1] MetaLandSim, (Frederico Mestre, Fernando Cánovas, Ricardo Pita, António Mira, Pedro Beja, 2016) MetaLandSim is an R package to simulate metapopulations. Last used 22/04/2022, available at https://CRAN.R-project.org/package=MetaLandSim

[2] Frederico Mestre (2019); Creating landscapes and simulating species occupation with MetaLandSim. Last used 22/04/2022, available at https://www.r-bloggers.com/2019/02/creating-landscapes-and-simulating-species-occupation-with-metalandsim/

[3] vlab.amrita.edu, (2011) Metapopulation Dynamics -Levins Model. Last used 22/04/2022, available at https://vlab.amrita.edu/?sub=3&brch=65&sim=772&cnt=2

[4] whfreeman.com, (unknown author, unknown developer, unknown creation date) Unnamed Metapopulation simulator. Last used 22/04/2022, available at https://whfreeman.com/BrainHoney/Resource/6716/SitebuilderUploads/Hillis2e/Student%20Resources/Animated%20Tutorials/pol2e_at_4203_Metapopulation_Simulation/pol2e_at_4203_Metapopulation_Simulation.html

[5] Unity Engine (Unity, 2022). Free game engine. Last used 22/04/2022, available at https://unity.com/

[6] Json.NET (newtonsoft.com, 2022). JSON Serialising utility for C#. Last used 22/04/2022, available at https://www.newtonsoft.com/json

[7] Kenny.nl interface-audio (@KennyNL, 2010): free creative commons audio pack for interfaces. Last used 22/04/2022 Available at https://kenny.nl

[8] Blender (blender.org 2022): free 3D modelling software. Last used 22/04/2022, available at https://www.blender.org/

[9] Unity New Input System (Unity, 2019): a new input system for unity projects in unity version 2019.1+. Last used 22/04/2022, available at https://docs.unity3d.com/Packages/com.unity.inputsystem@1.0/manual/Installation.html

[10]   Unity Project Files (Isaac Manzi-Rickaby, 2022): The software package development files discussed in this report. Last used 25/04/2022, available at https://github.com/manzi-labs/Metapopulation-Simulator

[11]   Unity Version 2021.1.21f (Unity, 2021): The version of the Unity engine used for the development of this project. Last used 25/04/2022, available at https://unity3d.com/get-unity/download/archive