Dynamic scene modeling in agent-based survival simulation

by

Riley Wall

A thesis submitted to
The Faculty of Graduate Studies of
The University of Manitoba
in partial fulfillment of the requirements
of the degree of

Master of Science

Department of Computer Science
The University of Manitoba
Winnipeg, Manitoba, Canada
December 2024

Thesis advisor

Dr. Parimala Thulasiraman

Author

Riley Wall

Dynamic scene modeling in agent-based survival simulation

Abstract

Agent-based models (ABMs) simulate agents and the interactions between agents and their environments. We focus on the coevolution of prey and predator ABM. This thesis proposes to train agent models for increased accuracy in scene modeling by selecting agents capable of recognizing and modeling the relative locations of landmarks in their simulated environment to test and demonstrate the feasibility of this method. There are, therefore, two goals to this thesis. First, to develop a simulation-based technique for training AI agents in proficiency of scene modeling using population-level "survivability" metrics. Since scene modeling is viewed as a competitive advantage in terms of survival in nature, the research investigates whether a survival simulation is a sufficient motivation for selecting agents capable of accurately modeling their surroundings. Through experiments, we demonstrate the effectiveness or ineffectiveness of this ABM-inspired survival simulation as a viable training method for simulating complex behaviors. Second, we show how procedural generation techniques can extend existing ABMs into the third dimension and allow the rendering of this environment from several unique perspectives. This extension opens the door between simulated robotics and the large base of available ABM models, allowing new methods for comparing robot behaviors in highly dynamic environments.

Contents

List of Figures

Acknowledgments

This thesis is dedicated to my parents Randall and Allison Wall. To Parimala Thulasiraman, their guidance, support and insight has been essential in completing this undertaking. And to Izzy Nuessler for their love and support. Without the help and support of all involved, this project would not have been made possible.

Dedication

Chapter 1

Introduction

Mobile robots need an internal model to navigate their environment. While navigating across a room may be trivial for people, this task is much more difficult for mobile robots which must account for variances such as wheel slippage. Wheel slippage is an important problem in mobile robots as wheel slippage may lead to accidents in inclement weather or poor road conditions. A mobile robot's position maybe calculated using a dead reckoning navigation system. Dead reckoning is calculated by estimating the position of the distance traveled from it's current position. This estimation may increase over time and distance [1]. Therefore, a robot needs an internal world model to compare its observations to combat the effect of dead reckoning. The creation of the internal model is known as *scene modeling*. The joint process of building an internal model while navigating an environment is a problem known as Simultaneous Localization And Mapping (SLAM). The SLAM problem can be formulated as a series of decisions the robots need to make at each time interval. The robot acquires new data and updates its estimates of both the environment and

its place within it[2, 3, 1, 4]. This scene modeling and localization combination is critical for mobile robots [4, 1, 3].

Agent-based models (ABMs) simulate agents and the interactions between agents and their environments [5, 6]. The agents make decisions similar to mobile robots. Therefore, we can map mobile robots to agents in a simulated agent environment. The agents also simulate time intervals in discrete units. The agent models update the world and the agents present at each time interval [5, 7]. ABMs describe various simulative models, from simulating forest fire spreading to using foot traffic flow models for shopping centers[8, 9, 10, 11]. This research focuses on a specific type of ABM commonly referred to as *predator-prey models*. The predator-prey model consists of two types of agents (prey and predator) competing for resources[12, 13, 14]. The prey-type agents consume energy from grass that regrows over time. Predator-type agents, on the other hand, can only acquire their necessary energy through hunting and eating their prey. These models are commonly used in wildlife conservation applications to show trends between animal species populations [15].

Predator-prey models can also include agents with an internal model of the agent's immediate area or neighborhood. These agents use the often-given information to make more informed decisions on where to move to in the next time interval. In most cases, these agents are given information about the environment and can assume it is accurate [12]. However, in cases where an agent is given incomplete or potentially inaccurate information, the model should account for the missing information (data) and its potential errors. This task then becomes a technical problem quite similar to the SLAM problem that acquires new data and updates its estimates of both the

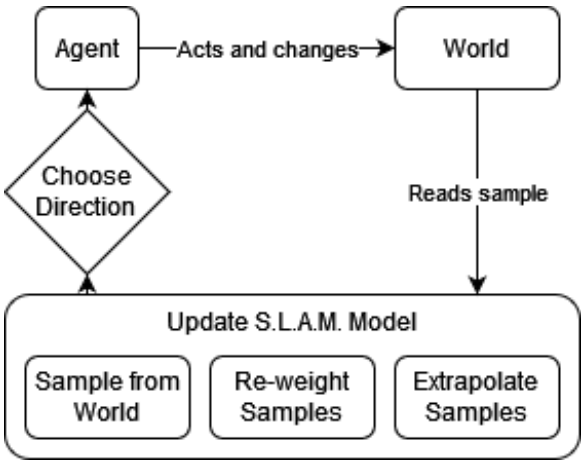

Figure 1.1: Agent-World interaction cycle

environment and its place within it.

As this group of ABMs tends to use a 2D grid, these models have height variance across the simulated ground of the world [5, 12, 15]. Adding a third axis to these models directly leads to massively increased computation time as the complexity of the grid size increases from n^2 to n^3. However, several procedural generation techniques can efficiently convert 2D grids to 3D renderable environments. For example, height maps can be used to generate terrain with height variances, and Perlin noise functions can be used to create height maps with large-scale and fine-scale surface features. The extra complexity in these environments leads to more complex integrations between agents. For instance, agents can be wholly or partially hidden by sections of the terrain.

We can view the ABM agents (prey and predator) and the mobile robot's interactions with their environment as a loop (cycle) shown in Figure 1.1. In the figure, the agents are denoted as *Agent*, and the environment as the *World*. The agents some observations of their environment. This data is provided to the SLAM model. The

model then compares the agent's current estimate of the environment (denoted as *Sample from World*). The estimate is then updated to include this new information (denoted as *Reweight Samples*). Finally, the model is used to extrapolate (denoted as *Extrapolate Samples*) to the next time interval and choose the action that will yield the most favorable outcome.

1.1 Goals

This thesis has two end goals. First, to develop an agent-based technique simulation for training artificial intelligent agents in proficiency of scene modeling using population-level "survivability" metrics. Since scene modeling is viewed as a competitive advantage in terms of survival in nature, I seek to investigate whether a survival simulation is a sufficient motivation for selecting agents capable of accurately modeling their surroundings. I plan to demonstrate the effectiveness or ineffectiveness of this ABM-inspired survival simulation as a viable training method for simulating complex behaviors. Second, to show how these procedural generation techniques can extend existing ABMs into the third dimension and allow the rendering of this environment from several unique perspectives. This extension opens the door between simulated robotics and the large base of available ABM models, allowing new methods for comparing robot behaviors in extremely dynamic environments.

This thesis will be broken down into the following sections. First, I will discuss the use of simulation-based approaches for robot testing. I will discuss the agent-based modeling techniques used in my experiment and their general use cases in current literature. Next, I will describe the problem of scene modeling and Simultaneous

Localization And Mapping, also referred to as SLAM. I will also describe how SLAM can be re-contextualized within the context of an agent-based model. Once I have provided some context to both SLAM and ABMs, I will discuss the ABM used in the experimental portion of this thesis. This will include a description of the predator-prey model to be used in the experiment and the various extensions to the model required to render the environment with a 3D representation. Then, I will describe the SLAM models that will be implemented in the ABM's agent representations. In the following chapter, I will describe the experimental setup, including the stop conditions and initial conditions for each trial. I will then present some of the observations noted during the simulated trials and what I believe these trends mean within the context of the ABM's population stability. I will then present my final notes on the matter, including where I believe this research can continue and what might be done differently in future studies.

Chapter 2

Literature Review and Research Gap

This chapter provides some background to simulation, scene modeling, and agent modeling and discusses the research gap in these areas.

2.1 Simulation Based approaches

Several attempts have been made in the past to use simulation to train robots. Simulation-centered approaches have been proposed to train perception models for scene modeling and object recognition [16, 17]. These perception models often use images of object-heavy environments and task the model with finding the configuration of objects from the image. This configuration of objects, positions, and rotations can be projected back into a simulated scene approximation. Scene modeling is an ongoing area of research in AI, augmented reality (AR), and robotics. Similarly,

simulations have been used to train more efficient walking gaits [18, 19]. Both evolutionary algorithms and traditional machine learning models have seen applications in generating energy-efficient walking gaits for bipedal and quadrupedal robots. Simulation has been leveraged with graph-based neural networks to model the physics and dynamics of 3D bodies within a simulated environment [20]. While very similar to scene modeling, physics modeling focuses primarily on the ability to use a representation of objects and predict future states of the scene. This is useful for planning locomotion capable of graph-structured body plans and how different physical bodies react when colliding.

2.2 Scene Modeling

The problem of scene modeling was initially chosen for this thesis as it serves as a foundational problem for many others. For instance, an agent must have a representation of the environment being traversed to plan a walking motion. Scene modeling in 3D-space[21, 16, 17] provides applications with the fundamental representation mechanisms. Genetic approaches toward scene modeling usually involve a population of scene configurations through the creation and progressive modifications, culminating in a final solution. This approach provides an approximation of object location within the scenes. Recently, a graph-based technique for generating scene models was proposed capable of encoding scene models [22]. This approach can model scenes in 2D and 3D space recursively. The authors note that the relations between objects can be viewed recursively and are prime candidates for modeling using graphs.

A significant driving factor for developing image processing and scene management

has been the development of autonomous robot agents. In the past, there have been several attempts at using simulation to train robots. Past research has typically focused on three separate applications for artificial intelligence (AI), these have been:

1. To train perception models for scene modeling and object recognition. [16, 17] These perception models often use images of object-heavy environments and task the model with finding the configuration of objects from the image. This configuration of objects, positions, and rotations can be projected back into a simulated scene approximation. Scene modeling is an ongoing area of research in AI, augmented reality (AR), and robotics.

2. To train more efficient walking gaits. [18, 19] Both evolutionary algorithms and traditional machine learning models have seen applications in generating energy-efficient walking gaits for bipedal and quadrupedal robots. 3) To model the physics and dynamics of an environment [20]. While very similar to scene modeling, physics modeling focuses primarily on the ability to use a representation of objects and predict future states of the scene. This is useful for planning locomotion, as it allows one to learn graph-structured body plans and how different physical bodies react when colliding.

3. Path planning and task planning. [23] To complete various complex tasks, steps must be completed in a predefined order. Planning is a critical aspect of many tasks and is important to the task proposed in this thesis, described in a later section.

Past simulation-based approaches for agent evaluation often struggle with task

evaluation issues, as it can become difficult to define a fitness/scoring function directly for certain tasks. Researchers have recently explored the potential of solving simplistic "Animal-AI" tests to foster more complex problem-solving for our AI agents [24, 25]. Problem-solving is a desirable skill for people. This is considered a limitation of current AI agents. In the past, research was inspired by the survival requirements of real-world animals. [24] Past attempts to find a general test bed for evaluating these problem-solving skills have focused on topics such as maze-solving and tests of object permanence. [25] Moreover, agent-based models simulate discrete agents interacting within a simulated environment. While these models typically represent some properties of a larger system, some ABMs have explored using machine learning models to control agent decision-making. These models typically use simple feed-forward neural networks [15] and inherently select for neuron parameters associated with the highest survivability. Convolutional neural networks (CNN) are a popular variation of the neural network intended to detect features and process images. CNN works by moving a filter over each pixel of a given input image. These networks will then typically include a pooling layer, further reducing the feature maps' scale. The pixel values of these much smaller feature maps are then passed to a fully connected neural network.

2.3 Agent Based Models

Agent-based models are often concerned primarily with the interactions between individual agents and their environment. These models consist of a collection of stimulated agents, including some environmental representation. An agent is defined by

rules indicating the interactions between itself and other agents and their environment. The representation of the environment is a significant component in agent-based models. Agents operate within and interact with this representation during the simulation process. Agent-based models have many applications [26]. Agents are used in modeling and recreating complex interactions for complex systems, such as foot-traffic modeling for increasing pedestrian flow in commercial applications [27] or modeling wildfire [28] spread or animal population for conservation.

2.3.1 Predator-Prey Models

The topic of ABMs covers a wide range of topics. This thesis utilizes just one type of ABM, commonly called a predator-prey model. These models are often used to model the effects of environmental changes within animal populations where one species is prey to another species[15]. These models have two main components: first, they represent their environment, and then we have the agents themselves. These agents are divided into two species: predators and prey. The prey agents must seek food from their environment in the form of metabolic energy to avoid starvation. The predator agents must acquire their metabolic energy from prey agents, removing the prey from the simulation in the process. Environment models vary widely in these types of ABMs as different representations help to illustrate different features of the model that the user may be looking for. For example, while Using a 2D grid of cells to represent the world is commonplace, some implementations choose not to have agents that can interact with their environment altogether [29]. This study instead uses floating point values for the positions of agents, allowing for less

restrictive movement as agents are no longer aligned to a grid. This study focused on implementing a swarming behavior for the predator agents. This thesis will use a more typical 2D grid as its initial model implementation, and using procedural generation techniques, it will create a 3D rendering of the 2D grid world. To our knowledge, the literature does not reveal any applications of ABM-like simulations for evaluating these sorts of scene modeling problems. During this research project, I seek to explore new opportunities to leverage these technologies to solve more complex problems of problem-solving and planning.

2.4 Evolutionary algorithms

Evolutionary algorithms have long been recognized for their ability to rapidly iterate and generate solutions to complex problems [30]. Genetic algorithms typically iterate through "generations," where a population of potential solutions is evaluated and modified. This modification is often the result of three distinct operators, "selection," "crossover," and "mutation." [31] The selection operator selects a subset of the solution population for the next generation. This selection is often problematically related to the evaluation of the given solution. The crossover operator is used to replace the lost members of the population, often by constructing new solutions based on two solutions from the selected sub-population. This newly created population replaces the old population for the next generation. Finally, the mutation operation may slightly modify each solution's values. This operator has a small chance to change solutions by a small amount. These three operators work together to generate new solutions and progressively trend toward better-fitting solutions [31].

One such use of these algorithms has been used to train models for developing robot locomotion[32]. Planning walking gaits is also a current problem in robotics, which is open to exploration and has seen success with the application of these genetic algorithms [33, 18, 19, 21]. Planning paths and motion often require some representation of the environment being navigated. Gong and Yang showed they were able to apply a genetic algorithm to the problem of stereo-image processing [34] In this method, two stereo images of the same scene are given, and a disparity map of the image is generated from the difference between the two images.

2.5 SLAM

Simultaneous localization and mapping, also known as SLAM, is the process of building and maintaining an agent's scene model and location within it while also navigating the agent's environment. This is done by alternating between taking measurements of perception information available to the agent and comparing those measurements to those predicted by the internal scene model of the agent. Once the internal model is updated, the agent takes action and moves within the environment. This alternating does not necessarily need to happen in a 1:1 ratio, and instead, multiple steps may be taken before updating the internal model[35, 36, 37]. SLAM models, which use visual information provided by cameras, are a fairly common approach[36]. Certain SLAM models have used neural nets in order to control agents' navigation[37]. The paper shows that these structures can be efficiently used in both mapping and navigation in simulated and real-world robots.

2.6 Research Gap

To our knowledge, however, there has been no direct study on the effectiveness of using agent-based models to train agents capable of on-the-fly scene modeling. Approaches for evaluating scene modeling typically rely on outside data sets such as the Compositional CLEVR dataset [22]. Likewise, I could not discover any applications of ABMs for evaluating such scene modeling problems. During this research, I seek to explore new opportunities to leverage these technologies to solve more complex problems of problem-solving and planning.

There has been no direct study on the effectiveness of using survival simulations to directly train agents capable of solving animal cognition tasks such as object permanence or environmental traversal, so this thesis aims to start filling that gap. The key inspiration behind this proposed thesis is the cognitive skills approach developed through the complex task of "survival." Past studies have attempted to deduce the key aspects of this task [24], such as "Object permanence", or "Maze-Solving", amongst others. However, whereas the initial papers on the animal AI testbed [25, 24] outlined ten problem-solving skills that are considered required for survival in nature, the use of ABMs as presented in this thesis allows us to model the process of survival in nature much more directly. I could not find applications for these models within an agent-based model that uses more complex machine learning models, such as graph-convolutional networks.

This is a valuable study area as it aims to develop better causal reasoning skills in AI, allowing for a much wider array of tasks to be more effectively handled by AI agents. This thesis attempts to train agent models for increased accuracy in

scene modeling by selecting agents capable of recognizing and modeling the relative locations of landmarks in their simulated environment to test and demonstrate the feasibility of this method. In this thesis, I model the complexities of this basic survival scenario by utilizing similar modeling techniques found in ABM applications. As effective scene modeling will provide a considerable advantage over agents with inaccurate scene reconstructions, I hypothesize that increased scene modeling proficiency would correlate with increased survivability. Additionally, while the agent-based model used in this thesis is a Predator-Prey type, various other ABMs may be rendered in a similar manner to create scenarios for training more complex agent behaviors.

Chapter 3

Predator-Prey Agent-Based Model Simulation

3.1 Agent Based Model Simulation

A predator-prey agent-based model was chosen as the underlying event controller of the simulation. The agents in the model maintain a metabolic requirement and compete for resources within the environment. The prey-predator model exhibits a distinctive population cycle that acts as an indicator of the stability of the simulated ecosystem.

There are two main components- the *world* represented as a 2D grid of cells and the *agents* occupying these cells. The model is simulated and updated in discrete time intervals. The simulation terminates when 100,000-time steps have occurred or if the population of either predator or prey agents drops to zero. Each cell in the world's representation is associated with an energy value. The energy amount represents

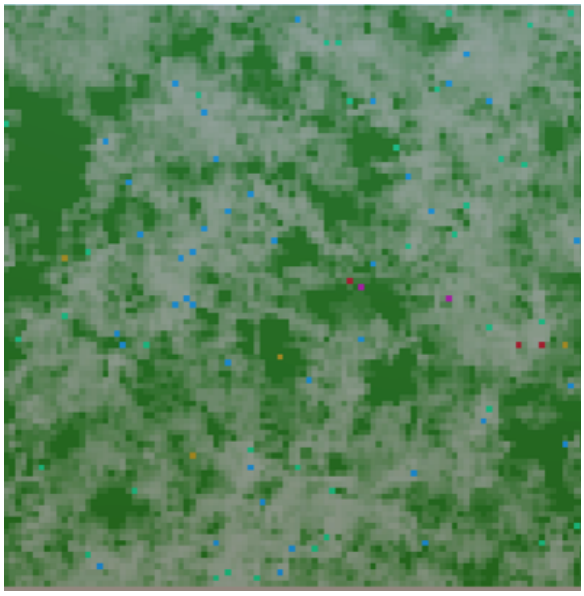

Figure 3.1: A snapshot of the ABM without additional rendering

the maximum available energy the cell could provide to an agent. In addition, the energy value is increased by a small amount at each time interval. This is the cell *regrowth* factor. Agents also have a metabolic *energy* variable. The agent's metabolic energy is its survivability indicator. The value could be above or below a user-defined reproductive threshold. At each time interval, the agent's metabolic energy is reduced by a *decay* factor. The parameters, regrowth, and decay have a significant impact on the total sustainable agent population. To produce the needed population cycles, the parameters should be fine-tuned.

At each time interval, agents must relocate to one of four adjacent positions on the world grid. For our typical agent, this is chosen randomly. In addition to this regular agent behavior, some agents use an internal SLAM model to decide which position to move to. Section 3.3 will explain the details of the SLAM model. These agents will attempt to move into an adjacent position with the lowest perceived chance of

another agent occupying that cell.

The 2D agent-based model (ABM) is rendered as a full 3D environment to fully utilize the perception model. This 2D model can be visualized in Figure 3.1. The agent-based model is implemented within the *Unity* game engine environment. This environment contains all the required features to extend the typical 2D ABM to the 3D landscape needed for more complex interactions between agents and their environment. For example, in a typical 2D ABM environment, an agent looking towards another agent may find it difficult to observe if blocked by some part of the environment. This is one of the shortcomings I try to address by providing a 3D spatial dimension.

3.2 3D Rendering

The process of expanding the 2D ABM to a 3D environment is described in this section. It's necessary to reconstruct the ABM environment in 3D for agent points-of-view (PoV) to be correctly rendered.

3.2.1 Environment Rendering

The first step is to create a height map using the Perlin noise generator. This process is done by adding layers of Perlin noise (octaves). Each point on the height map is converted to a vertex point, and the X and Y positions in the height map and the map's value at that position are used to determine the Z coordinate. This allows a series of vertices, which can then be stitched together to create a 3D mesh for each cell in the ABM. Each of these cells is constructed using two additional

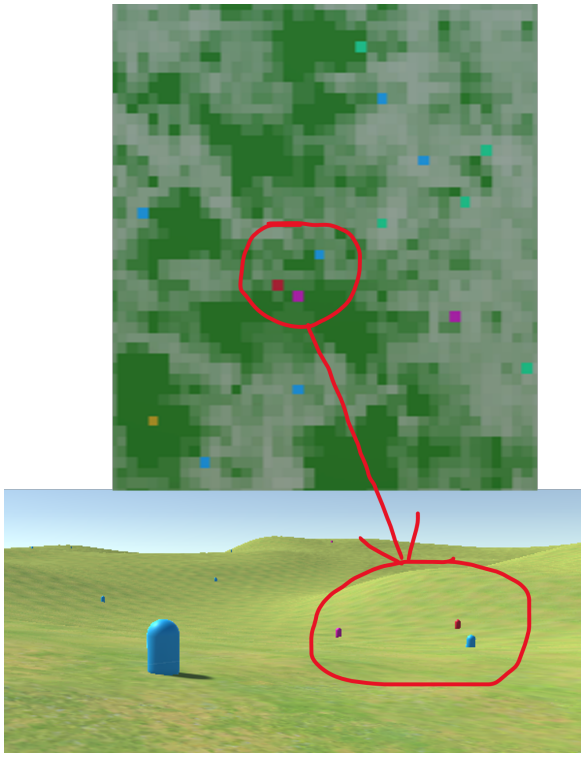

Figure 3.2: Example of 3D extrapolation of a portion of the ABM

parameters. The *Cell-Scale* parameter is a scalar value that determines the size of the ABM cell to be rendered. The other parameter, *Cell-Complexity* determines the number of vertices to be used along the length and width of a cell. This parameter is used to make the terrain more or less detailed. Once the mesh is constructed from these vertices, the created terrain mesh is ready for the start of the simulation. An example of such terrain rendering can be seen in figure 3.2.

3.2.2 Agents Rendering

Agents are represented within the 3D rendering. They appear as capsules colorized to match their ABM counterparts. With both the environment and agents rendered,

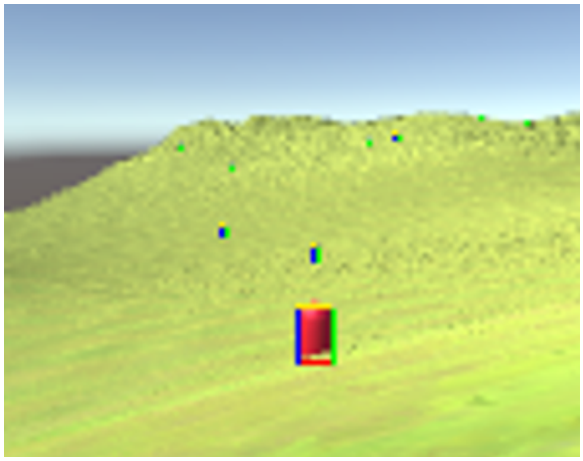

Figure 3.3: POV example with other agents outlined

agent perspectives can be integrated into the ABM model. Agent perspectives can be used in several ways. First, the agent's perspectives can be rendered to an image directly. This is useful for training convolutional neural networks within ABMs to save time on training later. However, this approach was extremely hardware intensive and could not be extended to every agent without significant performance impact. Therefore, other methods of determining the agent's PoV were needed so that it could be applied to all the agents in the simulation.

One approach is to calculate the region surrounding the agent's camera. This provides more regional information to determine if any agent or object of interest is within that region. This is shown in Figure 3.3. However, using this approach, parts of the world may occlude the object of interest. This effect is shown in Figure 3.4. Therefore, I use a feature from the Unity engine (called *raycast*) to circumvent this occlusion. This feature determines if an object is in line of sight. It casts a 'ray' from an origin point to a destination point and returns the first object the ray intersects with along its path. The ray is cast from the observer agent to all objects of interest

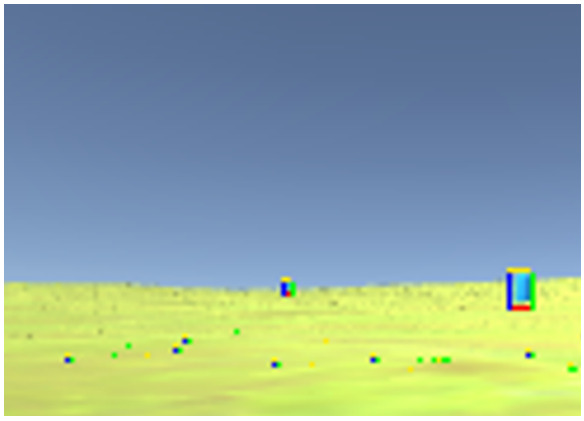

Figure 3.4: POV example without raycasting.

within the agent's viewpoint. Collision of the ray with the object indicates the object is within the visual area of the agent's interest and in line of sight. This method of determining what is visible to each agent is much more efficient than directly rendering the scene from all the agent's perspectives. Moreover, the information provided by this method allows agents to track the location of the nearest agent from their perspective position. At this point, the agents have mapped an area within its location. To better understand how this is done, I will explain the SLAM model used in the agents in greater detail in the next section.

Additionally, The visual field can be sampled directly for use with CNN structures. An example of some commonly used convolutes was implemented using Unity's compute shaders. The outcome is shown in figure 3.5. Unfortunately, due to processing times present in this approach, it is not viable to sample the visual field of all agents within the simulation at each time interval. For this reason, the simpler ray-tracing process outlined above will be the main focus of this thesis. This CNN example is retained as a proof of concept potentially for applications that don't involve such a

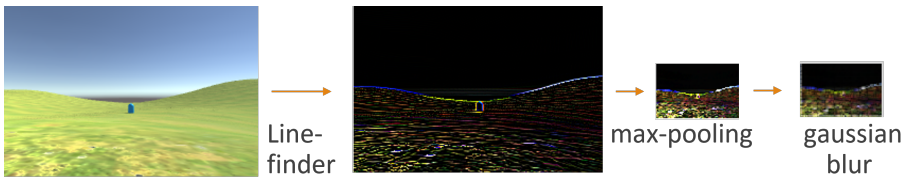

Figure 3.5: ABM CNN image processing example.

large number of agents.

3.3 Agents' SLAM model

The agent's SLAM model consists of a set of sample "guesses" as to the location of the nearest agents. As the number of these samples increases, the resolution of the model increases. However, this also increases the complexity. Therefore, the number of samples and the number of agents with independent instances of the SLAM model limits the number of samples that can used in each SLAM model instance. Therefore, in our experiments, I limit the number of samples in each model to 100. The SLAM model performs distinct updates in three steps: Sampling, Extrapolation, and Prediction.

3.3.1 Sampling

First, the agent's visual field returns a list of agents within its line of sight. The SLAM model tracks the closest agent by comparing the distance between the agent (observer) and the observed agents within line of sight. From this comparison, the closest agent is chosen for sampling. As mentioned above, the agent's SLAM model consists of a set of sample "guesses" as to the location of the nearest agents. The

SLAM model assigns a weight to each of the sample "guesses" by measuring the Manhattan distance between the sampled location and the guesses' estimated positions. At each interval, each agent moves in one of the four directions on the grid. Therefore, the probability that any given direction will be chosen is 0.25. The weight of each sample is calculated as $(0.25)\hat{\text{M}}\text{anhattan\_Distance}$. The next generation of samples is selected randomly from these weighted samples. This has the desired effect of focusing on the guesses around the observed agent.

3.3.2 Extrapolation

The simple SLAM model also extrapolates on the predicted location of the observed agent. As each agent moves in one of four directions with a probability of 0.25, each model's sample simulates one potential outcome in the next time interval by applying a movement in one of these directions. The extrapolation effect expands the model sampled estimates, allowing the samples to not converge at a single point.

3.3.3 Prediction

At the end of the extrapolation step, the model predicts the likelihood of an agent present at any given position. The model is queried with a 2D position on the ABM map grid. This position is compared to each of the resampled and extrapolated guesses in the model, and the number of samples that place the closest agent in that position is counted. The proportion of samples encountering the nearest agent in that position is returned as the predicted likelihood that the closest agent is in that position. By comparing the probability at each of the four positions around the

SLAM-avoidance agent, the agent selects the adjacent tile with a lesser probability of encountering another agent. In the event of a tie, the location is chosen from those options at random.

3.4 Granularity

ABMs are updated incrementally in discrete time intervals. This means that agents' positions are updated from tile to tile instantly. Depending on the scaled size of each ABM land tile, the actual perceived position for the agent may be significantly different from the previous time interval. To address this, as the agent moves through the simulation, each step can be further broken down into segments. This allows the agents' SLAM model to receive multiple update steps while the agents' position is updated. This is done simply by interpolating each agent's position and rotation from its starting position and orientation to its final position and orientation. To keep the set of available movements simple for the agents, the orientation of the agent is represented as the direction of travel from the previous step, i.e., the direction the agent is facing after it finishes its movement. During the experiment presented in the next Chapter, I will use a step granularity of five updates per agent step. Using the improved PoV calculations, the speed attained with this granularity was such that it could be performed for all agents in the simulation without a significant decrease in computation time.

Chapter 4

Experiments and Results

This chapter explains the experimentation conditions and results. During the defense, a demonstration of the agents' behavior will help explain the experiments. The appendix provides the program code.

The experiment is as follows. First, the implemented ABM parameters are tuned until the simulated population reaches equilibrium with a reasonable number of agents to compute. The population cycles typically noted in these types of ABMs fluctuate across a wide range. I found the range of roughly 50 to 200 agents to be an ideal trade-off between computation time per time interval and ensuring there are sufficient agents for the population to remain stable. The SLAM agents are introduced once this baseline is established, and the experimental trials begin. These SLAM agents will have an internal SLAM model described in chapter 3.3 for tracking other nearby agents. The intent is to bring the two agent types into direct competition. The completed trials are then compared to see which agents remained at the end of each trial.

4.1 Starting Conditions

In each experimental trial, in the beginning, the simulation is initialized with 100 prey agents. Fifty of the prey agents have internal SLAM models. The simulation is also initialized with 8 predator agents, with 4 of these agents having internal the SLAM model. The agents are placed in a uniform distribution on the map. All-terrain cells' energy density is set to half its total possible energy. This is done for two reasons. First, when the environment is set to its maximum energy density, the prey species will respond by rapidly increasing its population. This, in turn, results in an abnormal increase in predator population, and the resulting ecosystem is generally not stable. The second reason is to simulate the energy demands on the environment before starting the simulation. While environments could be regenerated in a new configuration, validating the agent-based model would be difficult. Therefore, the height map and resulting terrain mesh are preserved across trials. Additionally, extra parameters such as *regrowth factor* of grass resources and *movement cost* are selected so that the simulation is stable before adding the agents with internal SLAM models. These models are often run to determine the stability of the agent population. To that end, the simulation must be run long enough for the simulated ecosystem to stabilize. I noted from previous tests that the population cycles once every 5,000 - 10,000 time intervals. Additionally, the longer the simulation is run, the higher the likelihood that the ecosystem's stability will break down before the trial ends. With this in mind, I chose to run the simulation at a total of 100,000 time intervals per trial. This is a user-defined number intended to show multiple cycles of the population trend in order to establish the ecosystem is stabilized without running the simulation for so

long that all agents expire. The population count of each of the four agent species is monitored at each time interval and is additionally recorded at the end of the experiment trial. Additionally, the simulation will terminate early if the population count of either the prey or predator agents drops to zero.

4.2 Results

The simulation was run for a total of 123 trials. Initially, the population count of both predator and prey agents fluctuates erratically. This erratic behavior eventually settles into the cyclical and repetitive population fluctuation typically observed in predator-prey agent-based models. At least one of the four agent varieties died out in all the trials. 63 trials ended early due to the population count of either the prey or predator agents reaching zero, which I refer to as incomplete trials. Results from the incomplete trials are shown in Table 4.1. This table breaks down the number of incomplete trials that ended with each agent type. The numbers 22, 22, 15, and 16 indicate the number of incomplete trials that finished with active agent variants: random walk prey, random walk predators, SLAM prey, and SLAM predators, respectively. The randomly walking predators and the randomly walking prey, (identified as *Random Walk. Prey and Random Walk.* Pred in Table 4.1, respectively) were the last agents remaining in approximately 35% (22/63 trials) of the incomplete cases. The SLAM variants of the predator and prey (identified as *SLAM-Avoid Prey* and *SLAM-Avoid Pred* in Table 4.1) remained at only 25% (16/63 trials) and 24% (15 trials/63), respectively. This shows a slight preference towards the randomly walking agents outperforming the SLAM agents. However, an interesting

trend appears when considering the completed trials.

Incomplete Trials			
Random Walk. Prey	Random Walk. Pred	SLAM-Avoid Prey	SLAM-Avoid Pred
22	22	15	16
35%	35%	24%	25%

Table 4.1: Incomplete Trials

A breakdown of remaining agent types across the completed trails is shown in Table 4.2. Of the remaining 60 trials that completed the full 100000-time intervals, agent species' survival rates were fairly uniform across all four agent variants. That is, for random walk prey and predators, it is approximately 48This distribution initially suggests that no individual agent variety is preferred over another. However, A slightly different trend is seen from Table 4.3.

Table 4.3 displays the proportion of strategies used by agents in the complete trials. Out of the 60 completed trials, 45 trials (or 75%) had either the predator or prey agents preferring to use the random walk strategy. This is a strong indicator that the population is more stable when the random walk strategy is present than when it is absent. In addition, I note that 80% (or 48 out of 60) of the completed trails had at least one of the predator or prey agents employing the SLAM agent-avoidance strategy. This indicates that the ecosystem is most stable when both navigation strategies are present.

Another interesting relationship is between agents of the same species. Only one trial was completed, and both movement strategies had prey agents. Similarly, none of the trials completed had predator agents of both movement strategies. One reason may be that agents commonly interact with the same movement strategy, providing

an opportunity for predators to frequently consume the prey species. This potentially creates instability in the simulated ecosystem, leading to its collapse before all time increments are simulated. These observations indicate the ecosystem strongly prefers that at least one agent type uses one movement strategy while the other uses another.

Complete Trials			
Random Walk Prey	Random Walk Pred	SLAM-Avoid Prey	SLAM-Avoid Pred
29	29	32	31
48%	48%	53%	52%

Table 4.2: Completed Trials - Remaining agents

Complete Trials			
Both Prey	Both Pred	Either Rand	Either SLAM
1	0	45	48
2%	0%	75%	80%

Table 4.3: Completed Trials - Employed strategy

Additionally, I performed a breakdown of the movement strategy preferred by each agent type. The proportion of prey strategy preferred by predator agents can be seen in Table 4.4. This table is separated into two sections for the two predator strategies. Regardless of the predator strategy chosen, there appears to be little to no preference for randomly walking prey or avoiding it with SLAM. Similarly, the proportion of predator strategy preferred by prey agents can be seen in Table 4.5. This breakdown also shows no clear preference for predator or prey agent types.

Prey breakdown by Pred			
Random Walk. Pred		SLAM-Avoid Pred	
Random Walk. Prey	SLAM-Avoid Prey	Random Walk. Prey	SLAM-Avoid Prey
13	17	16	15
43%	57%	52%	48%

Table 4.4: Breakdown of remaining prey agent strategies preferred by predator agents

Pred breakdown by Prey			
Random Walk. Prey		S. Prey	
Random Walk. Pred	SLAM-Avoid Pred	Random Walk. Pred	SLAM-Avoid Pred
13	16	17	15
45%	55%	53%	47%

Table 4.5: Breakdown of remaining predator agent strategies preferred by prey agents

4.2.1 Simulation run observations

The difference between the agents' behavior can be seen while the simulation runs. The SLAM-avoid agents will spread themselves out evenly across all available space. When predator and prey agents employ this strategy, they maintain a fairly consistent and short distance. In this case, there is a negative effect on the SLAM-prey agents as the proximity of predator agents puts them at constant risk. Agents that instead employ the randomly walking strategy tend to be much more densely grouped. These agent groups are much more isolated when compared to the more uniform distribution of the SLAM-avoid agents. This allows areas of relative safety for the prey agents as long as predator and prey agents employ different strategies. This leads to greater ecosystem stability within the simulation. These observations are consistent with the trend seen in the completed trials, which shows the agents prefer a combination of

navigation strategies.

Chapter 5

Conclusion and Future Work

While this is fundamentally a ground-laying research endeavor, I found from my analysis that the system prefers a variety of movement strategies rather than either random walk or SLAM avoidance. This ensures a stronger boundary between the separate agent species, leading to a population-stabilizing effect within the simulation. These findings open the door to many new avenues of experimentation. For instance, it could open other ecological niches by slightly modifying the agent's internal model and movement behaviors. This would allow for more complex simulations involving more sub-species to more accurately model behaviors noted in studies of animals in the wild.

On the agent-cognition side, these simulative models can test other SLAM models in an environment that can occlude vision and account for lost lines of sight. The environment itself could be substituted for an indoor environment where agents are graded by how carefully they can navigate the space. Models inspired by this one could form the basis of a testing platform that could test the robustness of these

SLAM algorithms while tracking moving objects.

The complexity of the presented simulation could also be increased. At its core, the ABM model is a simple implementation of standard predator-prey models. Other predator-prey models have had more complex environmental interactions, such as bodies of water. These additions could likewise be applied to our ABM to increase the complexity of the simulation and, by extension, the complexity of the agents' behaviors within the ABM.

Finally, modifications can be made to the agent's internal model to better account for the environment. This could be an internal elevation map or the ability to track landmarks. Additionally, as future navigation and environmental modeling techniques are developed, they may also be implemented similarly to the simplified SLAM model seen in this implementation.

Bibliography

[1] Lei Zhang, Rene Zapata, and Pascal Lepinay. Self-adaptive monte carlo localization for mobile robots using range finders. *Robotica*, 30(2):229–244, 2012.

[2] Patrick Pfaff, Wolfram Burgard, and Dieter Fox. Robust monte-carlo localization using adaptive likelihood models. In *European robotics symposium 2006*, pages 181–194. Springer, 2006.

[3] Cynthia Matuszek, Evan Herbst, Luke Zettlemoyer, and Dieter Fox. Learning to parse natural language commands to a robot control system. In *Experimental robotics: the 13th international symposium on experimental robotics*, pages 403–415. Springer, 2013.

[4] David Hall, Ben Talbot, Suman Raj Bista, Haoyang Zhang, Rohan Smith, Feras Dayoub, and Niko Sünderhauf. The robotic vision scene understanding challenge. *arXiv preprint arXiv:2009.05246*, 2020.

[5] Andrew T Crooks and Alison J Heppenstall. Introduction to agent-based modelling. In *Agent-based models of geographical systems*, pages 85–105. Springer, 2011.

[6] Mattia Guerini and Alessio Moneta. A method for agent-based models validation. *Journal of Economic Dynamics and Control*, 82:125–141, 2017.

[7] Charles Macal and Michael North. Introductory tutorial: Agent-based modeling and simulation. In *Proceedings of the winter simulation conference 2014*, pages 6–20. IEEE, 2014.

[8] Farid Mirahadi, Brenda McCabe, and Arash Shahi. Ifc-centric performance-based evaluation of building evacuations using fire dynamics simulation and agent-based modeling. *Automation in Construction*, 101:1–16, 2019.

[9] Zhang Y Dai D. Simulating fire spread in a community using an agent-based model. In *Proceedings of the 12th International Conference on GeoComputation. LIESMARS Wuhan University, Wuhan, China*, pages 130–132, 2013.

[10] Michael Batty, Bin Jiang, and Mark Thurstain-Goodwin. Local movement: agent-based models of pedestrian flows. 1998.

[11] Amgad Naiem, Mohammed Reda, Mohammed El-Beltagy, and Ihab El-Khodary. An agent based approach for modeling traffic flow. In *2010 The 7th international conference on informatics and systems (INFOS)*, pages 1–6. IEEE, 2010.

[12] Megan M Olsen and Rachel Fraczkowski. Co-evolution in predator prey through reinforcement learning. *Journal of computational science*, 9:118–124, 2015.

[13] WJ Chivers, W Gladstone, RD Herbert, and MM Fuller. Predator–prey systems depend on a prey refuge. *Journal of theoretical biology*, 360:271–278, 2014.

[14] Jacopo A Baggio, Kehinde Salau, Marco A Janssen, Michael L Schoon, and Örjan Bodin. Landscape connectivity and predator–prey population dynamics. *Landscape Ecology*, 26:33–45, 2011.

[15] Adam J McLane, Christina Semeniuk, Gregory J McDermid, and Danielle J Marceau. The role of agent-based models in wildlife ecology and management. *Ecological modelling*, 222(8):1544–1556, 2011.

[16] Heesoo Myeong, Ju Yong Chang, and Kyoung Mu Lee. Learning object relationships via graph-based context model. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 2727–2734. IEEE, 2012.

[17] Muzammal Naseer, Salman Khan, and Fatih Porikli. Indoor scene understanding in 2.5/3d for autonomous agents: A survey. *IEEE access*, 7:1859–1887, 2018.

[18] Stelian Coros, Philippe Beaudoin, and Michiel Van de Panne. Generalized biped walking control. *ACM Transactions On Graphics (TOG)*, 29(4):1–9, 2010.

[19] Duy Nguyen-Tuong and Jan Peters. Model learning for robot control: a survey. *Cognitive processing*, 12(4):319–340, 2011.

[20] Alvaro Sanchez-Gonzalez, Nicolas Heess, Jost Tobias Springenberg, Josh Merel, Martin Riedmiller, Raia Hadsell, and Peter Battaglia. Graph networks as learnable physics engines for inference and control. In *International Conference on Machine Learning*, pages 4470–4479. PMLR, 2018.

[21] Robert Eidenberger and Josef Scharinger. Active perception and scene modeling

by planning with probabilistic 6d object poses. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1036–1043. IEEE, 2010.

[22] Fei Deng, Zhuo Zhi, Donghun Lee, and Sungjin Ahn. Generative scene graph networks. In *International Conference on Learning Representations*, 2020.

[23] Yixin Lin, Austin S Wang, Eric Undersander, and Akshara Rai. Efficient and interpretable robot manipulation with graph neural networks. *IEEE Robotics and Automation Letters*, 2022.

[24] Matthew Crosby. Building thinking machines by solving animal cognition tasks. *Minds and Machines*, 30(4):589–615, 2020.

[25] Matthew Crosby, Benjamin Beyret, Murray Shanahan, José Hernández-Orallo, Lucy Cheke, and Marta Halina. The animal-ai testbed and competition. In *NeurIPS 2019 competition and demonstration track*, pages 164–176. PMLR, 2020.

[26] Dirk Helbing. Agent-based modeling. In *Social self-organization: Agent-based simulations and experiments to study emergent social behavior*, pages 25–70. Springer, 2012.

[27] Andrew Crooks, Arie Croitoru, Xu Lu, Sarah Wise, John M Irvine, and Anthony Stefanidis. Walk this way: Improving pedestrian agent-based models through scene activity analysis. *ISPRS International Journal of Geo-Information*, 4(3): 1627–1656, 2015.

[28] Sarah A Grajdura, Sachraa G Borjigin, and Deb A Niemeier. Agent-based wildfire evacuation with spatial simulation: a case study. In *Proceedings of the 3rd ACM SIGSPATIAL International Workshop on GeoSpatial Simulation*, pages 56–59, 2020.

[29] Vladimir Zhdankin and JC Sprott. Simple predator-prey swarming model. *Physical Review E—Statistical, Nonlinear, and Soft Matter Physics*, 82(5):056209, 2010.

[30] Adrian Agogino, Ritchie Lee, and Dimitra Giannakopoulou. Challenges of explaining control. In *2nd ICAPS Workshop on Explainable Planning (XAIP'19)*, 2019.

[31] Manoj Kumar, Mohammad Husain, Naveen Upreti, and Deepti Gupta. Genetic algorithm: Review and application. *Available at SSRN 3529843*, 2010.

[32] Sonia Chernova and Manuela Veloso. An evolutionary approach to gait learning for four-legged robots. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, volume 3, pages 2562–2567. IEEE, 2004.

[33] Qiang Huang, Kazuhito Yokoi, Shuuji Kajita, Kenji Kaneko, Hirohiko Arai, Noriho Koyachi, and Kazuo Tanie. Planning walking patterns for a biped robot. *IEEE Transactions on robotics and automation*, 17(3):280–289, 2001.

[34] Minglun Gong and Yee-Hong Yang. Multi-resolution stereo matching using ge-

netic algorithm. In *Proceedings IEEE Workshop on Stereo and Multi-Baseline Vision (SMBV 2001)*, pages 21–29. IEEE, 2001.

[35] Weifeng Chen, Guangtao Shang, Aihong Ji, Chengjun Zhou, Xiyang Wang, Chonghui Xu, Zhenxiong Li, and Kai Hu. An overview on visual slam: From tradition to semantic. *Remote Sensing*, 14(13):3010, 2022.

[36] Guangyu Fan, Jiaxin Huang, Dingyu Yang, and Lei Rao. Sampling visual slam with a wide-angle camera for legged mobile robots. *IET Cyber-Systems and Robotics*, 4(4):356–375, 2022.

[37] Devendra Singh Chaplot, Dhiraj Gandhi, Saurabh Gupta, Abhinav Gupta, and Ruslan Salakhutdinov. Learning to explore using active neural slam. *arXiv preprint arXiv:2004.05155*, 2020.

[38] S. Saeedi, M. Trentini, M. Seto, and H. Li. Multiple-robot simultaneous localization and mapping: a review. *J. Field Robotics*, 33:3–46, 2016.

[39] W. Rone and P. Ben-Tzvi. Mapping, localization and motion planning in mobile multi-robotic systems. *Robotica*, 31:1–23.

[40] Miquel Kegeleirs, Giorgio Grisetti, and Mauro Birattari. Swarm SLAM: Challenges and perspectives. *Frontiers in Robotics and AI*, 17, 2021.

[41] J. Katan and L. Perez. Abwise v1.0: toward an agent-based approach to simulating wildfire spread. *Natural Hazards and Earth System Sciences*, 21(10): 3141–3160, 2021.

[42] Montello D.R. Navigation. In Shah P. and Miyake A., editors, *The Cambridge Handbook of Visuospatial Thinking*, page 257–294. Cambridge University Press, New York, NY, USA, 2005.

[43] Tolman E.C. Cognitive maps in rats and men. In *Psychol. Rev.*, volume 55, page 189–208, 1948.

[44] S. Schwarz, A. Wystrach, and K. Cheng. Ants' navigation in an unfamiliar environment is influenced by their experience of a familiar route. *Sci Rep*, 7 (14161), 2017.

[45] Delgado Mikel M. and Jacobs Lucia F. Caching for where and what: evidence for a mnemonic strategy in a scatter-hoarder. 4.

[46] Jun Cheng, Liyan Zhang, Qihong Chen, Xinrong Hu, and Jingcao Cai. A review of visual slam methods for autonomous driving vehicles. *Engineering Applications of Artificial Intelligence*, 114, 2022.

[47] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020.

[48] William E Lorensen and Harvey E Cline. Marching cubes: A high resolution 3d surface construction algorithm. *ACM siggraph computer graphics*, 21(4):163–169, 1987.

[49] John H Holland. *Adaptation in natural and artificial systems: an introductory*

analysis with applications to biology, control, and artificial intelligence. MIT press, 1992.

[50] Sun-Chong Wang. Genetic algorithm. In *Interdisciplinary computing in java programming*, pages 101–116. Springer, 2003.

[51] Nico Cornelis, Bastian Leibe, Kurt Cornelis, and Luc Van Gool. 3d urban scene modeling integrating recognition and reconstruction. *International Journal of Computer Vision*, 78(2):121–141, 2008.

[52] Jingen Liu and Mubarak Shah. Scene modeling using co-clustering. In *2007 IEEE 11th International Conference on Computer Vision*, pages 1–7. IEEE, 2007.

[53] Matthew Fisher, Manolis Savva, Yangyan Li, Pat Hanrahan, and Matthias Nießner. Activity-centric scene synthesis for functional 3d scene modeling. *ACM Transactions on Graphics (TOG)*, 34(6):1–13, 2015.

[54] Karl Moritz Hermann, Felix Hill, Simon Green, Fumin Wang, Ryan Faulkner, Hubert Soyer, David Szepesvari, Wojciech Marian Czarnecki, Max Jaderberg, Denis Teplyashin, et al. Grounded language learning in a simulated 3d world. *arXiv preprint arXiv:1706.06551*, 2017.

[55] Terry Winograd. Understanding natural language. *Cognitive psychology*, 3(1): 1–191, 1972.

[56] Shinya Sumikura, Mikiya Shibuya, and Ken Sakurada. Openvslam: A versatile visual slam framework. In *Proceedings of the 27th ACM International Conference on Multimedia*, pages 2292–2295, 2019.

[57] Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. *Communications of the ACM*, 65(1):99–106, 2021.

[58] Daniil Lisus and Connor Holmes. Towards open world nerf-based slam. *arXiv preprint arXiv:2301.03102*, 2023.

[59] Adithyavairavan Murali, Tao Chen, Kalyan Vasudev Alwala, Dhiraj Gandhi, Lerrel Pinto, Saurabh Gupta, and Abhinav Gupta. Pyrobot: An open-source robotics framework for research and benchmarking. *arXiv preprint arXiv:1906.08236*, 2019.

[60] Yunzhi Lin, Jonathan Tremblay, Stephen Tyree, Patricio A Vela, and Stan Birchfield. Multi-view fusion for multi-level robotic scene understanding. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 6817–6824. IEEE, 2021.

Appendix A

Source Code

ABMController.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.UI;
using System.IO;


public class ABMController : MonoBehaviour {
    public int numChosen = 1;
    private static ABMController singleton = null;
    public static string filePrefix = "AgentPaths\\agent_";
    public float timeDelta = 4.0f / 60.0f; //4/60 = 15 fps
    public int mapLength = 100;
    public float cellScale = 10;
    public List<AgentABM> chosenAgents;

    public int visionSegments = 5;
    public ControlMode globalControlMode;
    public int predCount = 8;
    public int preyCount = 60;
    public float upkeepCost = 0.02f;
    public int initialBreedDelay = 100;
```

```
public List<AgentABM> agents;
public int agentABMVisionDist = 2;

public ABMMapTile[,] map;
public Color theColor = new Color32(255, 0, 255, 255);
public GameObject miniMap;
private Texture2D miniMapImage;
public Gradient grassColorFood;

public float grassRegrowth = 0.01f;

public int iterationCount = 0;

public int globalPopulationCount = 0;
private List<int> popCountHistory = new List<int>();

public int globalPreyPopulationCount = 0;
private List<int> popPreyCountHistory = new List<int>();

public int globalPredPopulationCount = 0;
private List<int> popPredCountHistory = new List<int>();

public int globalPreySLAMPopulationCount = 0;
private List<int> popPreySLAMCountHistory = new List<int>();

public int globalPredSLAMPopulationCount = 0;
private List<int> popPredSLAMCountHistory = new List<int>();


public bool useSLAM = true;
public Color PredColor = new Color32(230, 0, 160, 255);
public Color PredColorSLAM = new Color32(230, 0, 160, 255);
public Color PreyColor = new Color32(0, 160, 230, 255);
public Color PreyColorSLAM = new Color32(0, 160, 230, 255);
Color deadColor = new Color32(230, 0, 20, 255);
public GameObject dummyTemplate;

public Text metricReadoutObj;

public ABMMetricChart preyChart;
public ABMMetricChart predChart;
public ABMMetricChart predDChart;
```

```
public ABMMetricChart preyDChart;

public HeightmapWorld terrainGen;
private float tileLength;

public bool drawBoundingBox = false;
public bool usingCNN = true;
public ComputeShader CNNcompute;

public int trialNumb = 0;

void writeOut()
{
    string output = "" + globalPopulationCount + " , "
        + globalPreyPopulationCount + " , " + globalPredPopulationCount
        + " , " + globalPreySLAMPopulationCount + " , "
        + globalPredSLAMPopulationCount + " , " + iterationCount + "\n";

    string outPath = Application.dataPath + "/../Trails/";
    if (!Directory.Exists(outPath))
    {
        Directory.CreateDirectory(outPath);
    }
    File.AppendAllText(outPath + "trial.txt", output);
}

void Start() {
    trialNumb = Random.Range(0, 100000);
    chosenAgents = new List<AgentABM>();
    for (int ii = 0; ii < numChosen; ii++)
    {
        chosenAgents.Add(null);
    }
    if (singleton == null) {
        singleton = this;
        InitializeABM();
    } else {
        Destroy(this.gameObject);
    }
}

public List<AgentABM> VisableAgents(Plane[] planes)
```

```
{
    List<AgentABM>  rVal= new List<AgentABM>();
    foreach (AgentABM agent in agents)
    {
        if (GeometryUtility.TestPlanesAABB(planes,
            agent.dummyBody.GetComponent<Collider>().bounds))
        {
            rVal.Add(agent);
        }
    }
    return rVal;
}

public void UpdateChosen()
{
    for (int ii = 0; ii < chosenAgents.Count; ii++)
    {
        if (chosenAgents[ii] == null || chosenAgents[ii].isDead == true)
        {
            chosenAgents.RemoveAt(ii);
            bool newAgent = false;
            while (!newAgent)
            {
                AgentABM newChosen =
                    agents[Random.Range(0, agents.Count)];
                if (!isChosen(newChosen))
                {
                    newAgent = true;
                    chosenAgents.Add(newChosen);
                    newChosen.isChosen = true;
                }
            }
        }
    }
}

public bool isChosen(AgentABM agent)
{
    bool isAgent = false;
    for (int ii = 0; ii < chosenAgents.Count && !isAgent; ii++)
    {
        if (chosenAgents[ii] == agent)
```

```
        {
            isAgent = true;
        }
    }
    return isAgent;
}

public Vector3 tileToWorldPoint(Vector2Int pos)
{
    Vector3 rval = Vector3.zero;
    if (terrainGen != null)
    {
        rval.x = (tileLength / 2f) + (tileLength * (pos.x))
            - (tileLength * mapLength / 2);
        rval.z = (tileLength / 2f) + (tileLength * (pos.y))
            - (tileLength * mapLength / 2);
        rval.y = terrainGen.getHeightAt(rval.x, rval.z);
    }
    return rval;
}

public Vector2Int worldPointToTile(Vector3Int pos)
{
    Vector2Int rval = Vector2Int.zero;
    if (terrainGen != null)
    {
        rval.x = (int)((pos.x - (0.5f * tileLength)) / tileLength);
        rval.y = (int)((pos.z - (0.5f * tileLength)) / tileLength);
    }
    return rval;
}


void InitializeABM()
{
    if (terrainGen != null)
    {
        terrainGen.generateWorld();
    }
    miniMapImage = new Texture2D(mapLength, mapLength);
    miniMapImage.filterMode = FilterMode.Point;
    map = new ABMMapTile[mapLength, mapLength];
```

```
        agents = new List<AgentABM>();
        for (int predCNT = 0; predCNT < predCount; predCNT++) {
            AgentABM agent = new AgentABM();
            agent.isPred = true;
            agent.metabolicEnergy *= 2f;//Suspect
            agent.reproThreshhold = 50f;
            agent.ABMPosition = new Vector2Int((int)(Random.value
                * mapLength), (int)(Random.value * mapLength));
            agent.breedDelay = initialBreedDelay;
            agent.brain = new AgentDecisionNetwork();
            agent.controlMode = globalControlMode;
            agent.generationCost = upkeepCost;
            agent.dummyBody = Instantiate(dummyTemplate);
            agent.dummyBody.GetComponent<MeshRenderer>().material.color =
                PredColor;
            agent.colorCode = PredColor;
            agent.cont = this;
            agents.Add(agent);
            agent.visSegments = visionSegments;
            agent.CNNcompute = CNNcompute;
            if(useSLAM == true && predCNT < predCount/2)
            {
                agent.controlMode = ControlMode.SLAM_integrated;
                agent.colorCode = PredColorSLAM;
            }
        }
        for (int preyCNT = 0; preyCNT < preyCount; preyCNT++) {
            AgentABM agent = new AgentABM();
            agent.isPred = false;
            agent.reproThreshhold = 50f;
            agent.ABMPosition = new Vector2Int((int)(Random.value
                * mapLength), (int)(Random.value * mapLength));
            agent.breedDelay = initialBreedDelay;
            agent.brain = new AgentDecisionNetwork();
            agent.controlMode = globalControlMode;
            agent.generationCost = upkeepCost;
            agent.dummyBody = Instantiate(dummyTemplate);
            agent.dummyBody.GetComponent<MeshRenderer>().material.color =
                PreyColor;
            agent.colorCode = PreyColor;
            agent.cont = this;
            agents.Add(agent);
```

```
            agent.visSegments = visionSegments;
            agent.CNNcompute = CNNcompute;
            if(useSLAM == true && preyCNT < preyCount / 2)
            {
                agent.controlMode = ControlMode.SLAM_integrated;
                agent.colorCode = PreyColorSLAM;
            }
        }

        //Time.timeScale (0);
        //Initialize ABM-scale map
        for (int iy = 0; iy < mapLength; iy++) {
            for (int ix = 0; ix < mapLength; ix++) {
                float rr = grassRegrowth * Time.deltaTime;
                float th = 0f;
                map[ix, iy] = new ABMMapTile(rr, th);
            }
        }
        //Init meshGenerator settings
        if (terrainGen != null)
        {
            tileLength = terrainGen.tileSize;
            terrainGen.tileCountX = mapLength;
            terrainGen.tileCountY = mapLength;
        }

        //Init GUI
        UpdateMiniMap();
        SimStart();
    }

void UpdateMetrics()
{
        if (iterationCount >= 100000 || (globalPredPopulationCount == 0
            && iterationCount > 10) || (globalPreyPopulationCount == 0
            && iterationCount > 10))
        {
            writeOut();
            Debug.Log("writeout");
            SceneManager.LoadScene (SceneManager.GetActiveScene().buildIndex);
            // Debug.Break();
            DestroyImmediate(this.gameObject);
```

```
        }

        iterationCount++;
 string s = "";
 s += "iteration count: " + iterationCount + "\n";
 s += ("Population Count: " + globalPopulationCount + "("
           + globalPreyPopulationCount + " prey| "
           + globalPredPopulationCount + " pred)\n");

 popCountHistory.Add (globalPopulationCount);
 globalPopulationCount = 0;

        popPredCountHistory.Add(globalPredPopulationCount);
        globalPredPopulationCount = 0;
        popPredSLAMCountHistory.Add(globalPredSLAMPopulationCount);
        globalPredSLAMPopulationCount = 0;

        popPreyCountHistory.Add(globalPreyPopulationCount);
        globalPreyPopulationCount = 0;
        popPreySLAMCountHistory.Add(globalPreySLAMPopulationCount);
        globalPreySLAMPopulationCount = 0;


        metricReadoutObj.text = s;

 preyChart.updateTexture (popPreyCountHistory,
           popPreyCountHistory.Count);
 predChart.updateTexture (popPredCountHistory,
           popPredCountHistory.Count);
 preyDChart.updateTexture (popPreySLAMCountHistory,
           popPreySLAMCountHistory.Count);
 predDChart.updateTexture (popPredSLAMCountHistory,
           popPredSLAMCountHistory.Count);
}

void UpdateMiniMap()
{
 Texture2D texture = miniMapImage;
 miniMap.GetComponent<Renderer>().material.mainTexture = miniMapImage;
 for (int iy = 0; iy < texture.height; iy++)
 {
  for (int ix = 0; ix < texture.width; ix++)
```

```
  {
   float fvalue = map [ix,iy].getMinMapTileColor ();
   Color color = grassColorFood.Evaluate (fvalue);
   //color = ((ix & iy) != 0 ? Color.white : Color.gray);
   texture.SetPixel(ix, iy, color);
  }
 }
 for(int ii = 0; ii < agents.Count; ii++) {
  AgentABM agent = agents [ii];
  if (!agent.isDead) {
   if (agent.isPred) {
                    if (agent.controlMode == ControlMode.SLAM_integrated)
                    {
                        texture.SetPixel(agent.ABMPosition.x,
                            agent.ABMPosition.y, PredColorSLAM);
                    }
                    else
                    {
                        texture.SetPixel(agent.ABMPosition.x,
                            agent.ABMPosition.y, PredColor);
                    }
   } else {
                    if (agent.controlMode == ControlMode.SLAM_integrated)
                    {
                        texture.SetPixel(agent.ABMPosition.x,
                            agent.ABMPosition.y, PreyColorSLAM);
                    }
                    else
                    {
                        texture.SetPixel(agent.ABMPosition.x,
                            agent.ABMPosition.y, PreyColor);
                    }
   }
  } else {
   texture.SetPixel (agent.ABMPosition.x, agent.ABMPosition.y,
                    deadColor);
  }
 }
 texture.Apply();
}

void SimStart()
```

```
{

}

void Update()
{
 if (singleton == this) {
  UpdateStep ();
 }
}

void UpdateStep () {

       UpdateChosen();
 List<AgentABM> newList = new List<AgentABM> ();
       for (int ii = 0; ii < agents.Count; ii++) {
           AgentABM agent = agents[ii];
           agent.Act();
           if (agent.isDead) {
               agent.deadframes--;
           }
           if (agent.deadframes > 0) {
               newList.Add(agent);
           }
           else
           {
               Destroy(agent.dummyBody);
           }
 }
 for (int xx = 0; xx < mapLength; xx++) {
  for (int yy = 0; yy < mapLength; yy++) {
   map [xx,yy].tileUpdate ();
  }
 }
       UpdateMetrics ();
       UpdateMiniMap();
       UpdateOverworld();
       agents = newList;

}

public static List<AgentABM> getAgentsNear(Vector2Int position)
```

```
{
 ABMController controller = getABM ();
 List<AgentABM> rList = new List<AgentABM> ();
 foreach (AgentABM agent in controller.agents)
 {
  if ((agent.ABMPosition.x == position.x
              && (Mathf.Abs(agent.ABMPosition.y - position.y) < 2))
              || (agent.ABMPosition.y == position.y
              && (Mathf.Abs(agent.ABMPosition.x - position.x) < 2)))
  {
   rList.Add (agent);
  }
 }
 return rList;
}

public Neighbourhood giveNeighbourhood(int xPos, int yPos)
{
 Neighbourhood rVal = new Neighbourhood (agentABMVisionDist);
       for (int ix = 0; ix < (2 * agentABMVisionDist) + 1; ix++)
       {
           for (int iy = 0; iy < (2 * agentABMVisionDist) + 1; iy++)
           {
               int globalX = (mapLength+(ix + (xPos - agentABMVisionDist)))
                   % mapLength;
               int globalY = (mapLength+(iy + (yPos - agentABMVisionDist)))
                   % mapLength;
               rVal.map[ix, iy] = map[globalX, globalY];
           }
       }

       foreach (AgentABM agent in agents)
       {
           if (Mathf.Abs(agent.ABMPosition.x - xPos) < 3
               && Mathf.Abs(agent.ABMPosition.y - yPos) < 3)
           {
               rVal.otherAgents.Add(agent);
           }
       }
 return rVal;
}
```

```
 public static ABMController getABM()
 {  return singleton; }

}
```

HeightmapWorld.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class HeightmapWorld : MonoBehaviour
{

    public int subdivisions = 32;
    public float tileSize = 10;
    public NoiseGenerator noise;

    public Material grassMat;

    public int tileCountX = 100;
    public int tileCountY = 100;

    private List<GameObject> currchunks;
    public float getHeightAt(float xx, float yy)
    {
        return noise.getHeight(xx, yy);
    }

    public void generateWorld()
    {
        for (int ii = 0; currchunks!= null && ii < currchunks.Count; ii++)
        {
            Destroy(currchunks[ii]);
        }
        currchunks = new List<GameObject>();

        for (int cx = -tileCountX / 2; cx < tileCountX / 2; cx++)
        {
            for (int cy = -tileCountY / 2; cy < tileCountY / 2; cy++)
            {
```

```
                GameObject newChunkGO = new GameObject("Chunk_(" + cx
                    + ", " + cy + ")");
                newChunkGO.AddComponent<MeshFilter>();
                newChunkGO.AddComponent<MeshRenderer>();
                newChunkGO.GetComponent<MeshRenderer>().material = grassMat;
                newChunkGO.AddComponent<MeshCollider>();
                HMChunk newChunk = newChunkGO.AddComponent<HMChunk>();
                newChunk.chunkCord = new Vector2Int(cx, cy);
                newChunk.setNoise(noise);
                newChunk.tileSize = tileSize;
                newChunk.subdivisions = subdivisions;
                newChunk.updateMesh();
                currchunks.Add(newChunkGO);
            }
        }
    }
}
```

ABMMapTile.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ABMMapTile
{

    public float regrowthRate = 0.1f;

    public float maxFood = 1f;
    public float currFood = 0.5f;

    public float elevation = 0.5f;

    public ABMMapTile(float regrowthRate, float elevation)
    {
        this.regrowthRate = regrowthRate;
        this.elevation = elevation;
    }

    public float getMinMapTileColor()
    {
```

```
        return currFood / maxFood;
    }

    public void tileUpdate()
    {
        currFood += (regrowthRate);
        if (currFood > maxFood)
        {
            currFood = maxFood;
        }
    }

    public float isEaten()
    {
        return isEaten(0.333f);
    }

    public float isEaten(float greedIndex)
    {
        float availableFood = currFood * greedIndex;
        currFood -= availableFood;
        return availableFood;
    }
}
```

AgentABM.cs

```
using System.Collections;
using System.Collections.Generic;
using System.IO;
using UnityEngine;
using System.Runtime.InteropServices;

public class AgentABM{
    private static int agentCount = 0;
    public int agentID = -1;
 public float metabolicEnergy = 5f;
 public Vector2Int ABMPosition = new Vector2Int();
 public float mutationIndex = 0.1f;
 public float reproThreshhold = 20f;
 public float greed = 0.3f;
 public int breedDelay = 100;
```

```
private int lastBread = 0;
public int deadframes = 60;
public bool isDead = false;
public bool isPred = false;
public float generationCost=0.02f;
   public GameObject dummyBody;
   public Camera dummyCam;
   public ABMController cont;
   public Color colorCode;
   public int visSegments = 5;
   public bool isChosen = false;

   public ControlMode controlMode;

   private Neighbourhood neighbourhood;
   public AgentDecisionNetwork brain;

   RenderTexture viewPoint;

   private Vector3 currentWorldPosition;
   private float currentWorldRotation;
   public ComputeShader CNNcompute;
   public SLAMModel slamModel = null;

   public static int nextID()
   {
       agentCount++;
       return agentCount;
   }


   public void CallCNN(Texture2D texture, string dirPath)
   {
       int kernelHandle = CNNcompute.FindKernel("CSMain");
       //layer1
       RenderTexture featuremap = new RenderTexture(240 - 2, 160 - 2, 24);
       RenderTexture featuremapb = new RenderTexture(240 - 2, 160 - 2, 24);
       RenderTexture poolingText = new RenderTexture((int)(240 - 2) / 3,
           (int)(160 - 2) / 3, 24);
       RenderTexture poolingTextb = new RenderTexture((int)(240 - 2) / 3,
           (int)(160 - 2) / 3, 24);
       //layer2
```

```
RenderTexture featuremap2Text = new RenderTexture(((int)(240 - 2) / 3)
    - 2, ((int)(160 - 2) / 3) - 2, 24);
RenderTexture pooling2Text = new RenderTexture((int)((
    (int)(240 - 2) / 3) - 2) / 3, (int)(((int)(160 - 2) / 3)
    - 2) / 3, 24);
RenderTexture featuremap3Text = new RenderTexture((
    (int)(240 - 2) / 3) - 2, ((int)(160 - 2) / 3) - 2, 24);
RenderTexture pooling3Text = new RenderTexture((int)((
    (int)(240 - 2) / 3) - 2) / 3, (int)((
    (int)(160 - 2) / 3) - 2) / 3, 24);
RenderTexture featuremap4Text = new RenderTexture(((int)(240 - 2) / 3)
    - 2, ((int)(160 - 2) / 3) - 2, 24);
RenderTexture pooling4Text = new RenderTexture((int)(((int)(240 - 2)
    / 3) - 2) / 3, (int)(((int)(160 - 2) / 3) - 2) / 3, 24);
RenderTexture featuremap5Text = new RenderTexture(((int)(240 - 2)
    / 3) - 2, ((int)(160 - 2) / 3) - 2, 24);
RenderTexture pooling5Text = new RenderTexture((int)(((int)
    (240 - 2) / 3) - 2) / 3, (int)(((int)(160 - 2) / 3) - 2) / 3, 24);
RenderTexture featuremap6Text = new RenderTexture(((int)(240 - 2) / 3)
    - 2, ((int)(160 - 2) / 3) - 2, 24);
RenderTexture pooling6Text = new RenderTexture((int)(((int)(240 - 2)
    / 3) - 2) / 3, (int)(((int)(160 - 2) / 3) - 2) / 3, 24);
RenderTexture featuremap7Text = new RenderTexture(((int)(240 - 2) / 3)
    - 2, ((int)(160 - 2) / 3) - 2, 24);
RenderTexture pooling7Text = new RenderTexture((int)(((int)(240 - 2)
    / 3) - 2) / 3, (int)(((int)(160 - 2) / 3) - 2) / 3, 24);
//layer1
featuremap.enableRandomWrite = true;
featuremap.Create();
featuremapb.enableRandomWrite = true;
featuremapb.Create();
poolingText.enableRandomWrite = true;
poolingText.Create();
poolingTextb.enableRandomWrite = true;
poolingTextb.Create();
//layer2
featuremap2Text.enableRandomWrite = true;
featuremap2Text.Create();
pooling2Text.enableRandomWrite = true;
pooling2Text.Create();
featuremap3Text.enableRandomWrite = true;
featuremap3Text.Create();
```

```
pooling3Text.enableRandomWrite = true;
pooling3Text.Create();
featuremap4Text.enableRandomWrite = true;
featuremap4Text.Create();
pooling4Text.enableRandomWrite = true;
pooling4Text.Create();
featuremap5Text.enableRandomWrite = true;
featuremap5Text.Create();
pooling5Text.enableRandomWrite = true;
pooling5Text.Create();
featuremap6Text.enableRandomWrite = true;
featuremap6Text.Create();
pooling6Text.enableRandomWrite = true;
pooling6Text.Create();
featuremap7Text.enableRandomWrite = true;
featuremap7Text.Create();
pooling7Text.enableRandomWrite = true;
pooling7Text.Create();
//Set texutres
CNNcompute.SetTexture(kernelHandle, "ImageInput", texture);
//layer1
CNNcompute.SetTexture(kernelHandle, "featureMap", featuremap);
CNNcompute.SetTexture(kernelHandle, "featureMapb", featuremapb);
CNNcompute.SetTexture(kernelHandle, "poolingMap", poolingText);
CNNcompute.SetTexture(kernelHandle, "poolingMapb", poolingTextb);
//layer2
CNNcompute.SetTexture(kernelHandle, "featureMap2", featuremap2Text);
CNNcompute.SetTexture(kernelHandle, "poolingMap2", pooling2Text);
CNNcompute.SetTexture(kernelHandle, "featureMap3", featuremap3Text);
CNNcompute.SetTexture(kernelHandle, "poolingMap3", pooling3Text);
CNNcompute.SetTexture(kernelHandle, "featureMap4", featuremap4Text);
CNNcompute.SetTexture(kernelHandle, "poolingMap4", pooling4Text);
CNNcompute.SetTexture(kernelHandle, "featureMap5", featuremap5Text);
CNNcompute.SetTexture(kernelHandle, "poolingMap5", pooling5Text);
CNNcompute.SetTexture(kernelHandle, "featureMap6", featuremap6Text);
CNNcompute.SetTexture(kernelHandle, "poolingMap6", pooling6Text);
CNNcompute.SetTexture(kernelHandle, "featureMap7", featuremap7Text);
CNNcompute.SetTexture(kernelHandle, "poolingMap7", pooling7Text);

//Set Kernals
ComputeBuffer kernal = new ComputeBuffer(10,
    Marshal.SizeOf(typeof(System.Single)), ComputeBufferType.Default);
```

```
kernal.SetData(new float[] { -1f, -1f, -1f, -1f, 8f, -1f, -1f,
    -1f, -1f, 1f });//Testing Line finder
CNNcompute.SetBuffer(kernelHandle, "KernalBuffer", kernal);

ComputeBuffer kernalb = new ComputeBuffer(10,
    Marshal.SizeOf(typeof(System.Single)), ComputeBufferType.Default);
kernalb.SetData(new float[] { -1f, -1f, -1f, -1f, 8f, -1f, -1f, -1f,
    -1f, 1f });//Testing Line finder
CNNcompute.SetBuffer(kernelHandle, "KernalBufferb", kernalb);

ComputeBuffer kernal2 = new ComputeBuffer(10,
    Marshal.SizeOf(typeof(System.Single)), ComputeBufferType.Default);
kernal2.SetData(new float[] { 1f, 1f, 1f, 1f, 1f, 1f, 1f, 1f, 1f, 1f /
    9f });//Testing Gausian Blur
CNNcompute.SetBuffer(kernelHandle, "KernalBuffer2", kernal2);

ComputeBuffer kernal3 = new ComputeBuffer(10,
    Marshal.SizeOf(typeof(System.Single)), ComputeBufferType.Default);
kernal3.SetData(new float[] { 0f, 0f, 0f, 0f, 1f, 0f, 0f, 0f, 0f,
    1f});//Testing identity
CNNcompute.SetBuffer(kernelHandle, "KernalBuffer3", kernal3);

ComputeBuffer kernal4 = new ComputeBuffer(10,
    Marshal.SizeOf(typeof(System.Single)), ComputeBufferType.Default);
kernal4.SetData(new float[] { 1f, 1f, 1f, 1f, 1f, 1f, 1f, 1f, 1f, 1f /
    9f });//Testing Gausian Blur
CNNcompute.SetBuffer(kernelHandle, "KernalBuffer4", kernal4);

//Dispatch Kernal
CNNcompute.Dispatch(kernelHandle, 512 / 8, 1024 / 8, 1);

//Write output images
RenderTexture.active = featuremap;
Texture2D featuremapTexture = new Texture2D(240 - 4, 160 - 4);
featuremapTexture.ReadPixels(new Rect(0, 0, featuremap.width,
    featuremap.height), 0, 0);
featuremapTexture.Apply();

byte[] imageBytes = featuremapTexture.EncodeToPNG();
File.WriteAllBytes(dirPath + "_featuremap_1" + ".png", imageBytes);
//*
RenderTexture.active = poolingText;
```

```
Texture2D poolingTextTexture = new Texture2D((int)(240 - 2) / 3, (int)
    (160 - 2) / 3);
poolingTextTexture.ReadPixels(new Rect(0, 0, poolingText.width,
    poolingText.height), 0, 0);
poolingTextTexture.Apply();

imageBytes = poolingTextTexture.EncodeToPNG();
File.WriteAllBytes(dirPath + "_featuremap_1_pool" + ".png",
    imageBytes);
//
RenderTexture.active = pooling2Text;
Texture2D poolingText2Texture = new Texture2D((int)(((int)(240 - 2) /
    3) - 2) / 3, (int)(((int)(160 - 2) / 3) - 2) / 3);
poolingText2Texture.ReadPixels(new Rect(0, 0, pooling2Text.width,
    pooling2Text.height), 0, 0);
poolingText2Texture.Apply();

imageBytes = poolingText2Texture.EncodeToPNG();
File.WriteAllBytes(dirPath + "_featuremap_2_pool" + ".png",
    imageBytes);
//
RenderTexture.active = featuremap2Text;
Texture2D featureText2Texture = new Texture2D((int)(240 - 2) / 3,
    (int)(160 - 2) / 3);
featureText2Texture.ReadPixels(new Rect(0, 0, featuremap2Text.width,
    featuremap2Text.height), 0, 0);
featureText2Texture.Apply();

imageBytes = featureText2Texture.EncodeToPNG();
File.WriteAllBytes(dirPath + "_featuremap_2" + ".png", imageBytes);
//
RenderTexture.active = pooling6Text;
Texture2D poolingText6Texture = new Texture2D((int)(((int)(240 - 2) /
    3) - 2) / 3, (int)(((int)(160 - 2) / 3) - 2) / 3);
poolingText6Texture.ReadPixels(new Rect(0, 0, pooling2Text.width,
    pooling2Text.height), 0, 0);
poolingText6Texture.Apply();

imageBytes = poolingText6Texture.EncodeToPNG();
File.WriteAllBytes(dirPath + "_featuremap_6_pool" + ".png",
    imageBytes);
//
```

```
        RenderTexture.active = featuremap6Text;
        Texture2D featureText6Texture = new Texture2D((int)(240 - 2) / 3,
            (int)(160 - 2) / 3);
        featureText6Texture.ReadPixels(new Rect(0, 0, featuremap6Text.width, featu
        featureText6Texture.Apply();

        imageBytes = featureText6Texture.EncodeToPNG();
        File.WriteAllBytes(dirPath + "_featuremap_6" + ".png", imageBytes);
        //*/

    }

    public List<AgentABM> getObservableAgents()
    {
        List<AgentABM> visableAgents = cont.VisableAgents(
            GeometryUtility.CalculateFrustumPlanes(dummyCam));
        List<AgentABM> observableAgents = new List<AgentABM>();
        foreach (AgentABM agent in visableAgents) if (agent.agentID != agentID)
            {
                Bounds bounds =
                    agent.dummyBody.GetComponent<Collider>().bounds;

                Rect viewBox = GetBoundingBoxOnScreen2(bounds, dummyCam);
                Ray losRay = dummyCam.ScreenPointToRay(viewBox.center);
                RaycastHit losHit;
                if (Physics.Raycast(losRay, out losHit, 1000f)) if
                    (losHit.collider.gameObject == agent.dummyBody.gameObject)
                {
                        if (!agent.isDead)
                        { ///Agent is visavble and living
                            observableAgents.Add(agent);
                        }
                }
            }
        return observableAgents;
    }

    public void move(Vector2Int delta)
    {
        int maplength = ABMController.getABM().mapLength;
        Vector3 oldPosition = cont.tileToWorldPoint(ABMPosition)
            + (Vector3.up * 0.65f);
```

```
ABMPosition += delta;
ABMPosition.x = (ABMPosition.x + maplength) % maplength;
ABMPosition.y = (ABMPosition.y + maplength) % maplength;
Vector3 newPosition = cont.tileToWorldPoint(ABMPosition)
    + (Vector3.up *0.65f);
float oldRotation = currentWorldRotation;
float newRotation = oldRotation;
if (delta.x == 1)
{
    newRotation = 270+180;
}
else if (delta.y == -1)
{
    newRotation = 0 + 180;
}
else if (delta.x == -1)
{
    newRotation = 90 + 180;
}
else if (delta.y == 1)
{
    newRotation = 180 + 180;
}


if (isChosen)
{//*
 //Interpolate and generate visual data
    dummyCam.gameObject.SetActive(true);
    for (float ratio = 0f; ratio <= 1f; ratio += (1f
        / (float)visSegments))
    {
        Vector3 interPos = Vector3.Lerp(oldPosition, newPosition,
            ratio);
        float interAngle = Mathf.LerpAngle(oldRotation, newRotation,
            ratio);
        if (viewPoint == null)
        {
            viewPoint = new RenderTexture(240, 160, 16);
            dummyCam.targetTexture = viewPoint;
        }
        interPos.y = cont.terrainGen.getHeightAt(interPos.x,
```

```
    interPos.z);
dummyBody.gameObject.transform.position = interPos;
dummyBody.transform.rotation = Quaternion.Euler(0f,
    interAngle, 0f);
dummyCam.Render();
Texture2D viewPoint2D = toTexture2D(viewPoint);

string dirPath = Application.dataPath + "/../SaveImages/agent"
    + agentID.ToString() + "/";
if (!Directory.Exists(dirPath))
{
    Directory.CreateDirectory(dirPath);
}

//Write metadata
List<AgentABM> visableAgents =
    cont.VisableAgents(GeometryUtility.CalculateFrustumPlanes(
    dummyCam));
List<AgentABM> observableAgents = new List<AgentABM>();
string metaData = "Global Position: " +
    this.dummyBody.transform.position.ToString() + "\n";
metaData += "Global Rotation: " +
    this.dummyBody.transform.eulerAngles.ToString() + "\n";
metaData += "Energy: " + this.metabolicEnergy + "\n\n";
foreach (AgentABM agent in visableAgents)
    if (agent.agentID != agentID)
    {
        Bounds bounds =
            agent.dummyBody.GetComponent<Collider>().bounds;
        //Rect viewBox =
            GetBoundingBoxOnScreen(
            agent.dummyBody.GetComponent<MeshFilter>().mesh,
            dummyCam);
        Rect viewBox = GetBoundingBoxOnScreen2(bounds,
            dummyCam);
        Ray losRay = dummyCam.ScreenPointToRay(viewBox.center);
        RaycastHit losHit;
        if (Physics.Raycast(losRay, out losHit, 1000f))
            if (losHit.collider.gameObject ==
                agent.dummyBody.gameObject)
        {
            observableAgents.Add(agent);
```

```
                    metaData += "Agent: " + agent.agentID + "\n";
                    if (agent.isPred) { metaData += "Type:
                        Predator\n"; }
                    else { metaData += "Type: Prey\n"; }
                    metaData += "ViewBox:(" + viewBox.ToString() +
                        ")\n";
                    metaData += "Bounds:" + bounds.ToString() + "\n";
                    if (agent.isDead)
                    { metaData += "Status: Dead\n"; }
                    else
                    { metaData += "Status: Alive\n"; }
                    Vector2Int screenSize = new
                        Vector2Int(viewPoint2D.width,
                        viewPoint2D.height);

                    Vector2 min = viewBox.min;
                    Vector2 max = viewBox.max;
                    if (ABMController.getABM().drawBoundingBox)
                    {
                        for (int xx = (int)min.x; xx <
                            (int)max.x; xx++)
                        {
                            int yy = (int)(min.y);
                            viewPoint2D.SetPixel(xx, yy, Color.red);
                            yy = (int)(max.y);
                            viewPoint2D.SetPixel(xx, yy, Color.yellow);
                        }
                        for (int yy = (int)min.y; yy < (int)max.y;
                            yy++)
                        {
                            int xx = (int)(min.x);
                            viewPoint2D.SetPixel(xx, yy, Color.blue);
                            xx = (int)(max.x);
                            viewPoint2D.SetPixel(xx, yy, Color.green);
                        }
                    }
                }
            }
        //if (ABMController.getABM().usingCNN)
        //{
        //    CallCNN(viewPoint2D, (dirPath + cont.iterationCount
            + "(" + ratio + ")"));
```

```
            //}
            viewPoint2D.Apply();
            byte[] imageBytes = viewPoint2D.EncodeToPNG();
            File.WriteAllBytes(dirPath + cont.iterationCount + "("
                + ratio + ")" + ".png", imageBytes);
            File.WriteAllText(dirPath + cont.iterationCount + "("
                + ratio + ")_meta" + ".txt", metaData);
        }
        dummyCam.gameObject.SetActive(false);
        //*/
    }
    dummyBody.gameObject.transform.position = newPosition;
    dummyBody.transform.rotation =
        Quaternion.Euler(dummyBody.transform.eulerAngles.x,
        newRotation, dummyBody.transform.eulerAngles.z);
    //Update exact position variable
    currentWorldPosition = newPosition;
    currentWorldRotation = newRotation;
}

public Rect GetBoundingBoxOnScreen2(Bounds bounds, Camera camera)
{
    Vector3 max = camera.WorldToScreenPoint(bounds.max);
    Vector3 min = camera.WorldToScreenPoint(bounds.min);
    Rect retVal = Rect.MinMaxRect(min.x, min.y, max.x, max.y);

    return retVal;
}

    public Rect GetBoundingBoxOnScreen(Mesh mesh, Camera camera)
{
    //https://gamedev.stackexchange.com/questions/187446/unity-how-to-get
    -the-visible-bounds-of-an-object-i-e-as-it-is-seen-from-the-c
    List<Vector3> vertices = new List<Vector3>();
    mesh.GetVertices(vertices);

    Rect retVal = Rect.MinMaxRect(float.MaxValue, float.MaxValue,
        float.MinValue, float.MinValue);

    for (int i = 0; i < vertices.Count; i++)
    {
        Vector3 v = camera.WorldToScreenPoint(vertices[i]);
```

```
            if (v.x < retVal.xMin)
            {
                retVal.xMin = v.x;
            }

            if (v.y < retVal.yMin)
            {
                retVal.yMin = v.y;
            }

            if (v.x > retVal.xMax)
            {
                retVal.xMax = v.x;
            }

            if (v.y > retVal.yMax)
            {
                retVal.yMax = v.y;
            }
        }

        return retVal;
    }


    Texture2D toTexture2D(RenderTexture rTex)
    {
        Texture2D tex = new Texture2D(240, 160, TextureFormat.RGB24, false);
        // ReadPixels looks at the active RenderTexture.
        RenderTexture.active = rTex;
        tex.ReadPixels(new Rect(0, 0, rTex.width, rTex.height), 0, 0);
        tex.Apply();
        return tex;
    }

    public void Act(){
        if(agentID < 0)
        {
            agentID = nextID();
        }
        if (dummyCam == null && dummyBody != null)
        {
```

```
            dummyCam = dummyBody.GetComponentInChildren<Camera>();
            dummyCam.gameObject.SetActive(false);
        }
  if (IsHealthy ()) {
   applyUpkeepCost ();
            //View ABM
            neighbourhood = getControlNeighbourhood();
            //Control Unit
            if (controlMode == ControlMode.RandomMove)
            {
                eat();
                float val = Random.value;
                if (val < 0.25f)
                {
                    move(new Vector2Int(0, 1));
                }
                else if (val < 0.5f)
                {
                    move(new Vector2Int(1, 0));
                }
                else if (val < 0.75f)
                {
                    move(new Vector2Int(0, -1));
                }
                else
                {
                    move(new Vector2Int(-1, 0));
                }
                if (metabolicEnergy > reproThreshhold)
                {
                    if (lastBread > 0)
                    {
                        lastBread--;
                    }
                    else
                    {
                        reproduce();
                    }
                }
            }
            else if (controlMode == ControlMode.SLAM_integrated)
            {
```

```
//get closest agent
List<AgentABM> agents = getObservableAgents();
AgentABM closest = null;
int DistanceMH = int.MaxValue;
foreach (AgentABM agent in agents)
{
    int distance = Mathf.Abs(agent.ABMPosition.x -
        ABMPosition.x) + Mathf.Abs(agent.ABMPosition.y -
        ABMPosition.y);
    if (distance < DistanceMH && agent != this)
    {
        DistanceMH = distance;
        closest = agent;
    }
}
//update SLAM module
if (slamModel == null)
{
    slamModel = new SLAMModel(100);
}
if (closest != null)
{
    slamModel.reWeight(closest.ABMPosition);
}
//Resample
slamModel.reSample();
//Predict
slamModel.extrapolate();
//If (any nearby spot is > 0 probabillity)
//     Move towards min probabillity
//else(
float[] values = new float[4];
values[0] = slamModel.querry(ABMPosition
    + new Vector2Int(0, 1));
values[1] = slamModel.querry(ABMPosition
    + new Vector2Int(1, 0));
values[2] = slamModel.querry(ABMPosition
    + new Vector2Int(0, -1));
values[3] = slamModel.querry(ABMPosition
    + new Vector2Int(-1, 0));
int choice = -1;
float smallest = float.MaxValue;
```

```
float largest = float.MinValue;
for (int ii = 0; ii < values.Length; ii++)
{
    float value = values[ii];
    if (value < smallest)
    {
        choice = ii;
        smallest = value;
    }
    if(value > largest)
    {
        largest = value;
    }
}
eat();
if (largest < 0.1f)//If no threat, move randomly
{
    float val2 = Random.value;
    if (val2 < 0.25f)
    {
        move(new Vector2Int(0, 1));
    }
    else if (val2 < 0.5f)
    {
        move(new Vector2Int(1, 0));
    }
    else if (val2 < 0.75f)
    {
        move(new Vector2Int(0, -1));
    }
    else
    {
        move(new Vector2Int(-1, 0));
    }
}
else if (choice == 0)//Make an informed decision
{
    move(new Vector2Int(0, 1));
}
else if (choice == 1)
{
    move(new Vector2Int(1, 0));
```

```
        }
        else if (choice == 2)
        {
            move(new Vector2Int(0, -1));
        }
        else
        {
            move(new Vector2Int(-1, 0));
        }
        if (metabolicEnergy > reproThreshhold)
        {
            if (lastBread > 0)
            {
                lastBread--;
            }
            else
            {
                reproduce();
            }
        }

        //end of controlmode
    }
    else if (controlMode == ControlMode.BrainMove)
    {
        float[] output = brain.calculateZone(neighbourhood, this);
        int choice = -1;
        float highest = float.MinValue;
        for (int choiceII = 0; choiceII < output.Length; choiceII++)
        {
            if (output[choiceII] > highest)
            {
                highest = output[choiceII];
                choice = choiceII;
            }
        }
        eat();
        if (lastBread > 0)
        {
            lastBread--;
        }
        if (choice == 0)
```

```
{
    if (lastBread <= 0)
    {
        reproduce();
    }
}
else if (choice == 1)
{
    float val = Random.value;
    if (val < 0.25f)
    {
        move(new Vector2Int(0, 1));
    }
    else if (val < 0.5f)
    {
        move(new Vector2Int(1, 0));
    }
    else if (val < 0.75f)
    {
        move(new Vector2Int(0, -1));
    }
    else
    {
        move(new Vector2Int(-1, 0));
    }
}
else if (choice == 2)
{
    move(new Vector2Int(0, 1));
}
else if (choice == 3)
{
    move(new Vector2Int(1, 0));
}
else if (choice == 4)
{
    move(new Vector2Int(0, -1));
}
else if (choice == 5)
{
    move(new Vector2Int(-1, 0));
}
```

```
}
else if (controlMode == ControlMode.BrainMove2)
{
    float[] output = brain.calculateZone(neighbourhood, this);
    int choice = -1;
    float highest = float.MinValue;
    for (int choiceII = 0; choiceII < output.Length; choiceII++)
    {
        if (output[choiceII] > highest)
        {
            highest = output[choiceII];
            choice = choiceII;
        }
    }
    eat();
    if (lastBread > 0)
    {
        lastBread--;
    }
    if (metabolicEnergy > reproThreshhold)
    {
        if (lastBread > 0)
        {
            lastBread--;
        }
        else
        {
            reproduce();
        }
    }
    if (choice == 1)
    {
        float val = Random.value;
        if (val < 0.25f)
        {
            move(new Vector2Int(0, 1));
        }
        else if (val < 0.5f)
        {
            move(new Vector2Int(1, 0));
        }
```

```
                else if (val < 0.75f)
                {
                    move(new Vector2Int(0, -1));
                }
                else
                {
                    move(new Vector2Int(-1, 0));
                }
            }
            else if (choice == 2)
            {
                move(new Vector2Int(0, 1));
            }
            else if (choice == 3)
            {
                move(new Vector2Int(1, 0));
            }
            else if (choice == 4)
            {
                move(new Vector2Int(0, -1));
            }
            else if (choice == 5)
            {
                move(new Vector2Int(-1, 0));
            }

        }

        //Update Statistics
        ABMController control = ABMController.getABM ();
control.globalPopulationCount++;
if (isPred) {
            if (controlMode == ControlMode.SLAM_integrated)
            {
                control.globalPredSLAMPopulationCount++;
            }
 control.globalPredPopulationCount++;
} else {
            if (controlMode == ControlMode.SLAM_integrated)
            {
                control.globalPreySLAMPopulationCount++;
            }
```

```
                control.globalPreyPopulationCount++;
   }

  }
  else {
   isDead = true;
            dummyBody.GetComponent<MeshRenderer>().material.color =
                new Color32(230, 0, 20, 255);
        }
}

    public Neighbourhood getControlNeighbourhood()
    {
        return ABMController.getABM().giveNeighbourhood(
            ABMPosition.x, ABMPosition.y);
    }


    public void eat ()
{
 if (isPred) {
  List<AgentABM> agents = ABMController.getAgentsNear (ABMPosition);
  bool eaten = false;
  for (int ii = 0; ii < agents.Count && !eaten; ii++) {
   if (agents [ii].isPred == false && agents [ii].isDead
                    == false) {
    eaten = true;
    metabolicEnergy += (agents [ii].metabolicEnergy)/2f;
    agents [ii].metabolicEnergy = 0;
    agents [ii].isDead = true;
   }
  }

 } else {
  float eatAmount = ABMController.getABM ().map [ABMPosition.x,
                ABMPosition.y].isEaten (greed);
  metabolicEnergy += eatAmount;
 }
}
```

```
public void reproduce()
{
 AgentABM newAgent = new AgentABM ();
 this.lastBread = this.breedDelay;
 newAgent.lastBread = this.breedDelay;
 newAgent.isPred = isPred;
 newAgent.ABMPosition = this.ABMPosition;
 newAgent.metabolicEnergy = this.metabolicEnergy/4f;
 this.metabolicEnergy = this.metabolicEnergy/2f;
       newAgent.generationCost = this.generationCost;
       newAgent.colorCode = colorCode;
       if (dummyBody != null)
       {
           dummyCam.gameObject.SetActive(true);
           GameObject newBody = GameObject.Instantiate(this.dummyBody);
           newAgent.dummyBody = newBody;
           newAgent.dummyBody.GetComponent<MeshRenderer>().material.color =
               colorCode;
           newAgent.dummyCam =
               newAgent.dummyBody.GetComponentInChildren<Camera>();
           newAgent.dummyCam.gameObject.SetActive(false);
           dummyCam.gameObject.SetActive(false);
       }
       newAgent.controlMode = this.controlMode;
 newAgent.breedDelay = this.breedDelay + (Random.Range (-5, 6));
 newAgent.mutationIndex = mutationIndex +
           ((Random.value-0.5f)*mutationIndex);
 newAgent.reproThreshhold = reproThreshhold +
           ((Random.value-0.5f)*mutationIndex);
       newAgent.greed = greed + ((Random.value - 0.5f) * mutationIndex);
       newAgent.cont = cont;
       newAgent.visSegments = visSegments;
       newAgent.CNNcompute = CNNcompute;
       if (newAgent.greed > 1) {
  newAgent.greed = 1;
 } else if (newAgent.greed < 0) {
  newAgent.greed = 0;
 }
       newAgent.brain =
           new AgentDecisionNetwork(this.brain, mutationIndex/10f);

       ABMController.getABM ().agents.Add (newAgent);
```

```
 }

 public bool IsHealthy()
 {
  return (metabolicEnergy > 0f && !isDead);
 }

 public void applyUpkeepCost()
 {
  metabolicEnergy -= generationCost;
 }

    public string filePrefix()
    {
        return ABMController.filePrefix + agentID.ToString() + ".txt";
    }
}

public class Neighbourhood
{
 public ABMMapTile[,] map;
 public List<AgentABM> otherAgents;

 public Neighbourhood(int radius)
 {
  map = new ABMMapTile[radius*2+1,radius*2+1];
  otherAgents = new List<AgentABM> ();
 }
}

public class SLAMModel
{
    private Vector2Int[] samples = new Vector2Int[100];
    private float[] sampleWeights = new float[100];
    private float directionChance = 0.25f; // 1/4 possible directions each step

    public SLAMModel(int sampleCount)
    {
        Vector2Int[] samples = new Vector2Int[sampleCount];
        float[] sampleWeights = new float[sampleCount];
        for (int ii = 0; ii < sampleCount; ii++)
        {
```

```
            sampleWeights[ii] = 1f / sampleCount;
            samples[ii] = new Vector2Int(Random.Range(0, 100),
                Random.Range(0, 100));//Initialized as random distribution
        }
    }

    public void reWeight(Vector2Int closestSeenAgentPos)
    {
        //Calculate weights given observation
        float totalWeight = 0f;
        for (int ii = 0; ii < samples.Length; ii++)
        {
            int manhattanDistance = Mathf.Abs(closestSeenAgentPos.x
                - samples[ii].x);
            manhattanDistance += Mathf.Abs(closestSeenAgentPos.y
                - samples[ii].y);
            sampleWeights[ii] = Mathf.Pow(directionChance, manhattanDistance);
            totalWeight += sampleWeights[ii];
        }
        //Renormalize Distribution
        float finaltotalWeight = 0f;
        for (int ii = 0; ii < sampleWeights.Length; ii++)
        {
            sampleWeights[ii] = sampleWeights[ii] / totalWeight;
            finaltotalWeight += sampleWeights[ii];
        }//Weights now will sum to 1
        //Debug.Log(finaltotalWeight);
    }

    public void reSample()
    {
        Vector2Int[] newSamples = new Vector2Int[samples.Length];
        float[] newWeights = new float[samples.Length];
        for (int ii = 0; ii < samples.Length; ii++)
        {
            float select = Random.value;
            bool found = false;
            int jj = 0;
            while (!found)
            {
                select = select - sampleWeights[jj];
                if (select > 0)
```

```
                {
                    found = true;
                }
                else
                {
                    jj++;
                    if (jj >= samples.Length)
                    {
                        found = true;
                        jj--;
                    }
                }
            }
            newSamples[ii] = samples[jj];
            newWeights[ii] = 1f / samples.Length;
        }
        samples = newSamples;
        sampleWeights = newWeights;
    }

    public void extrapolate()
    {
        for (int ii = 0; ii < samples.Length; ii++)
        {
            float select = Random.value;
            if (select > 0.5f)
                //randomly select a single direction to move prediction to.
            {
                if (select > 0.75f)
                {
                    samples[ii].y = samples[ii].y + 1;
                }
                else
                {
                    samples[ii].y = samples[ii].y - 1;
                }
            }
            else
            {
                if (select > 0.25f)
                {
                    samples[ii].x = samples[ii].x + 1;
```

```
                }
                else
                {
                    samples[ii].x = samples[ii].x - 1;
                }
            }
        }
    }

    public float querry(Vector2Int position)
    {
        float probabillity = 0f;
        foreach (Vector2Int sample in samples)
        {
            if (sample.x == position.x && sample.y == position.y)
            {
                probabillity++;
            }
        }
        return probabillity / samples.Length;
    }
}



[System.Serializable]
public class AgentDecisionNetwork
{
    public Neighbourhood input;

    public int[] shape = { 51, 20 , 20, 5 };
    public List<List<List<float>>> weights = new List<List<List<float>>>();

    public AgentDecisionNetwork()
    {
        initializeRandomWeights();
    }

    public AgentDecisionNetwork(AgentDecisionNetwork other,
        float mutationIndex)
    {
        for (int ii = 0; ii < other.shape.Length;ii++)
```

```
        {
            shape[ii] = other.shape[ii];
        }
        weights = new List<List<List<float>>>();
        for (int layerii = 0; layerii < shape.Length - 1; layerii++)
        {
            List<List<float>> layerWeights = new List<List<float>>();
            for (int nodeII = 0; nodeII < shape[layerii]; nodeII++)
            {
                List<float> nodeWeights = new List<float>();
                for (int nextII = 0; nextII < shape[layerii + 1]; nextII++)
                {
                    nodeWeights.Add(other.weights[layerii][nodeII][nextII]
                        + (Random.value - 0.5f) * mutationIndex);
                        //Random value ranging from -(mutationindex/2)
                        //to (mutationindex/2 )
                }
                layerWeights.Add(nodeWeights);
            }
            weights.Add(layerWeights);
        }
    }

    public void initializeRandomWeights()
    {
        weights = new List<List<List<float>>>();
        for (int layerii = 0; layerii < shape.Length-1; layerii++)
        {
            List<List<float>> layerWeights = new List<List<float>>();
            for (int nodeII = 0; nodeII < shape[layerii]; nodeII++)
            {
                List<float> nodeWeights = new List<float>();
                for(int nextII = 0; nextII < shape[layerii+1]; nextII++)
                {
                    nodeWeights.Add((Random.value - 0.5f) * 0.125f);
                    //Random value ranging from -0.25 to 0.25
                }
                layerWeights.Add(nodeWeights);
            }
            weights.Add(layerWeights);
        }
```

```
}

public float[] calculateZone(Neighbourhood neighbourhood, AgentABM self)
{
    List<float> rVal = new List<float>();
    //Set up input from neighbourhood object
    List<float> inputValues = new List<float>();
    inputValues.Add(self.metabolicEnergy);
    for (int ix = 0; ix < neighbourhood.map.GetLength(0); ix++)
    {
        for (int iy = 0; iy < neighbourhood.map.GetLength(1); iy++)
        {
            ABMMapTile tile = neighbourhood.map[ix, iy];
            if (tile == null)
            {
                Debug.Log("Error Tile(" + ix + "," + iy + ") set to null");
            }
            else
            {
                inputValues.Add(tile.currFood);
            }
        }
    }
    for (int ix = 0; ix < neighbourhood.map.GetLength(0); ix++)
    {
        for (int iy = 0; iy < neighbourhood.map.GetLength(1); iy++)
        {
            foreach(AgentABM agent in neighbourhood.otherAgents)
            {
                int foundVal = 0;
                if (agent.ABMPosition.x == ix && agent.ABMPosition.y
                    == iy && agent != self)
                {
                    foundVal = 1;
                    if (agent.isPred != self.isPred)
                    {
                        foundVal = -1;
                    }
                }
                inputValues.Add(foundVal);
            }
        }
```

```
        }
        //Evaluate each layer
        List<List<float>> values = new List<List<float>>();
        float[] pastValues = inputValues.ToArray();
        float[] nextValues = new float[0];
        for (int ii = 0; ii < shape.Length-1; ii++)
        {
            nextValues = new float[shape[ii+1]];
            for (int nodeII = 0; nodeII < shape[ii]; nodeII++)
            {
                for (int nextII = 0; nextII < nextValues.Length; nextII++)
                {
                    nextValues[nextII] += (pastValues[nodeII] * weights[ii]
                        [nodeII][nextII]);
                }
            }
            pastValues = nextValues;
        }
        return nextValues;
    }
}

public enum ControlMode
{
    RandomMove,
    BrainMove,
    BrainMove2,
    SLAM_integrated
};
```

ABMMetricChart.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ABMMetricChart : MonoBehaviour {
 public Vector2Int textureSize;

 public int boarderSpace = 5;

 private Texture2D graphTexture;
```

```
private Color[] blankPixels;

public int lineThickness;
private Color[] dot;

void Start()
{
 graphTexture = new Texture2D (textureSize.x, textureSize.y);
 graphTexture.filterMode = FilterMode.Point;

 dot = new Color[(lineThickness) * (lineThickness*2)];
 for(int ii = 0; ii < dot.Length; ii++) {
  dot[ii] = Color.black;
 }

 blankPixels = graphTexture.GetPixels ();
}

public void updateTexture(List<int> values, int maxCount)
{
 //First, reset the background to white
 GetComponent<Renderer>().material.mainTexture = graphTexture;
 graphTexture.SetPixels (blankPixels);
 int largestValue = int.MinValue;
 int smallestValue = int.MaxValue;
 foreach (int value in values) {
  if (value > largestValue) {
   largestValue = value;
  }
  if (value < smallestValue) {
   smallestValue = value;
  }
 }
 int xSpace = textureSize.x - (2 * boarderSpace);
 int ySpace = textureSize.y - (2 * boarderSpace);
 int limit = Mathf.Min (values.Count-1, maxCount);

 for (int px = boarderSpace; px < textureSize.x - boarderSpace; px++) {
  int index = (int)(((float)(px - boarderSpace) / (float)xSpace)
               * limit);
  int py = (int)(/*(ySpace + boarderSpace) - */( ((float)
               (values[index]-smallestValue) / ((float)largestValue
```

```
                -smallestValue)) * ySpace + boarderSpace) );

   graphTexture.SetPixel (px, py, Color.black);
   graphTexture.SetPixel (px+1, py+1, Color.black);
   graphTexture.SetPixel (px+1, py-1, Color.black);
   graphTexture.SetPixel (px-1, py+1, Color.black);
   graphTexture.SetPixel (px-1, py-1, Color.black);
  }

  graphTexture.Apply();
 }

 public void updateTexturef(List<float> values, int maxCount)
 {
  //First, reset the background to white
  GetComponent<Renderer>().material.mainTexture = graphTexture;
  graphTexture.SetPixels (blankPixels);
  float largestValue = float.MinValue;
  float smallestValue = float.MaxValue;
  foreach (float value in values) {
   if (value > largestValue) {
    largestValue = value;
   }
   if (value < smallestValue) {
    smallestValue = value;
   }
  }
  //Debug.Log (largestValue);
  int xSpace = textureSize.x - (2 * boarderSpace);
  int ySpace = textureSize.y - (2 * boarderSpace);
  int limit = Mathf.Min (values.Count-1, maxCount);

  for (int px = boarderSpace; px < textureSize.x - boarderSpace; px++) {
   int index = (int)(((float)(px - boarderSpace) /
                (float)xSpace) * limit);
   int py = (int)(/*(ySpace + boarderSpace) - */( ((float)
                (values[index]-smallestValue) / ((float)largestValue
                -smallestValue)) * ySpace + boarderSpace) );
   graphTexture.SetPixel (px, py, Color.black);
   graphTexture.SetPixel (px+1, py+1, Color.black);
   graphTexture.SetPixel (px+1, py-1, Color.black);
   graphTexture.SetPixel (px-1, py+1, Color.black);
```

```
   graphTexture.SetPixel (px-1, py-1, Color.black);
  }
  graphTexture.Apply();
 }


 public void updateTexture2(List<int> values, int maxCount)
 {
  //First, reset the background to white
  graphTexture = new Texture2D(textureSize.x, textureSize.y);
  graphTexture.filterMode = FilterMode.Point;
  GetComponent<Renderer>().material.mainTexture = graphTexture;
  int largestValue = int.MinValue;
  foreach (int value in values) {
   if (value > largestValue) {
    largestValue = value;
   }
  }

  int xSpace = textureSize.x - (2 * boarderSpace);
  int ySpace = textureSize.y - (2 * boarderSpace);
  int limit = Mathf.Min (values.Count, maxCount);

  for (int ix = 0; ix < limit; ix++) {
   int pixelX = (int)(((float)ix / (float)maxCount) * xSpace
                + boarderSpace);
   int pixelY = (int)(((float)values[ix]/(float)largestValue)
                * ySpace + boarderSpace);
   graphTexture.SetPixel (pixelX, pixelY, Color.black);
  }

  graphTexture.Apply();
 }
}
```

ComputeCNN.compute

```
    // Each #kernel tells which function to compile; you can have many kernels
#pragma kernel CSMain

// Create a RenderTexture with enableRandomWrite flag and set it
// with cs.SetTexture
```

```
RWTexture2D<float4> Result;
Texture2D<float4> ImageInput;

StructuredBuffer<float> KernalBuffer;

float2 flip;

//https://github.com/G4ND44/computeShaderBlur/blob/master/Assets
    /Shaders/boxBlur.compute
//https://www.youtube.com/watch?v=ub7JwtJjRSI
//https://github.com/Firnox/ShaderStories-EdgeDetection
[numthreads(8,8,1)]
void CSMain (uint3 id : SV_DispatchThreadID)
{
    float4 pixel = float4(0, 0, 0, 0);
    // Convolution: ID
    //   |  0  1  2  |
    // 9 |  3  4  5  |
    //   |  6  7  8  |
    pixel += KernalBuffer[0] * ImageInput[id.xy - int2(-1, 1)];
    pixel += KernalBuffer[1] * ImageInput[id.xy - int2(0, 1)];
    pixel += KernalBuffer[2] * ImageInput[id.xy - int2(1, 1)];

 pixel += KernalBuffer[3] * ImageInput[id.xy - int2(-1, 0)];
 pixel += KernalBuffer[4] * ImageInput[id.xy];
    pixel += KernalBuffer[5] * ImageInput[id.xy - int2(1, 0)];

 pixel += KernalBuffer[6] * ImageInput[id.xy - int2(-1, -1)];
    pixel += KernalBuffer[7] * ImageInput[id.xy - int2(0, -1)];
    pixel += KernalBuffer[8] * ImageInput[id.xy - int2(1, -1)];

 pixel *= KernalBuffer[9];
    // Retain the original pixel alpha value.
    pixel.a = ImageInput[id.xy].a;
    // Set the calculate pixel
    Result[id.xy] = pixel;
}
```

HMChunk.cs (HeightMapChunk)

```
using System.Collections;
using System.Collections.Generic;
```

```
using UnityEngine;

public class HMChunk : MonoBehaviour
{
    public float avgHeight = float.MinValue;
    public float maxHeight = float.MinValue;
    public float minHeight = float.MaxValue;
    public Vector2Int chunkCord = Vector2Int.zero;
    public int subdivisions = 32;
    public float tileSize = 10;


    public NoiseGenerator noise;

    public Mesh mesh;
    public List<Vector3> vertices;
    public List<Vector2> uvs;
    public List<int> trigs;

    private void Start()
    {
        //updateMesh();//temporary for testing single chunks
    }

    public void setNoise(NoiseGenerator newNoise)
    {
        noise = newNoise;
    }

    public void updateMesh()
    {
        List<Vector3> vertices = new List<Vector3>();
        List<Vector2> uvs = new List<Vector2>();
        List<int> trigs = new List<int>();
        float heightTally = 0f;
        maxHeight = float.MinValue;
        minHeight = float.MaxValue;
        //calculate vertices && UVs
        for (int vX = 0; vX < subdivisions + 1; vX++)
        {
            for (int vY = 0; vY < subdivisions + 1; vY++)
            {
```

```
            float posX = ((chunkCord.x * tileSize) - (tileSize / 2f)) +
                ((tileSize / (float)subdivisions) * vX);
            float posY = ((chunkCord.y * tileSize) - (tileSize / 2f)) +
                ((tileSize / (float)subdivisions) * vY);
            if (vY % 2 == 1)
            {
                posX += ((tileSize / (float)subdivisions) * 0.5f);
            }
            float posZ = noise.getHeight(posX, posY);
            vertices.Add(new Vector3(posX, posZ, posY));
            heightTally += posZ;
            if (posZ < minHeight) { minHeight = posZ; }
            if (posZ > maxHeight) { maxHeight = posZ; }
            float uvX = (float)vX / subdivisions + 1;
            float uvY = (float)vY / subdivisions + 1;
            if (vY % 2 == 1)
            {
                uvX += ((1f / (float)subdivisions) * 0.5f);
            }
            uvs.Add(new Vector2(uvX, uvY));
        }
    }
    avgHeight = heightTally / vertices.Count;
    //calculate triangles
    for (int vX = 0; vX < subdivisions; vX++)
    {
        for (int vY = 0; vY < subdivisions; vY++)
        {
            int index1 = (vX * (subdivisions + 1)) + vY;
            int index2 = (vX * (subdivisions + 1)) + vY+1;
            int index3 = ((vX + 1) * (subdivisions + 1)) + vY;
            int index4 = ((vX + 1) * (subdivisions + 1)) + vY+1;
            if (vY % 2 == 0)
            {
                //trig 1/2
                trigs.Add(index1);
                trigs.Add(index2);
                trigs.Add(index3);
                //trig 2/2
                trigs.Add(index2);
                trigs.Add(index4);
                trigs.Add(index3);
```

```
                }
                else
                {
                    //trig 1/2
                    trigs.Add(index1);
                    trigs.Add(index2);
                    trigs.Add(index3);
                    //trig 2/2
                    trigs.Add(index2);
                    trigs.Add(index4);
                    trigs.Add(index3);
                }

            }
        }
        //Update mesh properties
        mesh = gameObject.GetComponent<MeshFilter>().mesh;
        if (mesh == null)
        {
            mesh = new Mesh();
        }
        mesh.Clear();
        mesh.SetVertices(vertices.ToArray());
        mesh.SetTriangles(trigs.ToArray(), 0);
        mesh.SetUVs(0, uvs.ToArray());
        mesh.RecalculateNormals();
        mesh.RecalculateBounds();
        mesh.RecalculateTangents();
        mesh.UploadMeshData(false);
        MeshCollider col = GetComponent<MeshCollider>();
        if (col != null)
        {
            col.sharedMesh = mesh;
        }
    }
}
```

NoiseGenerator.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
```

```
[System.Serializable]
public class NoiseGenerator
{
    public int seedX = 9122023;  //random seed value for noise
    public int seedY = 3202219;  //random seed value for noise

    public Octive[] octives;

    public float getHeight(float xx, float yy)
    {
        float rval = 0f;
        foreach (Octive oct in octives)
        {
            rval += (oct.getHeight(xx, yy, seedX, seedY));
        }
        return rval;
    }

    [System.Serializable]
    public class Octive
    {
        public float magnitude = 1;
        public float scale = 10f;
        public float offset = 0f;
        public float power = 1f;
        public Octive subOctive = null;

        public float getHeight(float px, float py, float seedOffsetX,
            float seedOffsetY)
        {
            float rval = 0f;
            float xPos = seedOffsetX + ((float)px / (float)scale);
            float yPos = seedOffsetY + ((float)py / (float)scale);
            rval = Mathf.PerlinNoise(xPos, yPos) * (float)magnitude
                + (float)offset;
            float absVal = Mathf.Abs(rval);
            float sign = rval / absVal;
            rval = ((Mathf.Pow(absVal, power)) * sign);
            if (subOctive != null)
            {
```

```
                rval += subOctive.getHeight(px, py, seedOffsetX, seedOffsetY);
            }
            return rval;
        }
    }
}
```