Do Predator-Prey Relationships Affect Whether Sexual or Asexual Reproduction Is Advantageous?

Zachary Goodson

Table Of Contents

Purpose

The purpose of this experiment is to determine whether predators affect which reproduction type is advantageous.

Hypothesis

The scientist hypothesizes that there will be advantages for one reproductive strategy in the presence of predators. More specifically, hermaphroditic sexual reproduction will be the most advantageous as opposed to asexual reproduction, and opposite sex (male/female) sexual reproduction, because the hermaphroditic creatures will be able to create offspring with variety, and will be able to find mates easier then opposite sex (male/female) creatures will.

Review Of Literature

What Is Natural Selection?

The idea of natural selection was formed as part of Charles Darwin's theory of evolution in 1859. Natural selection is the idea that over time populations of living things adapt more and more to the environment around them. For example, an individual may possess a certain trait that makes it more likely for it to survive; that individual is then more likely to reproduce, passing the successful gene (and trait) onto the next generation. Natural selection is limited by certain rules (Khan Academy, n.d.). A species' traits and/or tendencies only change to reflect the current environment; genes tend to stay in a species if individuals with that gene survive long enough to mate (reproduce). However, the limited range of traits for a given species are determined by the available genes (making up the gene pool)--so new traits do not spring up from nowhere--unless there is a new mutation that can be tested against the selection pressures of the environment. When a species has a large population with a distribution of genes, natural selection essentially experiments with which traits or genes should stay in species to help keep it alive over time. Eventually, favored traits will win out. Natural selection cannot work without a reproducing population. For most species, the absolute minimum number is 2 of a species in a given place (e.g. one male, one female). However, species that can reproduce asexually only need 1. Reproduction is when the changes to a population can really take place. Natural selection can't change a species' current generation, but rather the following generation would be where the changes occur in the population (for example, genes leading to improved survival would be more common).

Sexual Reproduction

Sexual reproduction requires two parents as well as two major cellular components. These are an egg (typically female), and sperm (typically male). When they come in contact with each other the parents' alleles combine to start making new genotypes. This is why creatures that reproduce sexually tend to look similar but not exactly like their offspring. This form of reproduction has a few flaws. One big issue is if a population is very low, and a female can't find a male or vice versa, then the population would die, and eventually become extinct. Another issue is that it does not guarantee that beneficial genes will be passed down to the offspring. However there are also some clear advantages to reproducing this way. For instance, if a bad gene was in a parent then it has a chance of not being passed down to the offspring. A very important aspect of sexual reproduction is the ability for mutations to occur during the genetic reshuffling. This is generally a good thing because it makes more variety among the population, decreasing the chance for one single thing (selection pressure) to wipe a population out.

What Is A Hermaphrodite?

Hermaphrodites are a special type of sexually reproducing creatures. Instead of having different sexes lime most sexually reproducing organisms where one carries an egg, and the other carries sperm, hermaphrodites have both egg and sperm (Britannica, n.d.). In reference to the issues brought up in the Secual Reproduction section, when a species is hermaphroditic it can allow them to find mates quicker because they don't need a specific sex, but instead simply another creature in their species.

Asexual Reproduction

Asexual reproduction requires only one parent, and can be done 4 different ways. These are binary fission, budding, fragmentation, and parthenogenesis (Khan Academy, n.d.). Essentially asexual reproduction is cloning. One flaw that comes with reproducing this way is that any bad genes that the parent had are virtually guaranteed to be given to the offspring. However, if a parent has a good gene, the offspring will receive it too, which is good. This form of reproducing has no chance of variation except for occasional random mutation. This lack of variation in a population can be bad in the long term, due to there not being a large enough variety among a population's individuals, so a single complication (e.g. a new predator or disease) could wipe them all out at once.

Predator-Prey Relationships

Predation theory is a very old idea, but one of the first people to realize it was a mathematician by the name of Volterra. He observed that when the fishing business in the Adriatic sea was good there would be lots of fishermen, but overtime the amount of fishermen would decrease, possibly because of an over harvest. This pattern would repeat in a cycle (University Of Michigan, 2005). These observations support that as the population of a predator (fishermen) increases the population of the prey (fish) decreases. As well as when the population of the prey (fish) rises the population of the predator (fishermen) rises as well.

Carrying Capacity

Carrying capacity is the amount of creatures in a population that can be supported by the environment. According to Britannica, "The carrying capacity is different for each species in a habitat because of that species' particular food, shelter, and social requirements." (Britannica). This leads the population to drop when it is over the carrying capacity, and rise when under it.

Computer Simulations

"A computer simulation or a computer model is a computer program that attempts to simulate an abstract model of a particular system." (Science Daily, 2018). Computer simulations have been used to study and replicate many different things. This is because of how practical it is to make and test things with a simulation. Computers can process data much faster than humans can. Computer simulations are arguably the most reliable--or at least the easiest--way to test things. Unlike live experiments, in a simulation there are no chances that the creature being experimented on can run away from you and escape, or for it to be too cloudy or rainy to perform an experiment. Also, some experiments are very difficult to perform because they take too much time or money, or because it can be very difficult to isolate or control all of the experimental variables. Basically, it is always easier to simulate an experiment than to actually perform an experiment. However, simulations do have some disadvantages. Simulations, by definition, are not "real", and sometimes it can be difficult to choose realistic parameters. In other words, while it can be easy to run a simulation, it can be harder to make sure the simulation is realistic. Also, the more realistic one tries to make a simulation, the more complicated it can get, and the more time it can take to make and run the simulation. It can also be hard to have true randomness in a simulation, or to have a simulation cover enough space.

Simulations of Natural Selection

Replicating natural selection with simulations has a strong history. It has been attempted many times over the last few decades. For example, SelSim is a simulation program made in an attempt to simulate the outcome of combining alleles (Spencer CC, 2004). It is a software that was made specifically for predicting the new genotype of mixing alleles on very large scale scenarios. Its math is based on something called a Monte Carlo simulation, which is used for calculating risks (or probabilities) of different situations. In this case, risk refers to the possible outcomes of alleles mixing, and SelSim uses the Monte Carlo simulation to determine and/or calculate the odds of a specific outcome. This means SelSim can be used to calculate aspects of natural selection by predicting which traits will exist in a population longer than others in the context of a selection pressure. On top of that, modeling the effects of mutations can also be done with this program (Spencer CC, 2004).

In an older result, another simulated experiment was performed by a scientist at U. Michigan (Glesener, 1979). He wanted to try and simulate the relationship between predators and population size for both asexually-reproducing and sexually-reproducing creatures. He did this by making a model where asexual and sexual creatures coexisted. Both types of creatures had to compete with each other for food, and try to survive against a predator which ate both types of creatures. The data gathered from the simulations showed that the sexually reproducing species did better at surviving than the asexual species did. This is probably because once the (sexually reproducing) predator adapts to more efficiently hunt the asexual prey species, it could just do the same thing each time, with the idea that asexual species tend to reproduce carbon copies of themselves. On the other hand, the predators would have to “re-adapt” how to hunt sexually-reproducing creatures every cycle. When the sexually-reproducing predators can only

hunt the sexually-reproducing prey species, both predators and prey live on without either one going extinct, as they must constantly re-adapt. However when the same predators can only hunt the asexual prey species, the predators "learn" how to hunt the asexual species too efficiently, and eventually hunt them to extinction; this of course leads to the predators themselves becoming extinct (Glesener, 1979).

Why is Sexual Reproduction Still Present?

This previous simulation gave a realistic result of coevolution between predator and prey that favors sexual reproduction. However, is a predator-prey relationship necessary to favor sexual reproduction? Otherwise, asexual reproduction seems like it would be the best form of reproduction because asexual reproduction is easier--because one does not need to find a partner, and thus the population count could skyrocket from a single well-adapted individual. However this isn't what the majority of species on Earth do, instead nearly all continue to reproduce sexually. This seems strange, because a well-adapted individual can only pass on half of its genes (E Klarreich, 2010).

S Scheu and B Drossel designed a model that looked at the effect of resource availability. They used creatures that could reproduce sexually or asexually where "...sexual reproduction sets in when resources become scarce, and that at a given place only a few genotypes can be present at the same time" (S. Scheu, 2007). Their model allowed for mutation that could benefit or take away from a species. The simulation ends when only one form of reproduction is left. The results showed that sexual reproduction took over in every scenario except one, when survival conditions are at their worst and the death rate is too high for sexual reproduction to function (S. Scheu, 2007).

The Math of Simple Gene Variation

One of the simplest methods scientists use to predict what genes a child or offspring of sexual reproducing creatures could receive is called a Punnett Square. “One of the easiest ways to calculate the mathematical probability of inheriting a specific trait was invented by an early 20th century English geneticist named Reginald Punnett. His technique employs what we now call a Punnett square. This is a simple graphical way of discovering all of the potential combinations of genotypes that can occur in children, given the genotypes of their parents. It also shows the odds of each of the offspring genotypes occurring.” (Dennis O'Neil, 2012). When making a Punnett Square one starts by making a small 3 by 3 grid on paper. Next, one would put the genotype of one parent across the top and the other parent’s genotype down the left side, leaving a 2 by 2 space. It does not matter which parent's genotype you put on the top or side as it will give one the same results. After that all you have to do is fill in the remaining boxes by copying the row and column-head letters across or down into the empty squares. Once one finishes mapping out the genotypes they have all the possibilities of the gene that the offspring can get (assuming no mutations).

In order to understand the data one should know the different identifiers, specifically homozygous, and heterozygous. Homozygous refers to a gene that has identical alleles on both homologous chromosomes. It is referred to by two capital letters (AA) which is homozygous dominant, and two lowercase letters (aa) for homozygous recessive (assuming that a given gene can be dominant or recessive). Heterozygous means having one of each two different alleles (Aa). As will be mentioned below, in the simulations created for this work, dominant/recessive genetics were not used. Instead, the offspring for asexually reproducing creatures were

essentially clones, and the offspring for sexually reproducing creatures (treated as hermaphrodites--creatures with both male and female characteristics) had “weighted averages” calculated from the parent “gene values” to determine “phenotypes” for their inheritable characteristics.

Mutation

Whenever the process of reproduction is happening there is a chance of mutation occurring. Mutation is an error in the DNA often caused by the miscopy of an allele, or over exposure to things like ultraviolet light or other types of ionizing radiation. The dictionary definition is, “ a relatively permanent change in hereditary material that involves either a change in chromosome structure or number (as in translocation, deletion, duplication, or polyploidy) or a change in the nucleotide sequence of a gene's codons (as in frameshift or missense errors) and that occurs either in germ cells or in somatic cells but with only those in germ cells being capable of perpetuation by sexual reproduction” (Merriam Webster, n.d.). Mutation adds random uncertainty to natural selection. Most genetic mutations actually don’t do anything. Others can be beneficial; for example, a mutation can make a creature immune to disease, or more energy efficient. On the other hand, mutations can make creatures worse off, for example instead of making something immune to a disease it might make it more vulnerable, or instead of making something more energy efficient it could make it less energy efficient--or more vulnerable to a predator. In the simulations described below, there was a 10% chance of mutation with every reproduction.

What is Unity?

Unity is a popular game engine with powerful simulated physics (that allow simulations of actual environments and forces), and graphics systems built in. It was developed and published by Unity Technologies. Utilizing Unity will allow me to make collision detection (e.g. between two creatures, or between a creature and food) and graphics rendering easier, as well as handling variables and cross-referencing between scripts that have been written in C# and compiled in Visual Studios (Unity, 2019).

Materials and Procedures

Materials

1. Unity (version: 2021.1.3f1)
2. Visual Studios (version: Community 2019) using C#
3. Computer Specifications: GTX 1650s graphics card, 16 Gb of 3200 mHz RAM, Ryzen 5 2600 processor, 1 Tb of storage, 500 watt power supply.

Procedures

To find an in-depth explanation of all code for the project see:

https://github.com/Quantam-Studios/Natural-Selection-Simulation--Science-Fair-21-22

To see all code turn to Appendix I.

Creating The Project

1. Create a new Unity project with the 2D template.
2. In “Player” settings, set “run-in-background” to true, graphics settings to low (to prevent needless performance loss as this is only rendering circles), and choose the OS you want to build to (this experiment was done on Windows 10).

Creating Creatures (both predator, and prey)

1. Create a new circular sprite game object, and add the following: Rigidbody2D, SpriteRenderer, CircleCollider2D.
2. On the Rigidbody2D set the gravity, angular momentum to 0, and lock rotation of the z axis.
3. Add a new script to this game object (this will control the behavior)
4. Make this game object a prefab

5. Repeat steps 1-5 six more times

Once the simulation app has been made follow these steps:

1. Begin setting up simulation by pressing the “Setup A Simulation” button.
2. Select the desired settings for the test.
    a. Choose the prey reproduction type
    b. Choose the initial amount of prey (10 for this experiment)
    c. Optional: Choose the predator reproduction type
    d. Optional: Choose the initial amount of predators (4 for this experiment)
    e. Choose the initial amount of food (300 for this experiment)
    f. Choose the spawn rate of food (0.33 seconds for this experiment)
    g. Choose the data collection rate (1.0 seconds for this experiment)
    h. Choose the time limit (5:00 minutes for this experiment)
    i. Choose a folder name
3. Click the “Run Simulation” button
4. Repeat until all variations have been tested with 3 trials each while ensuring that initial food, initial prey, initial predators, food spawn rate, time limit, and data collection rate are constant from simulation to simulation.

An Explanation of the Simulation Mechanics

Frame Rate:

The frame rate of the simulation is held at a constant 60 frames per second (fps). This means that all calculations are done 60 times a second.

Initial Trait Values:

Initial Trait Values In Predators and Prey			
Trait	Speed	Size	Sense Radius
Predators	5	1	4
Prey	2	1	3

Figure 1. This table shows the initial values of traits in the initial creatures that are spawned at the beginning of a simulation.

Energy Mechanics:

Note: *Random.Range()* means that a random value is chosen between the two provided values.

Initial Energy Values:

Energy Constants In Predators and Prey					
Value	Min. to Rep.	Min. to Starve	Food	Rep. Cost	Initial Energy
Predators	20,000	8,000	10,000	10,000	5,000
Prey	10,000	5,000	2,500	5,000	2,500

Figure 2. This table shows the constant energy values at play in creatures throughout a simulation.

Resting Cost:

All creatures (both predators and prey) use the following formula to calculate energy loss when resting:

$$energyLoss = Time.deltaTime$$

This means that when a creature rests it loses 60 energy every second it rests (because it updates at 60 fps).

Movement Cost:

All creatures (both predators and prey) use the following formula to calculate energy loss when moving:

$$energyLoss = ½(size)(speed/time.deltaTime)^2$$

This means that when a creature moves it loses its speed divided by the size multiplied by the time they have been moving. Example: Let's say a creature had 5,000 energy before moving, a speed of 2, and a size of 1, and then moved 1 second (keep in mind this means this calculation occurred 60 times). Plugging these values into the equation we have:

*(energyLoss = ½(1)(2/1)^2) * 60 = 120*

So the creature would have lost 120 energy from its original 5,000 energy leaving it 4880.

Reproduction Cost:

Energy cost for reproduction is a constant in both predators and prey. Predators have to use 10,000 energy to reproduce. Prey have to use 5,000 energy to reproduce.

Traits:

1. Speed: Determines how fast a creature moves.
2. Size: Determines how much space the creature takes up in the world
3. Sense Radius: Determines how large of a range a creature can sense food, mates, and predators.

Reproduction:

Minimum and Maximum Values Of Traits In Predators and Prey						
Trait	Min. Speed	Max. Speed	Min. Size	Max. Size	Min. Sense Radius	Max. Sense Radius
Predators	1	10	0.2	2	0.1	10
Prey	1	8	0.1	1.5	0.1	10

Figure 3. This table shows the minimum, and maximum values of traits in both predators, and prey.

Gendered Sexual Creatures:

Only sexual creatures with genders have to determine the sex of the offspring. This is done with the following equation:

randSex = Random.Range(1, 2)

If the value of randSex is 1 then a female will be produced, and if the value is a 2 then a male will be produced. This means there is a 50% chance of having a male as well as a female.

Sexual Creatures:

Initial Trait Generation:

Both predator, and prey will first generate a trait value with the formula:

offspringTraitValue = (parent1TraitValue + parent2TraitValue)/2

Trait Variation:

Both predator, and prey will secondly generate and then apply random variation with the formula:

*traitVariation = Random.Range((offspringTraitValue * -0.2), (offspringTraitValue * 0.2))*

This variation value is then applied to the offspring trait value with the following formula.

offspringTraitValue + traitVariation

Asexual Creatures:

Initial Trait Generation:

Asexual prey, and predators both initally generate a value for the offspring which is identical to the parent:

offspringTraitValue = parentTraitValue

All Creatures:

Mutation:

All reproductive types of predators and prey undergo the same mutation method seen in the following equations. First a random value is generated with the following equation:

Random.Range(0, 10)

If the generated value is 1 then the trait will be mutated with the following equation:

offspringTraitValue = Random.Range(minTraitValue, MaxTraitValue)

This results in a completely random trait within the set restrictions of the trait values (see Figure 3 to see minimum and maximum values for each trait). This means there is a 10% chance of mutation on each trait.

General Prey:

All reproductive types have states 1-3, but asexual prey do not have state 4 as they do not need a mate to reproduce. Prey have priorities and they are in the following order: if unsafe then "Flee", if no food or mates are sensed then "Wander", if food is sensed and mating is not an option then "GetFood", if mating is an option and a mate is sensed then "GetMate".

When Can Prey Mate?

Prey can mate when the following are true: The creature has the minimum energy to reproduce and not starve, a viable mate has been sensed.

States:

1. Flee: In this state the creature will run in the opposite direction of nearby predators
2. Wander: In this state the creature will generate a random position and move to it.
3. GetFood: In this state the creature will move directly to a piece of food
4. GetMate: In this state the creature will move directly to a mate and when collision occurs offspring will be produced.

General Predator:

All reproductive types have states 1-2, but asexual predators do not have state 3 as they do not need a meed to reproduce. Predators have priorities, and they are in the following order: if

no food or mates are sensed then “Wander”, if food is sensed and mating is not an option then “GetFood”, if mating is an option and a mate is sensed then “GetMate”.

States:

1. Wander: In this state the creature will generate a random position and move to it.
2. GetFood: In this state the creature will chase its food (prey).
3. GetMate: In this state the creature will move directly towards a mate, and when collision occurs offspring will be produced.

Results

Figure 4. This figure shows the screen of which the user can set initial values of prey, predators, food, choose the rate of data collection and food spawning, select the prey and predator reproductive types, and choose a folder name for the log files to be stored in.

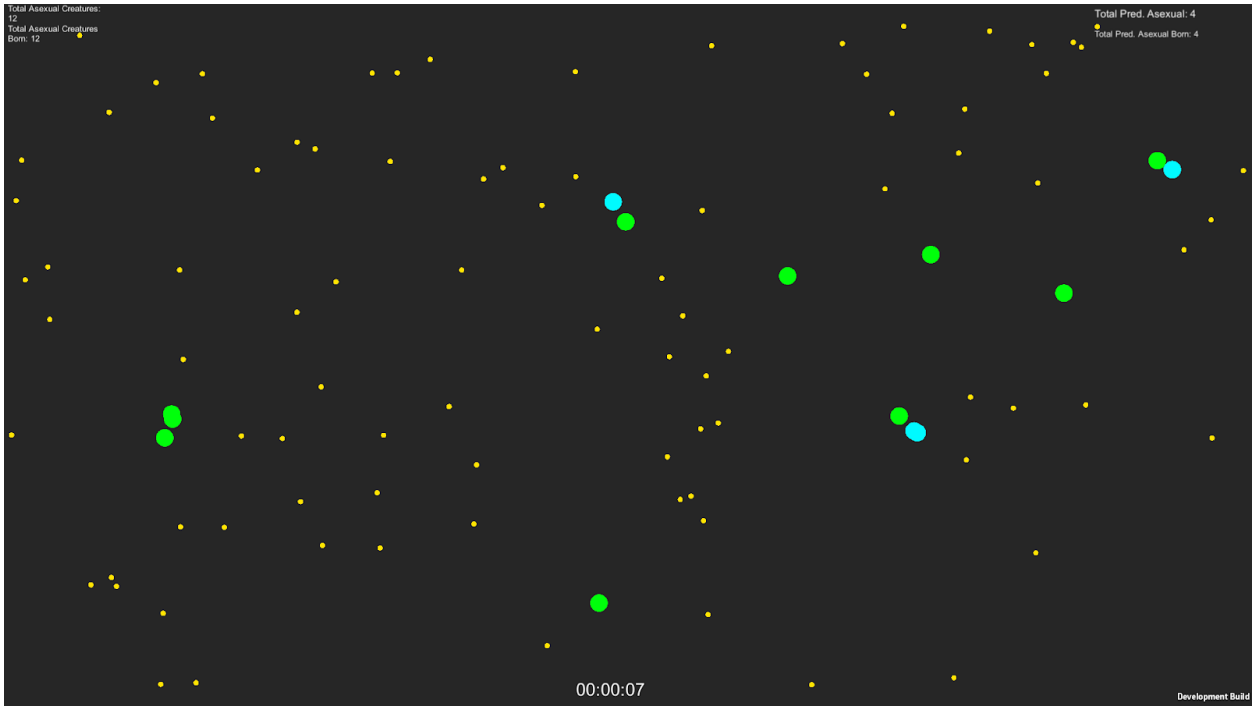


Figure 5. This figure shows the beginning of a simulation with asexual prey (green circles), and asexual predators (teal circles)

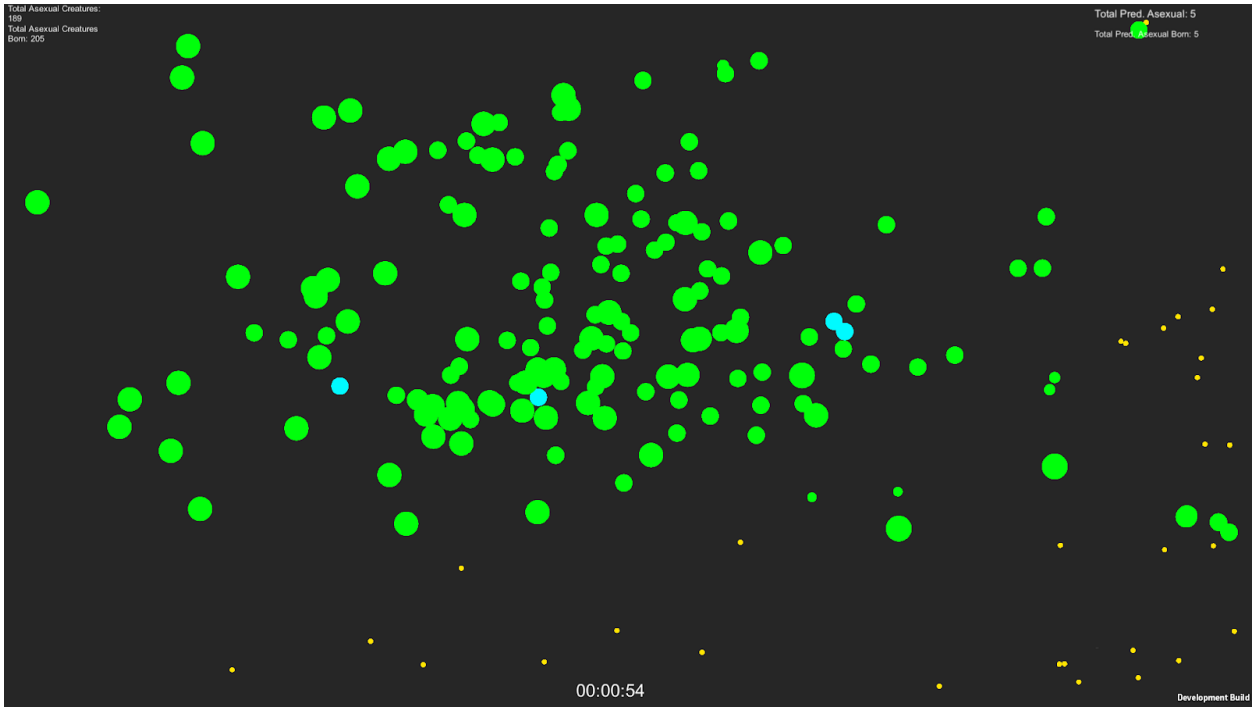


Figure 6. This figure shows the asexual prey (green circles) population in an "exponential" growth phase as the asexual predators (teal circles) begin their own growth phase.

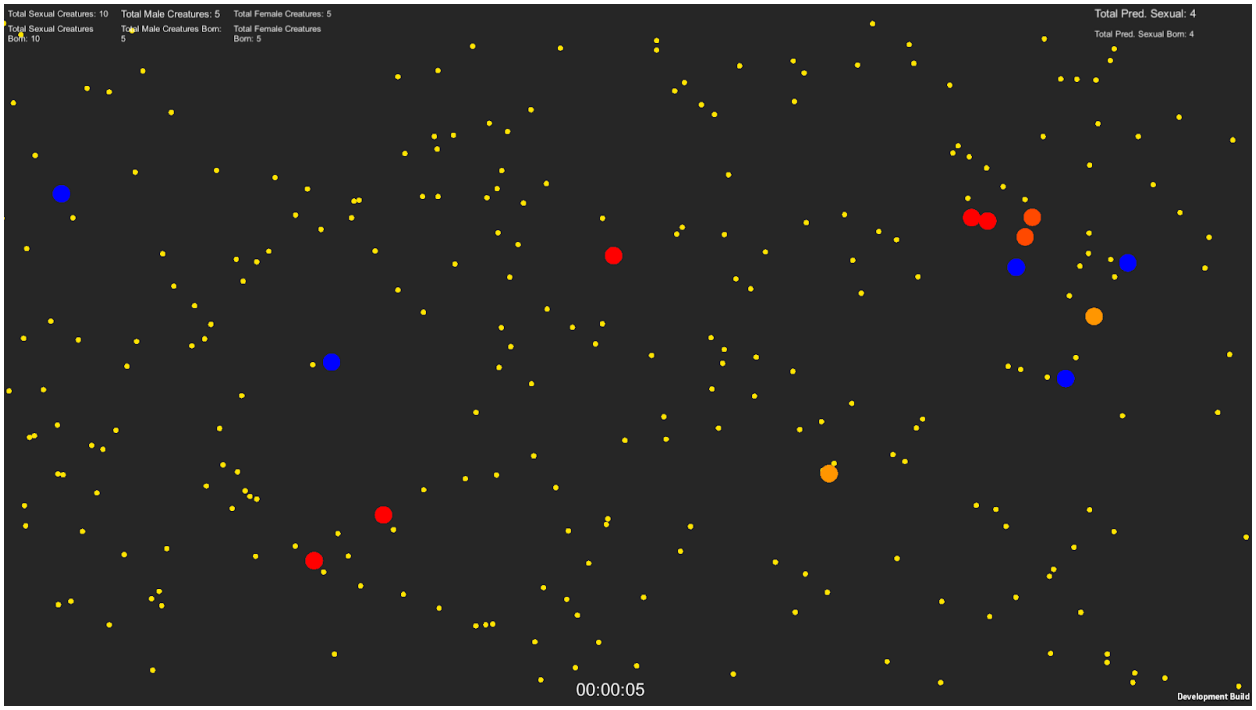


Figure 7. This figure shows the beginning of a simulation with sexual creatures and predators where prey females are blue circles, prey males are red circles, predator females are dark orange circles, and predator males are bright orange circles.

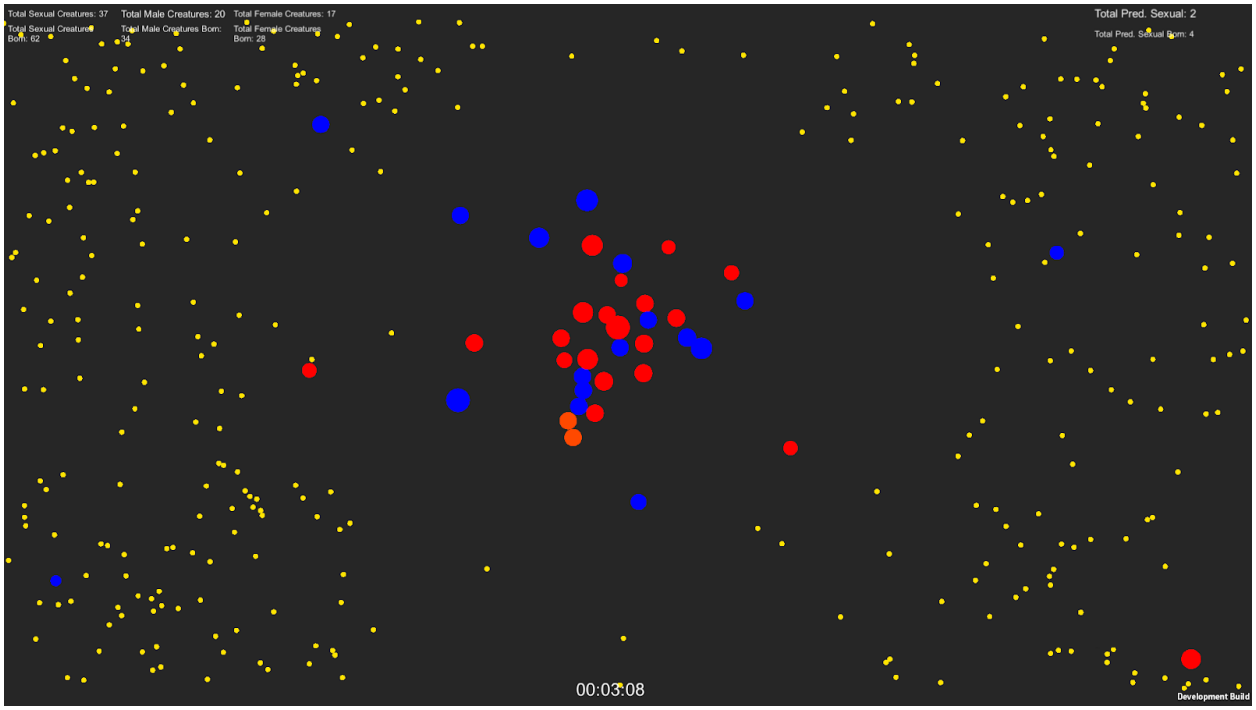
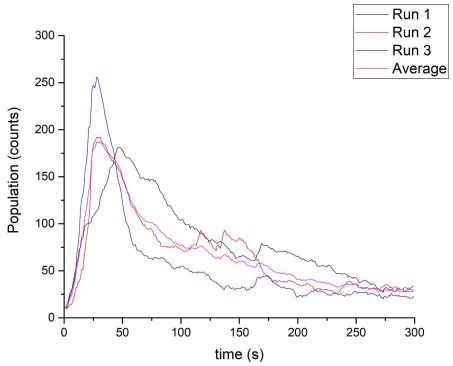


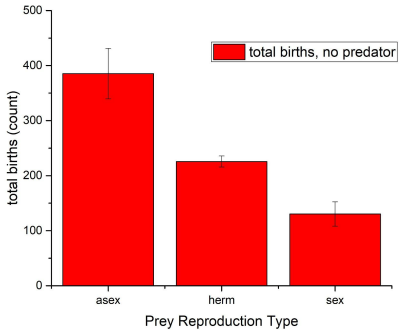
Figure 8. This figure shows the middle of a simulation with sexual prey, and sexual predators. The sexual prey (blue and red circles) have begun to take off in terms in of population size. However, the sexual prey (orange shaded circles) currcently only have females alive which means the predator population is not going to get any bigger.

Figure 9. No Predators



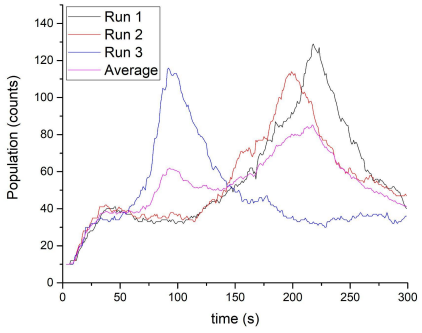
A.

This figure shows the asexual population over time in 3 trials, and the average.



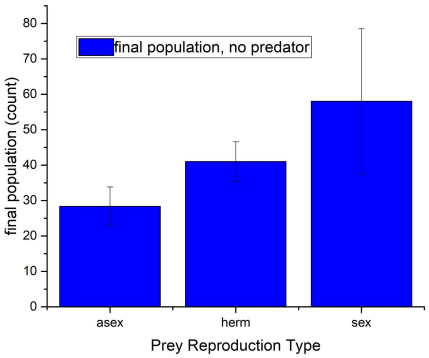
D.

This figure shows the total creatures birthed for a given reproductive type.



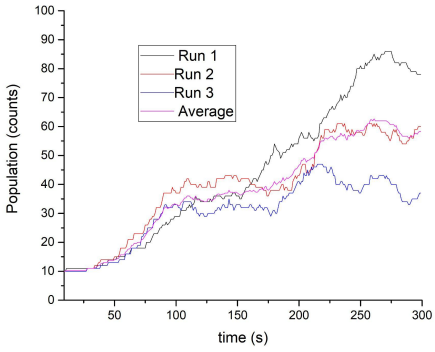
B.

This figure shows the hermaphrodite population over at time in 3 trials, and the average.



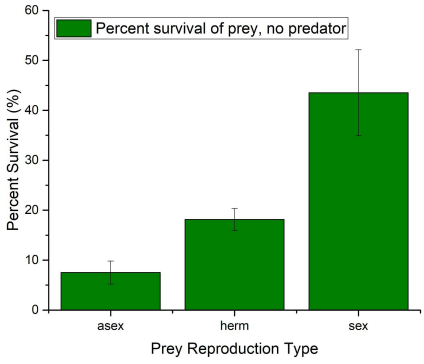
E.

This figure shows the number of creatures the end of a simulation.



C.

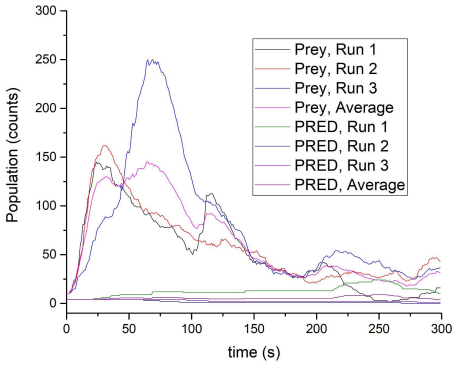
This figure shows the sexual population over each time in 3 counts over trials, and the average.



F.

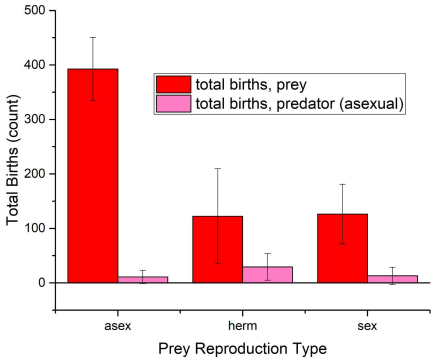
This figure shows the survival rate of reproductive type.

Figure 10. Asexual Predators



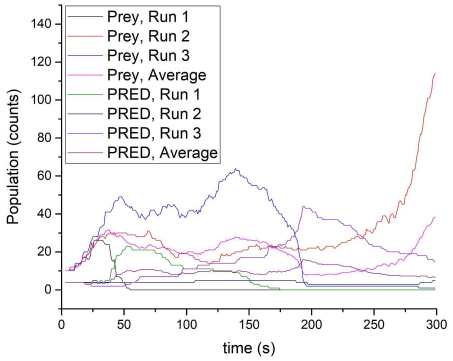
A.

This figure shows the asexual population over time in 3 trials, and the average.



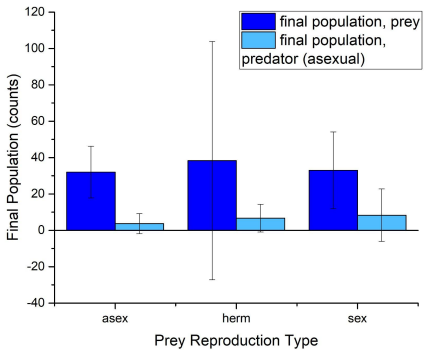
D.

This figure shows the total creatures birthed for a given reproductive type.



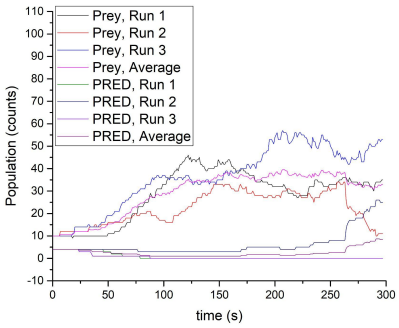
B.

This figure shows the hermaphrodite population over at time in 3 trials, and the average.



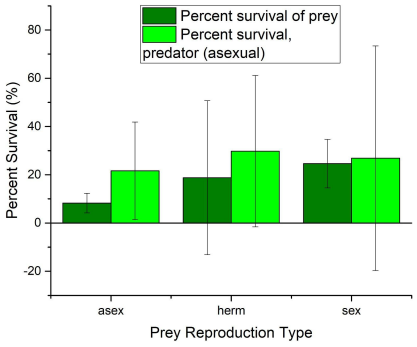
E.

This figure shows the number of creatures the end of a simulation.



C.

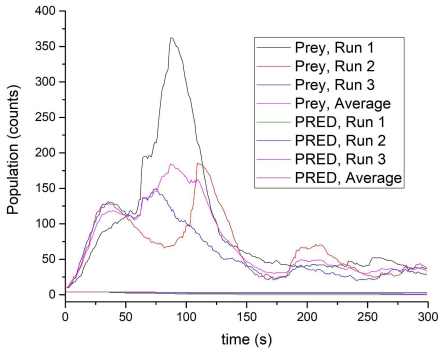
This figure shows the sexual population over each time in 3 counts over trials, and the average.



F.

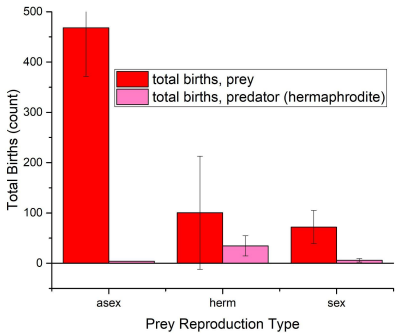
This figure shows the survival rate of reproductive type.

Figure 11. Hermaphrodite Predators



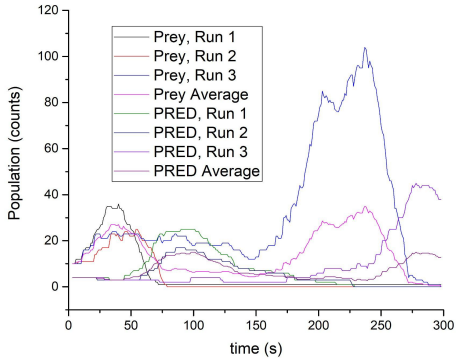
A.

This figure shows the asexual population over time in 3 trials, and the average.



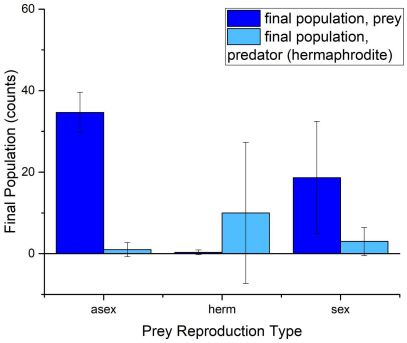
D.

This figure shows the total creatures birthed for a given reproductive type.



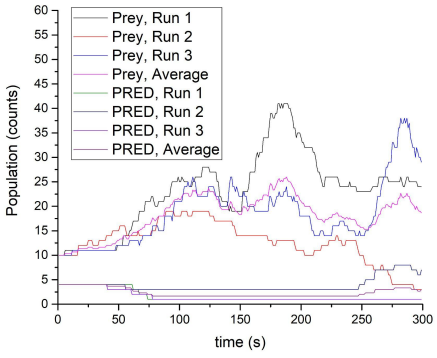
B.

This figure shows the hermaphrodite population over at time in 3 trials, and the average.



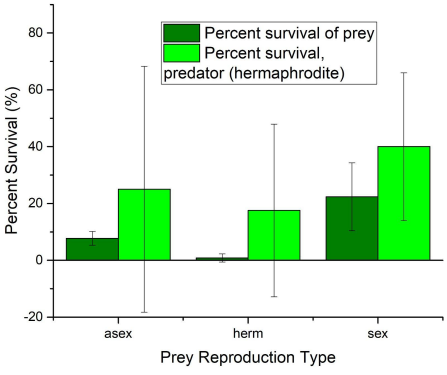
E.

This figure shows the number of creatures the end of a simulation.



C.

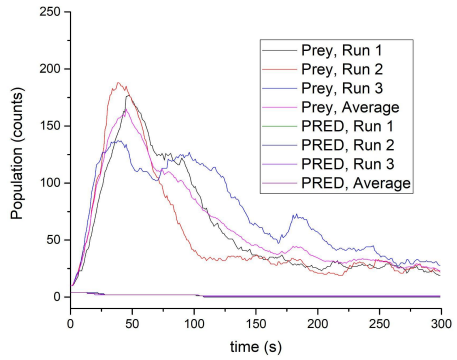
This figure shows the sexual population over each time in 3 counts over trials, and the average.



F.

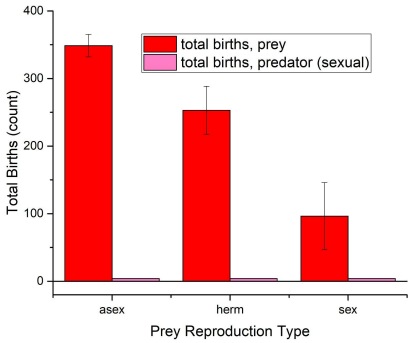
This figure shows the survival rate of reproductive type.

Figure 12. Sexual Predators



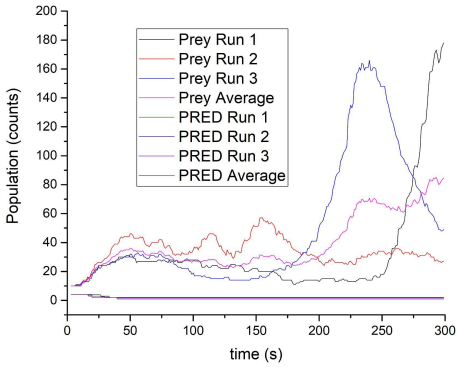
A.

This figure shows the asexual population over time in 3 trials, and the average.



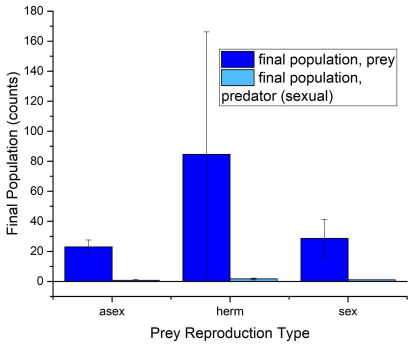
D.

This figure shows the total creatures birthed for a given reproductive type.



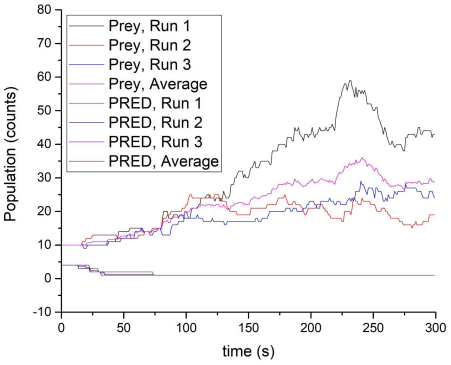
B.

This figure shows the hermaphrodite population over at time in 3 trials, and the average.



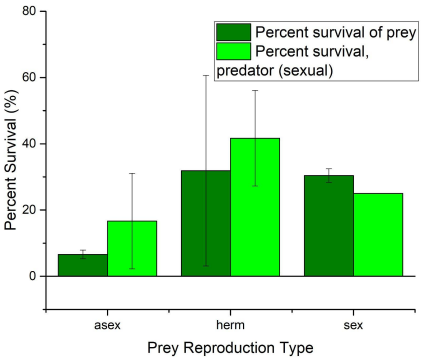
E.

This figure shows the number of creatures the end of a simulation.



C.

This figure shows the sexual population over each time in 3 counts over trials, and the average.



F.

This figure shows the survival rate of reproductive type.

Discussion

The simulations without predators, results shown in Figure 9, have a lot of important findings. In asexual prey it is clear that the trends of the population over time are the same. In general asexual prey populations tended to spike early on as there was a large amount of food, but soon after the population size would plummet because the food source wasn't replenished at the rate needed to support a lot of creatures. The results in Figure 9A also show that the carrying capacity seems to be roughly 30-45 creatures when it comes to asexual prey. Hermaphrodites on the other hand seem to be extremely varying. According to Figure 9B run 1 and run 3 spiked in population at opposite ends of the allotted time. Run 1 spiked in the 2 minutes before dropping off, but run 3 spiked early on before falling off. Overall though the results are similar to asexual prey as both seem to support that the carrying capacity is roughly 30-45 and that populations tend to follow a "boom and bust" pattern. Sexual prey population seems to be fairly consistent. Sexual reproduction breaks the "boom and bust" trend seen in hermaphrodites and asexual prey. This is seen in Figure 9C with the population steadily rising. Figure 9D shows that on average asexual prey managhed to reproduce the most which is probably because they don't have to find a mate, and instead can instantly reproduce when enough energy has been amassed. Hermaphrodites managed to reproduce the second highest amount, and this is probably because they do not have to find opposite sex mates, and instead have a larger mating pool. This leaves sexual prey in the third spot. This seems fitting because seuxal prey have to find mates that are the opposite sex in order to reproduce which limits their options. In contrast Figure 9E shows that sexual reproduction paid off and resulted in the highest amount of offspring in the end. With Hermaphrodite in second, and asexual in third. This is most likely because sexual reproduction had less inter-species competition for food. When the populations of asexual creatures would

spike all of the food would be eaten resulting in more offspring. However, that is not sustainable as shown in Figure 9A and Figure 9E, this leads the population to eat all of the food supply causing lots of death due to starvation. Figure 9F shows that on average sexually reproducing creatures had a higher chance of surviving when compared to asexual. Sexual prey saw a 43.5% survivability rate, with hermaphrodite at 18.2%, and lastly asexual prey with only a 7.5% rate. This could be because of higher variability among offspring allowing for more “fit” offspring to be produced.

As seen in many figures ranging from 10-12, the populations of predators vary so much it is hard to form conclusive thoughts. Asexual prey saw almost difference in population when a predator was intordued as shown in figures 10A, 11A, and 12A. The figures still show the same “boom and bust” seen when no predators were present. When looking at results with predators present the carrying capacity is shown to broaden its range to around 20-60 creatures. Hermaphrodite prey have results that vary a lot. In fIgures 10B, 11B, 12B it can be seen that sometimes a population would never manage to take off and die while some would have a “boom and bust” pattern. Hermaphrodite predators whose results are shown in figures 11A-F show that they would often eat all prey extremely fast resulting in them starving to death themselves. Opposite sex predators seem to also have varying results. On average though the end population of prey trended upwards when compared to those without predators. This might be because of the selection pressures that the prey had to face. However, it is not statistically significant as the error bars are massive. Asexual predators proved to be a dominant force when it came to eating prey. Shown in Figures 10A-F roughly 2 of the 3 runs resulted in 0 prey. However in contrast 1 of the 3 runs resulted in a survival rate over 55%. Opposite sex prey seemed to have some trouble when it came to predators being introduced with only a 20-30% rate of survival depending on the predator present, but they still proved to have the highest survivability rate in all predators. Overall predators being present proved to impact the survivability rates of prey as shown in figures 10F, 11F, and 12F. In general, opposite sex prey

proved to have the highest survivability rates in all circumstances. THis is likely because unlike other reproductive type the sexual prey seemed to avoid the “boom and bust” pattern. This is probably caused by the fact that it is harder for them to reproduce, meaning that the food supply doesn’t drop rapidly.

Conclusion

The scientist successfully created a software platform for executing simulations of population growth and decay in the context of both single-organism (plus food source) and two-organism (predator/prey) populations; the platform can run stably with several hundred simultaneous independent simulated organisms (roughly 800), and keep track of a number of variables and report them to log files. Logged values include instantaneous population counts and total births, plus distributions of inheritable phenotypes (speed, sense radius, and size). The code has been completely rewritten becoming more efficient, and feature filled. A proper implementation of a sense radius has been added in, allowing for creatures to truly sense things such as mates, food, and predators in their environment. Sexually reproducing creaturs now have different sexes. Sexually reproducing creatures can sense when a potential mate (a creature of the opposite sex) is also ready to mate allowing them to save energy. Phenotypes no longer have hard set values, and instead feature smooth variation. Size is a real phenotype that impacts energy conservation when moving. All values are now logged in text files that can be accessed at any time once taken, and it is done automatically. This is a major update, and prevents the need for manually recording data. A GUI has been implemented allowing for the user to easily change values to their liking. Predators have become equal with prey, and now feature genetic inheritance, and sensory. The entire code base is very modular in the sense that all values can be changed almost instantly.

The hypothesis was partially accepted, and partially rejected. Sexual reproduction showed higher survivability rates in comparison to asexual reproduction. However sexual

reproduction that had males and females proved to be the best in terms of survivability, with and without predators compared to hermaphroditic sexual reproduction. Male/female sexual reproduction also had more consistent results.

There are a multitude of things that could be improved, as well as expanded upon in this project. One improvement that could be made would be an increase in initial creature amounts, particularly predators. However, doing this in a way that would result in stable relationships would likely require a higher end computer. Another thing that could be further investigated is phenotypic distributions. Looking into this would allow for a greater understanding of how the predator-prey relationships affect each other as well as how they co-evolve. More trials could also have improved the quality, and reliability of the data. This could also be improved with longer time limits.

Bibliography

Encyclopædia Britannica, inc. (n.d.). *Carrying capacity*. Encyclopædia Britannica. Retrieved March 16, 2022, from https://www.britannica.com/science/carrying-capacity

Encyclopædia Britannica, inc. (n.d.). *Hermaphroditism*. Encyclopædia Britannica. Retrieved March 11, 2022, https://www.britannica.com/science/hermaphroditism

Trophic links: Predation and parasitism. Predator-Prey Relationships. (n.d.). Retrieved March 11, 2022, https://globalchange.umich.edu/globalchange1/current/lectures/predation/predation.html

"mathworks.com" *What Is the Genetic Algorithm? - MATLAB & Simulink*, https://www.mathworks.com/help/gads/what-is-the-genetic-algorithm.html.

UCSF. "Basic Genetics." Https://Kintalk.org/, UCSF, kintalk.org/genetics-101/.

KenshinKenshin. "Can Sexual Reproduction Create New Genetic Information?" *Biology Stack Exchange*, 1 Sept. 1963, https://biology.stackexchange.com/questions/8518/can-sexual-reproduction-create-new-genetic-information.

&Lt Bio 52 - Sexual Reproduction : Reducing Mutations &Gt, https://www.cs.hmc.edu/~rmcknigh/projects/bio52-project3/bio52-mutation.html.

Reference, G. (2019). *What kinds of gene mutations are possible?*. [online] Genetics Home Reference. Available at: https://ghr.nlm.nih.gov/primer/mutationsanddisorders/possiblemutations [Accessed 3 Mar. 2022].

Foundation, C. (2019). *Reproduction*. [online] CK-12 Foundation. Available at: https://www.ck12.org/biology/reproduction/lesson/Asexual-vs.-Sexual-Reproduction-BIO/ [Accessed 3 Mar. 2022].

\Www2.palomar.edu. (2019). *Basic Principles of Genetics: Probability of Inheritance*. [online] Available at: https://www2.palomar.edu/anthro/mendel/mendel_2.htm [Accessed 3 Mar. 2022].

Biography. (2019). *Charles Darwin*. [online] Available at: https://www.biography.com/scientist/charles-darwin [Accessed 3 Mar. 2022].

Khan Academy. (2019). *Darwin, evolution, & natural selection*. [online] Available at: https://www.khanacademy.org/science/biology/her/evolution-and-natural-selection/a/darwin-evolution-natural-selection [Accessed 3 Mar. 2022].

Ndsu.edu. (2019). *Population and Evolutionary Genetics*. [online] Available at: https://www.ndsu.edu/pubweb/~mcclean/plsc431/popgen/popgen5.htm [Accessed 3 Mar. 2022].

Darwins-theory-of-evolution.com. (2019). *Darwin's Theory Of Evolution*. [online] Available at: https://www.darwins-theory-of-evolution.com/ [Accessed 3 Mar. 2022].

Encyclopedia Britannica. (2019). *natural selection | Definition & Processes*. [online] Available at: https://www.britannica.com/science/natural-selection [Accessed 3 Mar. 2022].

Merriam-webster.com. (2019). *Definition of ASEXUAL*. [online] Available at: https://www.merriam-webster.com/dictionary/asexual [Accessed 3 Mar. 2022].

Scienceprimer.com. (2019). *Alleles, Genotype and Phenotype | Science Primer*. [online] Available at: https://scienceprimer.com/Alleles-genotype-phenotype [Accessed 3 Mar. 2022].

ScienceDaily. (2019). *Computer simulation*. [online] Available at: https://www.sciencedaily.com/terms/computer_simulation.htm [Accessed 3 Mar. 2022].

Baer, Charles F. "Does mutation rate depend on itself." *PLoS biology* vol. 6,2 (2008): e52. doi:10.1371/journal.pbio.0060052

Bitstorm.org. (2019). *John Conway's Game of Life*. [online] Available at: https://bitstorm.org/gameoflife/ [Accessed 3 Mar. 2022].

Scheu, S, and B Drossel. "Sexual reproduction prevails in a world of structured resources in short supply." *Proceedings. Biological sciences* vol. 274,1614 (2007): 1225-31. doi:10.1098/rspb.2007.0040

Spencer, C. and Coop, G. (2004). SelSim: a program to simulate population genetic data with natural selection and recombination. *Bioinformatics*, 20(18), pp.3673-3675.

GLESENER, R. (1979). Recombination in a Simulated Predator-Prey Interaction. *American Zoologist*, 19(3), pp.763-771.

Klarriech, E. (2019). Why Sex?. *Science Foundation*, [online] pp.105, 107. Available at: https://www.simonsfoundation.org/2010/05/18/why-sex/#sidebar-anchor [Accessed 3 Mar. 2022].

Unity. (2019). *Game engines - how do they work? - Unity*. [online] Available at: https://unity3d.com/what-is-a-game-engine [Accessed 3 Mar. 2022].

Appendix I.

Total lines of C# code: 4455

Total source lines of C# code (sloc): 4124

Prey Spawning: (46 total lines, 42 sloc)

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

public class PredatorSpawning : MonoBehaviour
{
    // PREDATORS
    [Header("Predators")]
    public GameObject[] predators;
    // predator holders
    public Transform[] predatorHolders;

    // BOUNDS
    [Header("Boundaries")]
    // min and max (x,y) values for food spawn locations
    public float maxX;
    public float maxY;
    public float minX;
    public float minY;

    // SPAWNER
    // this function is called from the SpawnInitialPredators() function in MenuManager which is called when the button "Run Simulation Button" is pressed
    public void spawnInitialPredators(int predatorCount, int activePredator)
    {
        for (int i = 0; i < predatorCount; i++)
        {
            // generate random position
            float x = Random.Range(minX, maxX);
            float y = Random.Range(minY, maxY);
            Vector3 spawnPos = new Vector3(x, y, 1);

            // spawn the predator
            // if the predator is NOT sexual
            if (activePredator != 0)
                Instantiate(predators[activePredator], spawnPos, Quaternion.identity, predatorHolders[activePredator]);
            else // else if the predator is sexual
            {
                // Ensures a close to even if not perfectly even distribution of females, and males.
                if(i % 2 != 0) // spawn a female
                    Instantiate(predators[0], spawnPos, Quaternion.identity, predatorHolders[activePredator]);
                else // spawn a male
                    Instantiate(predators[3], spawnPos, Quaternion.identity, predatorHolders[activePredator]);
            }
        }
    }
}
```

Asexual Prey: (344 total lines, 316 sloc)

```
using UnityEngine;

public class AsexualCreature : MonoBehaviour
{
    // TRAITS
    [Header("Trait Values")]
    public float speed;
    public float size;
    public float senseRadius;
    // tracking of traits
    private CreatureStatistics creatureStatistics;
    // trait divisions
    public int speedDiv;
    public int sizeDiv;
```

```
public int senseRadiusDiv;

// ENERGY SETTINGS
[Header("Energy Settings")]
public float minEnergyForRep;
public float minEnergyToBeHungry;
public float energyInFood;
public float energyToRep;

 // RUN TIME EFFECTED VARIABLES
[Header("Simulated Variables")]
public float energy;
public bool readyForRep;
// MOVEMENT
public bool move;
private Vector2 targetPos;
private float timeBtwDecision;
public float setTimeBtwDecision;
// PERIODIC BOUNDS
public float setHitBoundsTime;
private float hitBoundsTime;
public bool swapSides;
public int initialLayer;

// MOVEMENT SETTINGS
[Header("Movement Values")]
public float maxX;
public float maxY;
public float minX;
public float minY;

// STATE
[Header("State Variables")]
public string currentState;

// SENSORY
[Header("Sensory Variables")]
// check for predators
public bool notSafe;
public LayerMask predators;
private Vector3 predatorPos;
// check for food
public bool foodClose;
public LayerMask food;
private Vector2 foodObjectPos;

// REPRODUCTION
[Header("Reproduction")]
public GameObject asexualCreature;
private Transform parentObjectOfOffspring;
// Mutations
[Header("Mutations")]
public int mutationRate;
public float minSize;
public float maxSize;
public float minSpeed;
public float maxSpeed;
public float minSenseRadius;
public float maxSenseRadius;

// Start is called before the first frame update
void Start()
{
    //Initialize values
    // Set size based on the size gene
    transform.localScale = new Vector3(size, size, size);
    readyForRep = false;
    // allow movement
    move = true;
    // reset timer of decisions
    timeBtwDecision = 0;
    // reset swap side bool
    swapSides = false;
    // Set initialLayer
    initialLayer = gameObject.layer;
    // Get statistics reference
    creatureStatistics = FindObjectOfType<CreatureStatistics>();

    // Update statistics
    // POPULATIONS ones
```

```
    CreatureStatistics.allTimeAsexualCreatureCount += 1;
    CreatureStatistics.asexualCreatureCount += 1;
    // TRAIT ones
    // Get Divisions
    // size
    sizeDiv = creatureStatistics.getSizeDivision(size);
    // speed
    speedDiv = creatureStatistics.getSpeedDivision(speed);
    // sense radius
    senseRadiusDiv = creatureStatistics.getSenseRadiusDivision(senseRadius);
    // Update Trait Stats
    // size
    CreatureStatistics.sizeDivisionTracker[sizeDiv] += 1;
    // speed
    CreatureStatistics.speedDivisionTracker[speedDiv] += 1;
    // sense radius
    CreatureStatistics.senseRadiusDivisionTracker[senseRadiusDiv] += 1;

    // Set parentObjectOfOffspring to the object holding all ASEXUAL creatures
    parentObjectOfOffspring = GameObject.FindGameObjectWithTag("AsexualCreatureHolder").transform;
}

// Update is called once per frame
void Update()
{
    // WANDER STATE
    // this state has the creature wander around the world.
    // In this state the creature can: Move, Rest.
    if (currentState == "Wander")
    {
        // Move or Not
        timeBtwDecision -= Time.deltaTime;
        if (timeBtwDecision <= 0)
        {
            // Reset Timer
            timeBtwDecision = setTimeBtwDecision;
            // Randomly Decide What To Do
            int randint = Random.Range(0, 2);
            if (randint == 0)
            {
                generateRandTargetPos();
                move = true;
            }
            else
                move = false;
        }
    }

    // FLEE STATE
    // this state has the creature move away from predators.
    // In this state the creature can: Move.
    if (currentState == "Flee")
    {
        // update targetPos to be the opposite of the predators direction
        targetPos = predatorPos;
        // move
        move = true;
    }

    // GETFOOD STATE
    // this state has the creature move to the food piece it sensed.
    // In this state the creature can: Move.
    if (currentState == "GetFood")
    {
        // Set targetPos to the food object.
        targetPos = foodObjectPos;
        // move
        move = true;
    }

    // MOVING / ENERGY CONSUMPTION
    // When move is true, Move towards the targetPos
    if (move == true)
    {
        // Move to targetPos
        transform.position = Vector2.MoveTowards(transform.position, targetPos, speed * Time.deltaTime);
        // This deals with the constant loss of energy from moving
        // size is now factored in because size effects how much energy is needed to move
        energy -= speed / (1f + size) * Time.timeScale;
    }
```

```
            else
            {
                // This deals with the constant loss of energy from resting
                energy -= (1 * Time.deltaTime);
            }

            // REPRODUCTION
            // if ready for replication then call reproduce()
            if (energy >= minEnergyForRep)
            {
                reproduce();
            }

            // PERIODIC BOUNDS
            // when the swapSides bool is true then start a timer that will wait for the creature to swap sides
            // this allows the creature to never become stuck in an endless state of swapping sides while also allowing normal sensing
            if (swapSides)
            {
                // Inititalize the count down timer
                hitBoundsTime = setHitBoundsTime;
                // Count down
                hitBoundsTime -= Time.deltaTime;
                // If hitBoundsTime hits 0
                if (hitBoundsTime <= 0)
                {
                    // Set Creature layer to its original layer
                    gameObject.layer = initialLayer;
                    // Stop the countdown by setting swapSides to false
                    swapSides = false;
                }
            }

            // DEATH FROM LACK OF ENERGY
            // this will run if the energy of the creature ever reaches 0
            if (energy <= 0)
                Destroy(gameObject);
        }

        private void FixedUpdate()
        {
            // SENSORY
            // This happens all of the time no matter what, and controls the states.
            // check for predators
            notSafe = Physics2D.OverlapCircle(transform.position, senseRadius, predators, 0);
            // check for food
            foodClose = Physics2D.OverlapCircle(transform.position, senseRadius, food, 0);

            // DECISION MAKING
            // priorities: #1 Make sure it is safe, #2 Keep energy up
            if (notSafe == false)
            {
                // CHeck if food is close and move towards it if so.
                if (foodClose == true)
                {
                    foodObjectPos = Physics2D.OverlapCircle(transform.position, senseRadius, food, 0).transform.position;
                    currentState = "GetFood";
                } // If food is not nearby then wander
                else
                {
                    currentState = "Wander";
                }
            } // If not then run away from predator.
            else
            {
                // get the opposite position and then set to predatorPos
                predatorPos = gameObject.transform.position - Physics2D.OverlapCircle(transform.position, senseRadius, predators,
0).gameObject.transform.position;
                if (Vector3.Distance(transform.position, predatorPos) < senseRadius)
                {
                    currentState = "Flee";
                }
                else
                {
                    currentState = "Wander";
                }
            }
        }

        // Generate new position
        void generateRandTargetPos()
```

```
{
    // Generate Random Position
    float randX = Random.Range(minX, maxX);
    float randY = Random.Range(minY, maxY);
    targetPos = new Vector2(randX, randY);
}

// Reproduce
void reproduce()
{
    // Create traits to passdown
    // SPEED DETERMINATION
    float offspringSpeed = speed;
    // mutations of speed
    int mutateSpeed = Random.Range(0, mutationRate);
    if (mutateSpeed == 1)
        // then mutate speed
        offspringSpeed = Random.Range(minSpeed, maxSpeed);

    // SIZE DETERMINATION
    float offspringSize = size;
    // mutations of size
    int mutateSize = Random.Range(0, mutationRate);
    if (mutateSize == 1)
        // then mutate size
        offspringSize = Random.Range(minSize, maxSize);

    // SENSORY DETERMINATION
    float offspringSenseRadius = senseRadius;
    // mutations of sense radius
    int mutateSenseRadius = Random.Range(0, mutationRate);
    if (mutateSenseRadius == 1)
        // then mutate sense radius
        offspringSenseRadius = Random.Range(minSenseRadius, maxSenseRadius);

    // Create Offspring
    GameObject offspring = Instantiate(asexualCreature, transform.position, Quaternion.identity, parentObjectOfOffspring);
    // give the newly detemrined trait values to the offspring
    offspring.GetComponent<AsexualCreature>().size = offspringSize;
    offspring.GetComponent<AsexualCreature>().speed = offspringSpeed;
    offspring.GetComponent<AsexualCreature>().senseRadius = offspringSenseRadius;
    offspring.GetComponent<AsexualCreature>().energy = 2500;

    // take away the energy that it takes to reproduce
    energy -= energyToRep;
}

// COLLISION
void OnCollisionEnter2D(Collision2D col)
{
    // Collision with food
    if (col.gameObject.CompareTag("Food"))
    {
        // check size of creature compared to food size
        // this ensures cratures only wat food that is smaller than them
        if (size >= 0.3f)
        {
            // Add energy
            energy += energyInFood;
            // destroy the food
            Destroy(col.gameObject);
        }
    }

    // Collision with periodic bounds
    // Deals with the layers
    // Does NOT deal with moving of the creature
    if (col.gameObject.CompareTag("Periodic"))
    {
        // Set Creature to a layer that can't be interacted with by periodicBounds layer
        gameObject.layer = 11;
        // Start timer to move back to original layer
        swapSides = true;
    }
}

// When Destroy() is called on this object
private void OnDestroy()
{
    // Update Statistics
```

```
        // Population
        CreatureStatistics.asexualCreatureCount -= 1;
        // Update Trait Stats
        // size
        CreatureStatistics.sizeDivisionTracker[sizeDiv] -= 1;
        // speed
        CreatureStatistics.speedDivisionTracker[speedDiv] -= 1;
        // sense radius
        CreatureStatistics.senseRadiusDivisionTracker[senseRadiusDiv] -= 1;
    }
}
```

Hermaphrodite Prey: (446 total lines, 412 sloc)

```
using UnityEngine;

public class HermaphroditeCreature : MonoBehaviour
{
    // TRAITS
    [Header("Trait Values")]
    public float speed;
    public float size;
    public float senseRadius;
    // tracking of traits
    private CreatureStatistics creatureStatistics;
    // trait divisions
    public int speedDiv;
    public int sizeDiv;
    public int senseRadiusDiv;

    // ENERGY SETTINGS
    [Header("Energy Settings")]
    public float minEnergyForRep;
    public float minEnergyToBeHungry;
    public float energyInFood;
    public float energyForRep;

    // RUN TIME EFFECTED VARIABLES
    [Header("Simulated Variables")]
    public float energy;
    public bool readyForRep;
    private float timeBtwRep;

    // MOVEMENT
    public bool move;
    private Vector2 targetPos;
    private float timeBtwDecision;
    public float setTimeBtwDecision;

    // PERIODIC BOUNDS
    public float setHitBoundsTime;
    private float hitBoundsTime;
    public bool swapSides;
    public int initialLayer;

    // MOVEMENT SETTINGS
    [Header("Movement Values")]
    public float maxX;
    public float maxY;
    public float minX;
    public float minY;

    // STATE
    [Header("State Variables")]
    public string currentState;

    // SENSORY
    [Header("Sensory Variables")]
    // check for predators
    public bool notSafe;
    public LayerMask predators;
    private Vector2 predatorPos;
    // check for food
    public bool foodClose;
    public LayerMask food;
    private Vector2 foodObjectPos;
    // check for mates
    public bool matesClose;
    public LayerMask mates;
    private Vector2 mateObjectPos;
```

```
// REPRODUCTION
[Header("Reproduction")]
public GameObject Hermaphrodite;
public float setTimeBtwRep;
private Transform parentObjectOfOffspring;
// Mutations
[Header("Mutations")]
public int mutationRate;
public float minSize;
public float maxSize;
public float minSpeed;
public float maxSpeed;
public float minSenseRadius;
public float maxSenseRadius;

// Start is called before the first frame update
void Start()
{
    //Initialize values
    // Set size based on the size gene
    transform.localScale = new Vector3(size, size, size);
    readyForRep = false;
    // allow movement
    move = true;
    // reset timer of decisions
    timeBtwDecision = 0;
    // reset swap side bool
    swapSides = false;
    // Set initialLayer
    initialLayer = gameObject.layer;
    // Get statistics reference
    creatureStatistics = FindObjectOfType<CreatureStatistics>();

    // Set timeBtwRep
    timeBtwRep = setTimeBtwRep;
    // Update statistics
    // POPULATIONS ones
    CreatureStatistics.allTimeHermaphroditeCreatureCount += 1;
    CreatureStatistics.hermaphroditeCreatureCount += 1;
    // TRAIT ones
    // Get Divisions
    // size
    sizeDiv = creatureStatistics.getSizeDivision(size);
    // speed
    speedDiv = creatureStatistics.getSpeedDivision(speed);
    // sense radius
    senseRadiusDiv = creatureStatistics.getSenseRadiusDivision(senseRadius);
    // Update Trait Stats
    // size
    CreatureStatistics.sizeDivisionTracker[sizeDiv] += 1;
    // speed
    CreatureStatistics.speedDivisionTracker[speedDiv] += 1;
    // sense radius
    CreatureStatistics.senseRadiusDivisionTracker[senseRadiusDiv] += 1;

    // Set parentObjectOfOffspring to the object holding all HERMAPHRODITE creatures
    parentObjectOfOffspring = GameObject.FindGameObjectWithTag("HermaphroditeHolder").transform;
}

// Update is called once per frame
void Update()
{
    // WANDER STATE
    // this state has the creature wander around the world.
    // In this state the creature can: Move, Rest.
    if (currentState == "Wander")
    {
        // Move or Not
        timeBtwDecision -= Time.deltaTime;
        if (timeBtwDecision <= 0)
        {
            // Reset Timer
            timeBtwDecision = setTimeBtwDecision;
            // Randomly Decide What To Do
            int randint = Random.Range(0, 2);
            if (randint == 0)
            {
                generateRandTargetPos();
                move = true;
```

```
        }
        else
            move = false;
    }
}

// FLEE STATE
// this state has the creature move away from predators.
// In this state the creature can: Move.
if (currentState == "Flee")
{
    // update targetPos to be the opposite of the predators direction
    targetPos = predatorPos;
    // move
    move = true;
}

// GETFOOD STATE
// this state has the creature move to the food piece it sensed.
// In this state the creature can: Move.
if (currentState == "GetFood")
{
    // Set targetPos to the food object.
    targetPos = foodObjectPos;
    // move
    move = true;
}

// GETMATE STATE
// this state has the creature move to a potential mate, and reporoduce
// In this state the creature can: Move, Reproduce.
if (currentState == "GetMate")
{
    // Set targetPos to potential mate
    targetPos = mateObjectPos;
    // move
    move = true;
    // IMPORTANT
    // reproduction only occurs on touch.
    // see OnCollisionEnter2D() for the calling of Reproduce()
}

// MOVING / ENERGY CONSUMPTION
// When move is true, Move towards the targetPos
if (move == true)
{
    // Move to targetPos
    transform.position = Vector2.MoveTowards(transform.position, targetPos, speed * Time.deltaTime);
    // This deals with the constant loss of energy from moving
    // size is now factored in because size effects how much energy is needed to move
    energy -= speed / (1f + size) * Time.timeScale;
}
else
{
    // This deals with the constant loss of energy from resting
    energy -= (1 * Time.deltaTime);
}

// Can Creature Reproduce?
timeBtwRep -= Time.deltaTime;
if (energy >= minEnergyForRep && timeBtwRep <= 0)
    readyForRep = true;
else
    readyForRep = false;

// PERIODIC BOUNDS
// when the swapSides bool is true then start a timer that will wait for the creature to swap sides
// this allows the creature to never become stuck in an endless state of swapping sides while also allowing normal sensing
if (swapSides)
{
    // Inititalize the count down timer
    hitBoundsTime = setHitBoundsTime;
    // Count down
    hitBoundsTime -= Time.deltaTime;
    // If hitBoundsTime hits 0
    if (hitBoundsTime <= 0)
    {
        // Set Creature layer to its original layer
        gameObject.layer = initialLayer;
        // Stop the countdown by setting swapSides to false
```

```
                swapSides = false;
            }
        }

        // DEATH FROM LACK OF ENERGY
        // this will run if the energy of the creature ever reaches 0
        if (energy <= 0)
            Destroy(gameObject);
    }

    private void FixedUpdate()
    {
        // SENSORY
        // This happens all of the time no matter what, and controls the states.
        // check for predators
        notSafe = Physics2D.OverlapCircle(transform.position, senseRadius, predators, 0);
        // check for food
        foodClose = Physics2D.OverlapCircle(transform.position, senseRadius, food, 0);
        // check for mates
        matesClose = Physics2D.OverlapCircle(transform.position, senseRadius, mates, 0);

        // DECISION MAKING
        // priorities: #1 Make sure it is safe, #2 Keep energy up, #3 Mate
        if (notSafe == false)
        {
            // If food or mate is close compare the options
            if (foodClose == true || matesClose == true)
            {
                // Check if Mating is a viable option or if energy is needed.
                if (energy < minEnergyForRep || energy < minEnergyToBeHungry || !matesClose)
                {
                    if (foodClose == true)
                    {
                        foodObjectPos = Physics2D.OverlapCircle(transform.position, senseRadius, food, 0).transform.position;
                        currentState = "GetFood";
                    }
                    else
                    {
                        currentState = "Wander";
                    }
                } // If Mating is viable than go to mate.
                else
                {
                    if (matesClose == true && readyForRep == true)
                    {
                        currentState = "GetMate";
                        GameObject potentialMate = Physics2D.OverlapCircle(transform.position, senseRadius, mates, 0).gameObject;
                        // If the sensed potential mate can also mate then move to the mate.
                            // && potentialMate != gameObject checks if the mate in question is not itself.
                            // This is VITAL because Physics2D.OverlapCircle() returns any objects within the specified LayerMask
                            // Meaning it can return the object it is being emitted from.
                            if (potentialMate.GetComponent<HermaphroditeCreature>().currentState == "GetMate" && potentialMate != gameObject)
                            {
                                mateObjectPos = Physics2D.OverlapCircle(transform.position, senseRadius, mates, 0).transform.position;
                            }
                    }
                    else
                    {
                        currentState = "Wander";
                    }
                }
            } // If not then wander
            else
            {
                currentState = "Wander";
            }
        } // If not then run away from predator.
        else
        {
            // get the opposite position and then set to predatorPos
            predatorPos = gameObject.transform.position - Physics2D.OverlapCircle(transform.position, senseRadius, predators,
0).gameObject.transform.position;
            if (Vector3.Distance(transform.position, predatorPos) < senseRadius)
            {
                currentState = "Flee";
            }
            else
            {
                currentState = "Wander";
            }
```

```
    }
}

// Generate new position
void generateRandTargetPos()
{
    // Generate Random Position
    float randX = Random.Range(minX, maxX);
    float randY = Random.Range(minY, maxY);
    targetPos = new Vector2(randX, randY);
}

// Reproduce
void reproduce(HermaphroditeCreature mateTraits)
{
    // Determine Speed Of Offspring
    float offspringSpeed = (mateTraits.speed + speed) / 2;
    // determine variation
    float speedVariation = Random.Range((offspringSpeed * -0.2f), (offspringSpeed * 0.2f));
    // apply variation
    offspringSpeed += speedVariation;
    // mutations of speed
    int mutateSpeed = Random.Range(0, mutationRate);
    if (mutateSpeed == 1)
        // then mutate speed
        offspringSpeed = Random.Range(minSpeed, maxSpeed);
    // final check
    // this makes sure the final traits don't go over set maximum or minimum values
    if (offspringSpeed < minSpeed)
        offspringSpeed = minSpeed;
    if (offspringSpeed > maxSpeed)
        offspringSpeed = maxSpeed;

    // Determine Size Of Offspring
    float offspringSize = (mateTraits.size + size) / 2;
    // determine variation
    float sizeVariation = Random.Range((offspringSize * -0.2f), (offspringSize * 0.2f));
    // apply variation
    offspringSize += sizeVariation;
    // mutations of size
    int mutateSize = Random.Range(0, mutationRate);
    if (mutateSize == 1)
        // then mutate size
        offspringSize = Random.Range(minSize, maxSize);
    // final check
    // this makes sure the final traits don't go over set maximum or minimum values
    if (offspringSize < minSize)
        offspringSize = minSize;
    if (offspringSize > maxSize)
        offspringSize = maxSize;

    // Determine Sense Radius Of Offspring
    float offspringSenseRadius = (mateTraits.senseRadius + senseRadius) / 2;
    // determine variation
    float senseRadiusVariation = Random.Range((offspringSenseRadius * -0.2f), (offspringSenseRadius * 0.2f));
    // apply variation
    offspringSenseRadius += senseRadiusVariation;
    // mutations of sense radius
    int mutateSenseRadius = Random.Range(0, mutationRate);
    if (mutateSenseRadius == 1)
        // then mutate sense radius
        offspringSenseRadius = Random.Range(minSenseRadius, maxSenseRadius);
    // final check
    // this makes sure the final traits don't go over set maximum or minimum values
    if (offspringSenseRadius < minSenseRadius)
        offspringSenseRadius = minSenseRadius;
    if (offspringSenseRadius > maxSenseRadius)
        offspringSenseRadius = maxSenseRadius;

    // SPAWNING OF NEW CREATURE (offSpring)
    GameObject offspring;
    offspring = Instantiate(Hermaphrodite, transform.position, Quaternion.identity, parentObjectOfOffspring);
    // give the newly detemrined trait values to the offspring
    offspring.GetComponent<HermaphroditeCreature>().size = offspringSize;
    offspring.GetComponent<HermaphroditeCreature>().speed = offspringSpeed;
    offspring.GetComponent<HermaphroditeCreature>().senseRadius = offspringSenseRadius;
    offspring.GetComponent<HermaphroditeCreature>().energy = 2500;

    // take away the energy that it takes to reproduce
    energy -= energyForRep;
```

```
        // esure the state doesn't become stuck in GetMate
        currentState = "Wander";
    }

    // COLLISION
    void OnCollisionEnter2D(Collision2D col)
    {
        // If Creature hits another alike Creature and Is in the GetMate State
        if (col.gameObject.CompareTag("HermaphroditeCreature") && currentState == "GetMate")
        {
            // Can Creature Reproduce?
            if (readyForRep == true)
            {
                reproduce(col.gameObject.GetComponent<HermaphroditeCreature>());
                timeBtwRep = setTimeBtwRep;
            }
        }

        // Collision with food
        if (col.gameObject.CompareTag("Food"))
        {
            // check size of creature compared to food size
            // this ensures cratures only wat food that is smaller than them
            if (size >= 0.3f)
            {
                // Add energy
                energy += energyInFood;
                // destroy the food
                Destroy(col.gameObject);
            }
        }

        // Collision with periodic bounds
        // Deals with the layers
        // Does NOT deal with moving of the creature
        if (col.gameObject.CompareTag("Periodic"))
        {
            // Set Creature to a layer that can't be interacted with by periodicBounds layer
            gameObject.layer = 11;
            // Start timer to move back to original layer
            swapSides = true;
        }
    }

    // When Destroy() is called on this object
    private void OnDestroy()
    {
        // Update Statistics
        // Population
        CreatureStatistics.hermaphroditeCreatureCount -= 1;
        // Update Trait Stats
        // size
        CreatureStatistics.sizeDivisionTracker[sizeDiv] -= 1;
        // speed
        CreatureStatistics.speedDivisionTracker[speedDiv] -= 1;
        // sense radius
        CreatureStatistics.senseRadiusDivisionTracker[senseRadiusDiv] -= 1;
    }
}
```

Sexual Male Prey: (369 total lines, 344 sloc)

```
using UnityEngine;

public class SexualCreatureMale : MonoBehaviour
{
    // TRAITS
    [Header("Trait Values")]
    public float speed;
    public float size;
    public float senseRadius;
    public string[] chromosomes;
    // tracking of traits
    private CreatureStatistics creatureStatistics;
    // trait divisions
    public int speedDiv;
    public int sizeDiv;
    public int senseRadiusDiv;

    // ENERGY SETTINGS
```

```
[Header("Energy Settings")]
public float minEnergyForRep;
public float minEnergyToBeHungry;
public float energyInFood;
public float energyToRep;

// RUN TIME EFFECTED VARIABLES
[Header("Simulated Variables")]
public float energy;
public bool readyForRep;
private float timeBtwRep;
// MOVEMENT
public bool move;
private Vector2 targetPos;
private float timeBtwDecision;
public float setTimeBtwDecision;
// PERIODIC BOUNDS
public float setHitBoundsTime;
private float hitBoundsTime;
public bool swapSides;
public int initialLayer;

// MOVEMENT SETTINGS
[Header("Movement Values")]
public float maxX;
public float maxY;
public float minX;
public float minY;

// STATE
[Header("State Variables")]
public string currentState;

// SENSORY
[Header("Sensory Variables")]
// check for predators
public bool notSafe;
public LayerMask predators;
private Vector2 predatorPos;
// check for food
public bool foodClose;
public LayerMask food;
private Vector2 foodObjectPos;
// check for mates
public bool matesClose;
public LayerMask mates;
private Vector2 mateObjectPos;

// REPRODUCTION
public float setTimeBtwRep;

// Start is called before the first frame update
void Start()
{
    //Initialize values
    // Set size based on the size gene
    transform.localScale = new Vector3(size, size, size);
    readyForRep = false;
    // allow movement
    move = true;
    // reset timer of decisions
    timeBtwDecision = 0;
    // reset swap side bool
    swapSides = false;
    // Set initialLayer
    initialLayer = gameObject.layer;
    // Get statistics reference
    creatureStatistics = FindObjectOfType<CreatureStatistics>();

    // Set chromosomes
    chromosomes = new string[2];
    chromosomes[0] = "X";
    chromosomes[1] = "Y";
    // Set timeBtwRep
    timeBtwRep = setTimeBtwRep;
    // Update statistics
    // POPULATIONS ones
    CreatureStatistics.sexualCreatureCount += 1;
    CreatureStatistics.maleSexualCreatureCount += 1;
    CreatureStatistics.allTimeMaleCreatureCount += 1;
```

```
    CreatureStatistics.allTimeSexualCreatureCount += 1;
    // TRAIT ones
    // Get Divisions
    // size
    sizeDiv = creatureStatistics.getSizeDivision(size);
    // speed
    speedDiv = creatureStatistics.getSpeedDivision(speed);
    // sense radius
    senseRadiusDiv = creatureStatistics.getSenseRadiusDivision(senseRadius);
    // Update Trait Stats
    // size
    CreatureStatistics.sizeDivisionTracker[sizeDiv] += 1;
    // speed
    CreatureStatistics.speedDivisionTracker[speedDiv] += 1;
    // sense radius
    CreatureStatistics.senseRadiusDivisionTracker[senseRadiusDiv] += 1;
}

// Update is called once per frame
void Update()
{
    // WANDER STATE
    // this state has the creature wander around the world.
    // In this state the creature can: Move, Rest.
    if (currentState == "Wander")
    {
        // Move or Not
        timeBtwDecision -= Time.deltaTime;
        if (timeBtwDecision <= 0)
        {
            // Reset Timer
            timeBtwDecision = setTimeBtwDecision;
            // Randomly Decide What To Do
            int randint = Random.Range(0, 2);
            if (randint == 0)
            {
                generateRandTargetPos();
                move = true;
            }
            else
                move = false;
        }
    }

    // FLEE STATE
    // this state has the creature move away from predators.
    // In this state the creature can: Move.
    if (currentState == "Flee")
    {
        // update targetPos to be the opposite of the predators direction
        targetPos = predatorPos;
        // move
        move = true;
    }

    // GETFOOD STATE
    // this state has the creature move to the food piece it sensed.
    // In this state the creature can: Move.
    if (currentState == "GetFood")
    {
        // Set targetPos to the food object.
        targetPos = foodObjectPos;
        // move
        move = true;
    }

    // GETMATE STATE
    // this state has the creature move to a potential mate, and reporoduce
    // In this state the creature can: Move, Reproduce.
    if (currentState == "GetMate")
    {
        // Set targetPos to potential mate
        targetPos = mateObjectPos;
        // move
        move = true;
        // IMPORTANT
        // reproduction only occurs on touch.
        // see OnCollisionEnter2D() for the calling of Reproduce()
    }
```

```
    // MOVING / ENERGY CONSUMPTION
    // When move is true, Move towards the targetPos
    if (move == true)
    {
        // Move to targetPos
        transform.position = Vector2.MoveTowards(transform.position, targetPos, speed * Time.deltaTime);
        // This deals with the constant loss of energy from moving
        // size is now factored in because size effects how much energy is needed to move
        energy -= speed / (1f + size) * Time.timeScale;
    }
    else
    {
        // This deals with the constant loss of energy from resting
        energy -= (1 * Time.deltaTime);
    }

    // Can Creature Reproduce?
    timeBtwRep -= Time.deltaTime;
    if (energy >= minEnergyForRep && timeBtwRep <= 0)
        readyForRep = true;
    else
        readyForRep = false;

    // PERIODIC BOUNDS
    // when the swapSides bool is true then start a timer that will wait for the creature to swap sides
    // this allows the creature to never become stuck in an endless state of swapping sides while also allowing normal sensing
    if (swapSides)
    {
        // Inititalize the count down timer
        hitBoundsTime = setHitBoundsTime;
        // Count down
        hitBoundsTime -= Time.deltaTime;
        // If hitBoundsTime hits 0
        if (hitBoundsTime <= 0)
        {
            // Set Creature layer to its original layer
            gameObject.layer = initialLayer;
            // Stop the countdown by setting swapSides to false
            swapSides = false;
        }
    }

    // DEATH FROM LACK OF ENERGY
    // this will run if the energy of the creature ever reaches 0
    if (energy <= 0)
        Destroy(gameObject);
}

private void FixedUpdate()
{
    // SENSORY
    // This happens all of the time no matter what, and controls the states.
    // check for predators
    notSafe = Physics2D.OverlapCircle(transform.position, senseRadius, predators, 0);
    // check for food
    foodClose = Physics2D.OverlapCircle(transform.position, senseRadius, food, 0);
    // check for mates
    matesClose = Physics2D.OverlapCircle(transform.position, senseRadius, mates, 0);

    // DECISION MAKING
    // priorities: #1 Make sure it is safe, #2 Keep energy up, #3 Mate
    if (notSafe == false)
    {
        // If food or mate is close compare the options
        if (foodClose == true || matesClose == true)
        {
            // Check if Mating is a viable option or if energy is needed.
            if (energy < minEnergyForRep || energy < minEnergyToBeHungry || !matesClose)
            {
                if (foodClose == true)
                {
                    foodObjectPos = Physics2D.OverlapCircle(transform.position, senseRadius, food, 0).transform.position;
                    currentState = "GetFood";
                }
                else
                {
                    currentState = "Wander";
                }
            } // If Mating is viable than go to mate.
            else
```

```
                    {
                        if (matesClose == true && readyForRep)
                        {
                            currentState = "GetMate";
                            GameObject potentialMate = Physics2D.OverlapCircle(transform.position, senseRadius, mates, 0).gameObject;
                            // If the sensed potential mate can also mate then move to the mate.
                            if (potentialMate.GetComponent<SexualCreatureFemale>().currentState == "GetMate")
                            {
                                mateObjectPos = Physics2D.OverlapCircle(transform.position, senseRadius, mates, 0).transform.position;
                            }
                        }
                        else
                        {
                            currentState = "Wander";
                        }
                    }
                } // If not then wander
                else
                {
                    currentState = "Wander";
                }
            } // If not then run away from predator.
            else
            {
                // get the opposite position and then set to predatorPos
                predatorPos = gameObject.transform.position - Physics2D.OverlapCircle(transform.position, senseRadius, predators,
0).gameObject.transform.position;
                if (Vector3.Distance(transform.position, predatorPos) < senseRadius)
                {
                    currentState = "Flee";
                }
                else
                {
                    currentState = "Wander";
                }
            }
        }

        // Generate new position
        void generateRandTargetPos()
        {
            // Generate Random Position
            float randX = Random.Range(minX, maxX);
            float randY = Random.Range(minY, maxY);
            targetPos = new Vector2(randX, randY);
        }

        // Reproduce
        void reproduce()
        {
            energy -= energyToRep;
            currentState = "Wander";
            // IMPORTANT
            // actual sharing of traits, and instantiating of new creatures is done in the female creature's reproduce() function.
        }

        // COLLISION
        void OnCollisionEnter2D(Collision2D col)
        {
            // If Creature hits another alike Creature and Is in the GetMate State
            if (col.gameObject.CompareTag("Female") && currentState == "GetMate")
            {
                // Can Creature Reproduce?
                if (readyForRep == true)
                {
                    reproduce();
                    timeBtwRep = setTimeBtwRep;
                }
            }

            // Collision with food
            if(col.gameObject.CompareTag("Food"))
            {
                // check size of creature compared to food size
                // this ensures cratures only wat food that is smaller than them
                if (size >= 0.3f)
                {
                    // Add energy
                    energy += energyInFood;
                    // destroy the food
```

```
                Destroy(col.gameObject);
            }
        }

        // Collision with periodic bounds
        // Deals with the layers
        // Does NOT deal with moving of the creature
        if(col.gameObject.CompareTag("Periodic"))
        {
            // Set Creature to a layer that can't be interacted with by periodicBounds layer
            gameObject.layer = 11;
            // Start timer to move back to original layer
            swapSides = true;
        }
    }

    // When Destroy() is called on this object
    private void OnDestroy()
    {
        // Update Statistics
        // Population
        CreatureStatistics.sexualCreatureCount -= 1;
        CreatureStatistics.maleSexualCreatureCount -= 1;
        // Update Trait Stats
        // size
        CreatureStatistics.sizeDivisionTracker[sizeDiv] -= 1;
        // speed
        CreatureStatistics.speedDivisionTracker[speedDiv] -= 1;
        // sense radius
        CreatureStatistics.senseRadiusDivisionTracker[senseRadiusDiv] -= 1;
    }
}
```

Sexual Female Prey: (474 total lines, 440 sloc)

```
using UnityEngine;

public class SexualCreatureFemale : MonoBehaviour
{
    // TRAITS
    [Header("Trait Values")]
    public float speed;
    public float size;
    public float senseRadius;
    public string[] chromosomes;
    // tracking of traits
    private CreatureStatistics creatureStatistics;
    // trait divisions
    public int speedDiv;
    public int sizeDiv;
    public int senseRadiusDiv;

    // ENERGY SETTINGS
    [Header("Energy Settings")]
    public float minEnergyForRep;
    public float minEnergyToBeHungry;
    public float energyInFood;
    public float energyForRep;

    // RUN TIME EFFECTED VARIABLES
    [Header("Simulated Variables")]
    public float energy;
    public bool readyForRep;
    private float timeBtwRep;
    // MOVEMENT
    public bool move;
    private Vector2 targetPos;
    private float timeBtwDecision;
    public float setTimeBtwDecision;

    // PERIODIC BOUNDS
    public float setHitBoundsTime;
    private float hitBoundsTime;
    public bool swapSides;
    public int initialLayer;

    // MOVEMENT SETTINGS
    [Header("Movement Values")]
    public float maxX;
    public float maxY;
```

```
public float minX;
public float minY;

// STATE
[Header("State Variables")]
public string currentState;

// SENSORY
[Header("Sensory Variables")]
// check for predators
public bool notSafe;
public LayerMask predators;
private Vector2 predatorPos;
// check for food
public bool foodClose;
public LayerMask food;
private Vector2 foodObjectPos;
// check for mates
public bool matesClose;
public LayerMask mates;
private Vector2 mateObjectPos;

// REPRODUCTION
[Header("Reproduction")]
public GameObject SexualMale;
public GameObject SexualFemale;
public float setTimeBtwRep;
// parent object holding all spawned offspring
private Transform parentObjectOfOffspring;
// Mutations
[Header("Mutations")]
public int mutationRate;
public float minSize;
public float maxSize;
public float minSpeed;
public float maxSpeed;
public float minSenseRadius;
public float maxSenseRadius;

// Start is called before the first frame update
void Start()
{
    //Initialize values
    // Set size based on the size gene
    transform.localScale = new Vector3(size, size, size);
    readyForRep = false;
    // allow movement
    move = true;
    // reset timer of decisions
    timeBtwDecision = 0;
    // reset swap side bool
    swapSides = false;
    // Set initialLayer
    initialLayer = gameObject.layer;
    // Get statistics reference
    creatureStatistics = FindObjectOfType<CreatureStatistics>();

    // Set chromosomes
    chromosomes = new string[2];
    chromosomes[0] = "X";
    chromosomes[1] = "X";
    // Set timeBtwRep
    timeBtwRep = setTimeBtwRep;
    // Update statistics
    // POPULATIONS ones
    CreatureStatistics.sexualCreatureCount += 1;
    CreatureStatistics.femaleSexualCreatureCount += 1;
    CreatureStatistics.allTimeFemaleCreatureCount += 1;
    CreatureStatistics.allTimeSexualCreatureCount += 1;
    // TRAIT ones
    // Get Divisions
    // size
    sizeDiv = creatureStatistics.getSizeDivision(size);
    // speed
    speedDiv = creatureStatistics.getSpeedDivision(speed);
    // sense radius
    senseRadiusDiv = creatureStatistics.getSenseRadiusDivision(senseRadius);
    // Update Trait Stats
    // size
    CreatureStatistics.sizeDivisionTracker[sizeDiv] += 1;
```

```
    // speed
    CreatureStatistics.speedDivisionTracker[speedDiv] += 1;
    // sense radius
    CreatureStatistics.senseRadiusDivisionTracker[senseRadiusDiv] += 1;

    // Set parentObjectOfOffspring to the object holding all SEXUAL creatures
    parentObjectOfOffspring = GameObject.FindGameObjectWithTag("SexualCreatureHolder").transform;
}

// Update is called once per frame
void Update()
{
    // WANDER STATE
    // this state has the creature wander around the world.
    // In this state the creature can: Move, Rest.
    if (currentState == "Wander")
    {
        // Move or Not
        timeBtwDecision -= Time.deltaTime;
        if (timeBtwDecision <= 0)
        {
            // Reset Timer
            timeBtwDecision = setTimeBtwDecision;
            // Randomly Decide What To Do
            int randint = Random.Range(0, 2);
            if (randint == 0)
            {
                generateRandTargetPos();
                move = true;
            }
            else
                move = false;
        }
    }

    // FLEE STATE
    // this state has the creature move away from predators.
    // In this state the creature can: Move.
    if (currentState == "Flee")
    {
        // update targetPos to be the opposite of the predators direction
        targetPos = predatorPos;
        // move
        move = true;
    }

    // GETFOOD STATE
    // this state has the creature move to the food piece it sensed.
    // In this state the creature can: Move.
    if (currentState == "GetFood")
    {
        // Set targetPos to the food object.
        targetPos = foodObjectPos;
        // move
        move = true;
    }

    // GETMATE STATE
    // this state has the creature move to a potential mate, and reporoduce
    // In this state the creature can: Move, Reproduce.
    if (currentState == "GetMate")
    {
        // Set targetPos to potential mate
        targetPos = mateObjectPos;
        // move
        move = true;
        // IMPORTANT
        // reproduction only occurs on touch.
        // see OnCollisionEnter2D() for the calling of Reproduce()
    }

    // MOVING / ENERGY CONSUMPTION
    // When move is true, Move towards the targetPos
    if (move == true)
    {
        // Move to targetPos
        transform.position = Vector2.MoveTowards(transform.position, targetPos, speed * Time.deltaTime);
        // This deals with the constant loss of energy from moving
        // size is now factored in because size effects how much energy is needed to move
        energy -= speed / (1f + size) * Time.timeScale;
```

```
    }
    else
    {
        // This deals with the constant loss of energy from resting
        energy -= (1 * Time.deltaTime);
    }

    // Can Creature Reproduce?
    timeBtwRep -= Time.deltaTime;
    if (energy >= minEnergyForRep && timeBtwRep <= 0)
        readyForRep = true;
    else
        readyForRep = false;

    // PERIODIC BOUNDS
    // when the swapSides bool is true then start a timer that will wait for the creature to swap sides
    // this allows the creature to never become stuck in an endless state of swapping sides while also allowing normal sensing
    if (swapSides)
    {
        // Inititalize the count down timer
        hitBoundsTime = setHitBoundsTime;
        // Count down
        hitBoundsTime -= Time.deltaTime;
        // If hitBoundsTime hits 0
        if (hitBoundsTime <= 0)
        {
            // Set Creature layer to its original layer
            gameObject.layer = initialLayer;
            // Stop the countdown by setting swapSides to false
            swapSides = false;
        }
    }

    // DEATH FROM LACK OF ENERGY
    // this will run if the energy of the creature ever reaches 0
    if (energy <= 0)
        Destroy(gameObject);
}

private void FixedUpdate()
{
    // SENSORY
    // This happens all of the time no matter what, and controls the states.
    // check for predators
    notSafe = Physics2D.OverlapCircle(transform.position, senseRadius, predators, 0);
    // check for food
    foodClose = Physics2D.OverlapCircle(transform.position, senseRadius, food, 0);
    // check for mates
    matesClose = Physics2D.OverlapCircle(transform.position, senseRadius, mates, 0);

    // DECISION MAKING
    // priorities: #1 Make sure it is safe, #2 Keep energy up, #3 Mate
    if (notSafe == false)
    {
        // If food or mate is close compare the options
        if (foodClose == true || matesClose == true)
        {
            // Check if Mating is a viable option or if energy is needed.
            if (energy < minEnergyForRep || energy < minEnergyToBeHungry || !matesClose)
            {
                if (foodClose == true)
                {
                    foodObjectPos = Physics2D.OverlapCircle(transform.position, senseRadius, food, 0).transform.position;
                    currentState = "GetFood";
                }
                else
                {
                    currentState = "Wander";
                }
            } // If Mating is viable than go to mate.
            else
            {
                if (matesClose == true && readyForRep)
                {
                    currentState = "GetMate";
                    GameObject potentialMate = Physics2D.OverlapCircle(transform.position, senseRadius, mates, 0).gameObject;
                    // If the sensed potential mate can also mate then move to the mate.
                    if (potentialMate.GetComponent<SexualCreatureMale>().currentState == "GetMate")
                    {
                        mateObjectPos = Physics2D.OverlapCircle(transform.position, senseRadius, mates, 0).transform.position;
```

```
                        }
                    }
                    else
                    {
                        currentState = "Wander";
                    }
                }
            } // If not then wander
            else
            {
                currentState = "Wander";
            }
        } // If not then run away from predator.
        else
        {
            // get the opposite position and then set to predatorPos
            predatorPos = gameObject.transform.position - Physics2D.OverlapCircle(transform.position, senseRadius, predators,
0).gameObject.transform.position;
            if (Vector3.Distance(transform.position, predatorPos) < senseRadius)
            {
                currentState = "Flee";
            }
            else
            {
                currentState = "Wander";
            }
        }
    }

    // Generate new position
    void generateRandTargetPos()
    {
        // Generate Random Position
        float randX = Random.Range(minX, maxX);
        float randY = Random.Range(minY, maxY);
        targetPos = new Vector2(randX, randY);
    }

    // Reproduce
    void reproduce(SexualCreatureMale mateTraits)
    {
        // Decide Sex Of Offspring
        int randChromosome = Random.Range(0, 2);
        string donatedChromosome = mateTraits.chromosomes[randChromosome];

        // FINAL SEX OF OFFSPRING
        string[] offspringChromosomes = new string[2];
        offspringChromosomes[0] = "X";
        offspringChromosomes[1] = donatedChromosome;

        // Determine Speed Of Offspring
        float offspringSpeed = (mateTraits.speed + speed)/2;
        // determine variation
        float speedVariation = Random.Range((offspringSpeed * -0.2f), (offspringSpeed * 0.2f));
        // apply variation
        offspringSpeed += speedVariation;
        // mutations of speed
        int mutateSpeed = Random.Range(0, mutationRate);
        if (mutateSpeed == 1)
            // then mutate speed
            offspringSpeed = Random.Range(minSpeed, maxSpeed);
        // final check
        // this makes sure the final traits don't go over set maximum or minimum values
        if (offspringSpeed< minSpeed)
            offspringSpeed = minSpeed;
        if (offspringSpeed > maxSpeed)
            offspringSpeed = maxSpeed;

        // Determine Size Of Offspring
        float offspringSize = (mateTraits.size + size)/2;
        // determine variation
        float sizeVariation = Random.Range((offspringSize * -0.2f), (offspringSize * 0.2f));
        // apply variation
        offspringSize += sizeVariation;
        // mutations of size
        int mutateSize = Random.Range(0, mutationRate);
        if (mutateSize == 1)
            // then mutate size
            offspringSize = Random.Range(minSize, maxSize);
        // final check
```

```
    // this makes sure the final traits don't go over set maximum or minimum values
    if (offspringSize < minSize)
        offspringSize = minSize;
    if (offspringSize > maxSize)
        offspringSize = maxSize;

    // Determine Sense Radius Of Offspring
    float offspringSenseRadius = (mateTraits.senseRadius + senseRadius)/2;
    // determine variation
    float senseRadiusVariation = Random.Range((offspringSenseRadius * -0.2f), (offspringSenseRadius * 0.2f));
    // apply variation
    offspringSenseRadius += senseRadiusVariation;
    // mutations of sense radius
    int mutateSenseRadius = Random.Range(0, mutationRate);
    if (mutateSenseRadius == 1)
        // then mutate sense radius
        offspringSenseRadius = Random.Range(minSenseRadius, maxSenseRadius);
    // final check
    // this makes sure the final traits don't go over set maximum or minimum values
    if (offspringSenseRadius < minSenseRadius)
        offspringSenseRadius = minSenseRadius;
    if (offspringSenseRadius > maxSenseRadius)
        offspringSenseRadius = maxSenseRadius;

    // SPAWNING OF NEW CREATURE (offSpring)
    GameObject offspring;
    if (offspringChromosomes[1] == "Y")
    {
        // the offspring should be a MALE
        offspring = Instantiate(SexualMale, transform.position, Quaternion.identity, parentObjectOfOffspring);
        // give the newly detemrined trait values to the offspring
        offspring.GetComponent<SexualCreatureMale>().size = offspringSize;
        offspring.GetComponent<SexualCreatureMale>().speed = offspringSpeed;
        offspring.GetComponent<SexualCreatureMale>().senseRadius = offspringSenseRadius;
        offspring.GetComponent<SexualCreatureMale>().energy = 2500;
    }
    else if(offspringChromosomes[1] == "X")
    {
        // the offspring should be a FEMALE
        offspring = Instantiate(SexualFemale, transform.position, Quaternion.identity, parentObjectOfOffspring);
        // give the newly detemrined trait values to the offspring
        offspring.GetComponent<SexualCreatureFemale>().size = offspringSize;
        offspring.GetComponent<SexualCreatureFemale>().speed = offspringSpeed;
        offspring.GetComponent<SexualCreatureFemale>().senseRadius = offspringSenseRadius;
        offspring.GetComponent<SexualCreatureFemale>().energy = 2500;
    }

    // take away the energy that it takes to reproduce
    energy -= energyForRep;
    // esure the state doesn't become stuck in GetMate
    currentState = "Wander";
}

// COLLISION
void OnCollisionEnter2D(Collision2D col)
{
    // If Creature hits another alike Creature and Is in the GetMate State
    if (col.gameObject.CompareTag("Male") && currentState == "GetMate")
    {
        // Can Creature Reproduce?
        if (readyForRep == true)
        {
            reproduce(col.gameObject.GetComponent<SexualCreatureMale>());
            timeBtwRep = setTimeBtwRep;
        }
    }

    // Collision with food
    if (col.gameObject.CompareTag("Food"))
    {
        // check size of creature compared to food size
        // this ensures cratures only wat food that is smaller than them
        if(size >= 0.3f)
        {
            // Add energy
            energy += energyInFood;
            // destroy the food
            Destroy(col.gameObject);
        }
    }
```

```
                // Collision with periodic bounds
                // Deals with the layers
                // Does NOT deal with moving of the creature
                if (col.gameObject.CompareTag("Periodic"))
                {
                    // Set Creature to a layer that can't be interacted with by periodicBounds layer
                    gameObject.layer = 11;
                    // Start timer to move back to original layer
                    swapSides = true;
                }
            }

            // When Destroy() is called on this object
            private void OnDestroy()
            {
                // Update Statistics
                // Population
                CreatureStatistics.sexualCreatureCount -= 1;
                CreatureStatistics.femaleSexualCreatureCount -= 1;
                // Update Trait Stats
                // size
                CreatureStatistics.sizeDivisionTracker[sizeDiv] -= 1;
                // speed
                CreatureStatistics.speedDivisionTracker[speedDiv] -= 1;
                // sense radius
                CreatureStatistics.senseRadiusDivisionTracker[senseRadiusDiv] -= 1;
            }
        }
```

Predator Spawning: (46 total lines, 42 sloc)

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

public class PredatorSpawning : MonoBehaviour
{
    // PREDATORS
    [Header("Predators")]
    public GameObject[] predators;
    // predator holders
    public Transform[] predatorHolders;

    // BOUNDS
    [Header("Boundaries")]
    // min and max (x,y) values for food spawn locations
    public float maxX;
    public float maxY;
    public float minX;
    public float minY;

    // SPAWNER
    // this function is called from the SpawnInitialPredators() function in MenuManager which is called when the button "Run Simulation Button" is pressed
    public void spawnInitialPredators(int predatorCount, int activePredator)
    {
        for (int i = 0; i < predatorCount; i++)
        {
            // generate random position
            float x = Random.Range(minX, maxX);
            float y = Random.Range(minY, maxY);
            Vector3 spawnPos = new Vector3(x, y, 1);

            // spawn the predator
            // if the predator is NOT sexual
            if (activePredator != 0)
                Instantiate(predators[activePredator], spawnPos, Quaternion.identity, predatorHolders[activePredator]);
            else // else if the predator is sexual
            {
                // Ensures a close to even if not perfectly even distribution of females, and males.
                if(i % 2 != 0) // spawn a female
                    Instantiate(predators[0], spawnPos, Quaternion.identity, predatorHolders[activePredator]);
                else // spawn a male
                    Instantiate(predators[3], spawnPos, Quaternion.identity, predatorHolders[activePredator]);
            }
        }
    }
}
```

Asexual Predator: (371 total lines, 340 sloc)

```
using UnityEngine;

public class PredatorAsexual : MonoBehaviour
{
    // TRAITS
    [Header("Trait Values")]
    public float speed;
    public float size;
    // tracking of traits
    private PredatorStatistics predatorStatistics;
    // trait divisions
    public int speedDiv;
    public int sizeDiv;
    public int senseRadiusDiv;

    // ENERGY SETTINGS
    [Header("Energy Settings")]
    public float minEnergyForRep;
    public float minEnergyToBeHungry;
    public float energyInFood;
    public float energyForRep;

    // CATCHING PREY
    [Header("Catching Prey")]
    public float setRestTime;
    public bool rest;

    // RUN TIME EFFECTED VARIABLES
    [Header("Simulated Variables")]
    public float energy;
    public bool readyForRep;
    private float restTime;
    // MOVEMENT
    public bool move;
    private Vector2 targetPos;
    private float timeBtwDecision;
    public float setTimeBtwDecision;

    // PERIODIC BOUNDS
    public float setHitBoundsTime;
    private float hitBoundsTime;
    public bool swapSides;
    public int initialLayer;

    // MOVEMENT SETTINGS
    [Header("Movement Values")]
    public float maxX;
    public float maxY;
    public float minX;
    public float minY;

    // STATE
    [Header("State Variables")]
    public string currentState;

    // SENSORY
    [Header("Sensory Variables")]
    public float senseRadius;
    // check for food
    public bool foodClose;
    public LayerMask food;
    private Vector2 foodObjectPos;

    // REPRODUCTION
    [Header("Reproduction")]
    public GameObject predatorAsexual;
    // parent object holding all spawned offspring
    private Transform parentObjectOfOffspring;
    // Mutations
    [Header("Mutations")]
    public int mutationRate;
    public float minSize;
    public float maxSize;
    public float minSpeed;
    public float maxSpeed;
    public float minSenseRadius;
    public float maxSenseRadius;

    // Start is called before the first frame update
    void Start()
```

```
{
    //Initialize values
    // Set size based on the size gene
    transform.localScale = new Vector3(size, size, size);
    readyForRep = false;
    // allow movement
    move = true;
    // reset timer of decisions
    timeBtwDecision = 0;
    // reset swap side bool
    swapSides = false;
    // Set initialLayer
    initialLayer = gameObject.layer;
    // Get statistics reference
    predatorStatistics = FindObjectOfType<PredatorStatistics>();

    // Set restTime
    restTime = setRestTime;
    // Set rest
    rest = false;
    // Update statistics
    // POPULATIONS ones
    PredatorStatistics.allTimePredatorAsexualCount += 1;
    PredatorStatistics.predatorAsexualCount += 1;
    // TRAIT ones
    // Get Divisions
    // size
    sizeDiv = predatorStatistics.getSizeDivision(size);
    // speed
    speedDiv = predatorStatistics.getSpeedDivision(speed);
    // sense radius
    senseRadiusDiv = predatorStatistics.getSenseRadiusDivision(senseRadius);
    // Update Trait Stats
    // size
    CreatureStatistics.sizeDivisionTracker[sizeDiv] += 1;
    // speed
    CreatureStatistics.speedDivisionTracker[speedDiv] += 1;
    // sense radius
    CreatureStatistics.senseRadiusDivisionTracker[senseRadiusDiv] += 1;

    // Set parentObjectOfOffspring to the object holding all PREDATOR creatures
    parentObjectOfOffspring = GameObject.FindGameObjectWithTag("PredatorAsexualHolder").transform;
}

// Update is called once per frame
void Update()
{
    // WANDER STATE
    // this state has the predator wander around the world.
    // In this state the predator can: Move, Rest.
    if (currentState == "Wander")
    {
        // Move or Not
        timeBtwDecision -= Time.deltaTime;
        if (timeBtwDecision <= 0)
        {
            // Reset Timer
            timeBtwDecision = setTimeBtwDecision;
            // Randomly Decide What To Do
            int randint = Random.Range(0, 2);
            if (randint == 0)
            {
                generateRandTargetPos();
                move = true;
            }
            else
                move = false;
        }
    }

    // GETFOOD STATE
    // this state has the predator move to the food piece it sensed.
    // In this state the predator can: Move.
    if (currentState == "GetFood")
    {
        // Set targetPos to the food object.
        targetPos = foodObjectPos;
        // move
        move = true;
    }
```

```
    // MOVING / ENERGY CONSUMPTION
    // When move is true AND rest is false, Move towards the targetPos
    if (move == true && rest == false)
    {
        // Move to targetPos
        transform.position = Vector2.MoveTowards(transform.position, targetPos, speed * Time.deltaTime);
        // This deals with the constant loss of energy from moving
        // size is now factored in because size effects how much energy is needed to move
        energy -= speed / (1f + size) * Time.timeScale;
    }
    else
    {
        // This deals with the constant loss of energy from resting
        energy -= (1 * Time.deltaTime);
    }

    // REPRODUCTION
    // if ready for replication then call reproduce()
    if (energy >= minEnergyForRep)
    {
        reproduce();
    }

    // PERIODIC BOUNDS
    // when the swapSides bool is true then start a timer that will wait for the predator to swap sides
    // this allows the predator to never become stuck in an endless state of swapping sides while also allowing normal sensing
    if (swapSides)
    {
        // Inititalize the count down timer
        hitBoundsTime = setHitBoundsTime;
        // Count down
        hitBoundsTime -= Time.deltaTime;
        // If hitBoundsTime hits 0
        if (hitBoundsTime <= 0)
        {
            // Set Predator layer to its original layer
            gameObject.layer = initialLayer;
            // Stop the countdown by setting swapSides to false
            swapSides = false;
        }
    }

    // DEATH FROM LACK OF ENERGY
    // this will run if the energy of the predator ever reaches 0
    if (energy <= 0)
    {
        Destroy(gameObject);
    }

    // FORCED REST
    // this only happens when this predator fails to catch prey (food)
    if (rest == true)
    {
        // start countdown of rest
        restTime -= Time.deltaTime;
        if (restTime <= 0)
        {
            rest = false;
            // reset rest time
            restTime = setRestTime;
        }
    }
}

private void FixedUpdate()
{
    // SENSORY
    // This happens all of the time no matter what, and controls the states.
    // check for food
    foodClose = Physics2D.OverlapCircle(transform.position, senseRadius, food, 0);

    // DECISION MAKING
    // priorities: #1 Keep energy up
    if (foodClose == true)
    {
        if (foodClose == true)
        {
            if (foodClose == true)
            {
```

```
                GameObject potentialPrey = Physics2D.OverlapCircle(transform.position, senseRadius, food, 0).gameObject;
                // check size of predator compared to food size
                // this ensures predators only eat food that is smaller than them
                // if the sensed prey is smaller then or the same size as the predator (itself) then chase the prey
                if (potentialPrey.transform.localScale.x <= size)
                {
                    foodObjectPos = potentialPrey.transform.position;
                    currentState = "GetFood";
                }
                else // if not then wander
                {
                    currentState = "Wander";
                }
            }
        } // If food is not nearby wander
        else
        {
            currentState = "Wander";
        }
    } // If not then wander
    else
    {
        currentState = "Wander";
    }
}

// Generate new position
void generateRandTargetPos()
{
    // Generate Random Position
    float randX = Random.Range(minX, maxX);
    float randY = Random.Range(minY, maxY);
    targetPos = new Vector2(randX, randY);
}

// Reproduce
void reproduce()
{
    // Create traits to passdown
    // SPEED DETERMINATION
    float offspringSpeed = speed;
    // mutations of speed
    int mutateSpeed = Random.Range(0, mutationRate);
    if (mutateSpeed == 1)
        // then mutate speed
        offspringSpeed = Random.Range(minSpeed, maxSpeed);

    // SIZE DETERMINATION
    float offspringSize = size;
    // mutations of size
    int mutateSize = Random.Range(0, mutationRate);
    if (mutateSize == 1)
        // then mutate size
        offspringSize = Random.Range(minSize, maxSize);

    // SENSORY DETERMINATION
    float offspringSenseRadius = senseRadius;
    // mutations of sense radius
    int mutateSenseRadius = Random.Range(0, mutationRate);
    if (mutateSenseRadius == 1)
        // then mutate sense radius
        offspringSenseRadius = Random.Range(minSenseRadius, maxSenseRadius);

    // Create Offspring
    GameObject offspring = Instantiate(predatorAsexual, transform.position, Quaternion.identity, parentObjectOfOffspring);
    // give the newly detemrined trait values to the offspring
    offspring.GetComponent<PredatorAsexual>().size = offspringSize;
    offspring.GetComponent<PredatorAsexual>().speed = offspringSpeed;
    offspring.GetComponent<PredatorAsexual>().senseRadius = offspringSenseRadius;
    offspring.GetComponent<PredatorAsexual>().energy = 5000;

    // take away the energy that it takes to reproduce
    energy -= energyForRep;
}

// COLLISION
void OnCollisionEnter2D(Collision2D col)
{
    // Collision with food
    if (food == (food | (1 << col.gameObject.layer)))
```

```
        {
            // check size of predator compared to food size
            // this ensures predators only wat food that is smaller than them
            if (size >= col.gameObject.transform.localScale.x)
            {
                // catch prey or not
                float sizeDifference = (size - col.gameObject.transform.localScale.x);
                float chanceToCatch = Random.Range(0f, sizeDifference);

                // if chanceToCatch is greater than 50% of sizeDifference then eat food
                if (chanceToCatch > sizeDifference / 2)
                {
                    // Add energy
                    energy += energyInFood;
                    // destroy the food
                    Destroy(col.gameObject);
                }
                else // Forced Rest
                {
                    rest = true;
                }
            }
        }

        // Collision with periodic bounds
        // Deals with the layers
        // Does NOT deal with moving of the predator
        if (col.gameObject.CompareTag("Periodic"))
        {
            // Set Predator to a layer that can't be interacted with by periodicBounds layer
            gameObject.layer = 11;
            // Start timer to move back to original layer
            swapSides = true;
        }
    }

    // When Destroy() is called on this object
    private void OnDestroy()
    {
        // Update Statistics
        // Population
        PredatorStatistics.predatorAsexualCount -= 1;
        // Update Trait Stats
        // size
        PredatorStatistics.sizeDivisionTracker[sizeDiv] -= 1;
        // speed
        PredatorStatistics.speedDivisionTracker[speedDiv] -= 1;
        // sense radius
        PredatorStatistics.senseRadiusDivisionTracker[senseRadiusDiv] -= 1;
    }
}
```

Hermaphrodite Predator: (462 total lines, 429 sloc)

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

public class PredatorHermaphrodite : MonoBehaviour
{
    // TRAITS
    [Header("Trait Values")]
    public float speed;
    public float size;
    // tracking of traits
    private PredatorStatistics predatorStatistics;
    // trait divisions
    public int speedDiv;
    public int sizeDiv;
    public int senseRadiusDiv;

    // ENERGY SETTINGS
    [Header("Energy Settings")]
    public float minEnergyForRep;
    public float minEnergyToBeHungry;
    public float energyInFood;
    public float energyForRep;

    // CATCHING PREY
    [Header("Catching Prey")]
```

```
public float setRestTime;
public bool rest;

// RUN TIME EFFECTED VARIABLES
[Header("Simulated Variables")]
public float energy;
public bool readyForRep;
public float timeBtwRep;
private float restTime;
// MOVEMENT
public bool move;
private Vector2 targetPos;
private float timeBtwDecision;
public float setTimeBtwDecision;

// PERIODIC BOUNDS
public float setHitBoundsTime;
private float hitBoundsTime;
public bool swapSides;
public int initialLayer;

// MOVEMENT SETTINGS
[Header("Movement Values")]
public float maxX;
public float maxY;
public float minX;
public float minY;

// STATE
[Header("State Variables")]
public string currentState;

// SENSORY
[Header("Sensory Variables")]
public float senseRadius;
// check for food
public bool foodClose;
public LayerMask food;
private Vector2 foodObjectPos;
// check for mates
public bool matesClose;
public LayerMask mates;
private Vector2 mateObjectPos;

// REPRODUCTION
[Header("Reproduction")]
public GameObject hermaphrodite;
public float setTimeBtwRep;
// parent object holding all spawned offspring
private Transform parentObjectOfOffspring;
// Mutations
[Header("Mutations")]
public int mutationRate;
public float minSize;
public float maxSize;
public float minSpeed;
public float maxSpeed;
public float minSenseRadius;
public float maxSenseRadius;

// Start is called before the first frame update
void Start()
{
    //Initialize values
    // Set size based on the size gene
    transform.localScale = new Vector3(size, size, size);
    readyForRep = false;
    // allow movement
    move = true;
    // reset timer of decisions
    timeBtwDecision = 0;
    // reset swap side bool
    swapSides = false;
    // Set initialLayer
    initialLayer = gameObject.layer;
    // Get statistics reference
    predatorStatistics = FindObjectOfType<PredatorStatistics>();

    // Set timeBtwRep
    timeBtwRep = setTimeBtwRep;
```

```
    // Set restTime
    restTime = setRestTime;
    // Set rest
    rest = false;
    // Update statistics
    // POPULATIONS ones
    PredatorStatistics.predatorHermaphroditeCount += 1;
    PredatorStatistics.allTimePredatorHermaphroditeCount += 1;
    // TRAIT ones
    // Get Divisions
    // size
    sizeDiv = predatorStatistics.getSizeDivision(size);
    // speed
    speedDiv = predatorStatistics.getSpeedDivision(speed);
    // sense radius
    senseRadiusDiv = predatorStatistics.getSenseRadiusDivision(senseRadius);
    // Update Trait Stats
    // size
    CreatureStatistics.sizeDivisionTracker[sizeDiv] += 1;
    // speed
    CreatureStatistics.speedDivisionTracker[speedDiv] += 1;
    // sense radius
    CreatureStatistics.senseRadiusDivisionTracker[senseRadiusDiv] += 1;

    // Set parentObjectOfOffspring to the object holding all PREDATOR creatures
    parentObjectOfOffspring = GameObject.FindGameObjectWithTag("PredatorHermaphroditeHolder").transform;
}

// Update is called once per frame
void Update()
{
    // WANDER STATE
    // this state has the predator wander around the world.
    // In this state the predator can: Move, Rest.
    if (currentState == "Wander")
    {
        // Move or Not
        timeBtwDecision -= Time.deltaTime;
        if (timeBtwDecision <= 0)
        {
            // Reset Timer
            timeBtwDecision = setTimeBtwDecision;
            // Randomly Decide What To Do
            int randint = Random.Range(0, 2);
            if (randint == 0)
            {
                generateRandTargetPos();
                move = true;
            }
            else
                move = false;
        }
    }

    // GETFOOD STATE
    // this state has the predator move to the food piece it sensed.
    // In this state the predator can: Move.
    if (currentState == "GetFood")
    {
        // Set targetPos to the food object.
        targetPos = foodObjectPos;
        // move
        move = true;
    }

    // GETMATE STATE
    // this state has the predator move to a potential mate, and reporoduce
    // In this state the predator can: Move, Reproduce.
    if (currentState == "GetMate")
    {
        // Set targetPos to potential mate
        targetPos = mateObjectPos;
        // move
        move = true;
        // IMPORTANT
        // reproduction only occurs on touch.
        // see OnCollisionEnter2D() for the calling of Reproduce()
    }

    // MOVING / ENERGY CONSUMPTION
```

```
    // When move is true AND rest is false, Move towards the targetPos
    if (move == true && rest == false)
    {
        // Move to targetPos
        transform.position = Vector2.MoveTowards(transform.position, targetPos, speed * Time.deltaTime);
        // This deals with the constant loss of energy from moving
        // size is now factored in because size effects how much energy is needed to move
        energy -= speed / (1f + size) * Time.timeScale;
    }
    else
    {
        // This deals with the constant loss of energy from resting
        energy -= (1 * Time.deltaTime);
    }

    // Can Predator Reproduce?
    timeBtwRep -= Time.deltaTime;
    if (energy >= minEnergyForRep && timeBtwRep <= 0)
        readyForRep = true;
    else
        readyForRep = false;

    // PERIODIC BOUNDS
    // when the swapSides bool is true then start a timer that will wait for the predator to swap sides
    // this allows the predator to never become stuck in an endless state of swapping sides while also allowing normal sensing
    if (swapSides)
    {
        // Inititalize the count down timer
        hitBoundsTime = setHitBoundsTime;
        // Count down
        hitBoundsTime -= Time.deltaTime;
        // If hitBoundsTime hits 0
        if (hitBoundsTime <= 0)
        {
            // Set Predator layer to its original layer
            gameObject.layer = initialLayer;
            // Stop the countdown by setting swapSides to false
            swapSides = false;
        }
    }

    // DEATH FROM LACK OF ENERGY
    // this will run if the energy of the predator ever reaches 0
    if (energy <= 0)
    {
        Destroy(gameObject);
    }

    // FORCED REST
    // this only happens when this predator fails to catch prey (food)
    if (rest == true)
    {
        // start countdown of rest
        restTime -= Time.deltaTime;
        if(restTime <= 0)
        {
            rest = false;
            // reset rest time
            restTime = setRestTime;
        }
    }
}

private void FixedUpdate()
{
    // SENSORY
    // This happens all of the time no matter what, and controls the states.
    // check for food
    foodClose = Physics2D.OverlapCircle(transform.position, senseRadius, food, 0);
    // check for mates
    matesClose = Physics2D.OverlapCircle(transform.position, senseRadius, mates, 0);

    // DECISION MAKING
    // priorities: #1 Keep energy up, #2 Mate
    // If food or mate is close compare the options
    if (foodClose == true || matesClose == true)
    {
        // Check if Mating is a viable option or if energy is needed.
        if (energy < minEnergyForRep || energy < minEnergyToBeHungry || !matesClose)
        {
```

```
            if (foodClose == true)
            {
                GameObject potentialPrey = Physics2D.OverlapCircle(transform.position, senseRadius, food, 0).gameObject;
                // check size of predator compared to food size
                // this ensures predators only eat food that is smaller than them
                // if the sensed prey is smaller then or the same size as the predator (itself) then chase the prey
                if (potentialPrey.transform.localScale.x <= size)
                {
                    foodObjectPos = potentialPrey.transform.position;
                    currentState = "GetFood";
                }
                else // if not then wander
                {
                    currentState = "Wander";
                }
            }
            else // if not then wander
            {
                currentState = "Wander";
            }
        } // If Mating is viable than go to mate.
        else
        {
            if (matesClose == true && readyForRep)
            {
                currentState = "GetMate";
                GameObject potentialMate = Physics2D.OverlapCircle(transform.position, senseRadius, mates, 0).gameObject;
                // If the sensed potential mate can also mate then move to the mate.
                // && potentialMate != gameObject checks if the mate in question is not itself.
                // This is VITAL because Physics2D.OverlapCircle() returns any objects within the specified LayerMask
                // Meaning it could return the object it is being emitted from.
                if (potentialMate.GetComponent<PredatorHermaphrodite>().currentState == "GetMate" && potentialMate != gameObject)
                {
                    mateObjectPos = Physics2D.OverlapCircle(transform.position, senseRadius, mates, 0).transform.position;
                }
            }
            else
            {
                currentState = "Wander";
            }
        }
    } // If not then wander
    else
    {
        currentState = "Wander";
    }
}

// Generate new position
void generateRandTargetPos()
{
    // Generate Random Position
    float randX = Random.Range(minX, maxX);
    float randY = Random.Range(minY, maxY);
    targetPos = new Vector2(randX, randY);
}

// Reproduce
void reproduce(PredatorHermaphrodite mateTraits)
{
    // Determine Speed Of Offspring
    float offspringSpeed = (mateTraits.speed + speed) / 2;
    // determine variation
    float speedVariation = Random.Range((offspringSpeed * -0.2f), (offspringSpeed * 0.2f));
    // apply variation
    offspringSpeed += speedVariation;
    // mutations of speed
    int mutateSpeed = Random.Range(0, mutationRate);
    if (mutateSpeed == 1)
        // then mutate speed
        offspringSpeed = Random.Range(minSpeed, maxSpeed);
    // final check
    // this makes sure the final traits don't go over set maximum or minimum values
    if (offspringSpeed < minSpeed)
        offspringSpeed = minSpeed;
    if (offspringSpeed > maxSpeed)
        offspringSpeed = maxSpeed;

    // Determine Size Of Offspring
    float offspringSize = (mateTraits.size + size) / 2;
```

```
    // determine variation
    float sizeVariation = Random.Range((offspringSize * -0.2f), (offspringSize * 0.2f));
    // apply variation
    offspringSize += sizeVariation;
    // mutations of size
    int mutateSize = Random.Range(0, mutationRate);
    if (mutateSize == 1)
        // then mutate size
        offspringSize = Random.Range(minSize, maxSize);
    // final check
    // this makes sure the final traits don't go over set maximum or minimum values
    if (offspringSize < minSize)
        offspringSize = minSize;
    if (offspringSize > maxSize)
        offspringSize = maxSize;

    // Determine Sense Radius Of Offspring
    float offspringSenseRadius = (mateTraits.senseRadius + senseRadius) / 2;
    // determine variation
    float senseRadiusVariation = Random.Range((offspringSenseRadius * -0.2f), (offspringSenseRadius * 0.2f));
    // apply variation
    offspringSenseRadius += senseRadiusVariation;
    // mutations of sense radius
    int mutateSenseRadius = Random.Range(0, mutationRate);
    if (mutateSenseRadius == 1)
        // then mutate sense radius
        offspringSenseRadius = Random.Range(minSenseRadius, maxSenseRadius);
    // final check
    // this makes sure the final traits don't go over set maximum or minimum values
    if (offspringSenseRadius < minSenseRadius)
        offspringSenseRadius = minSenseRadius;
    if (offspringSenseRadius > maxSenseRadius)
        offspringSenseRadius = maxSenseRadius;

    // SPAWNING OF NEW CREATURE (offSpring)
    GameObject offspring;
    offspring = Instantiate(hermaphrodite, transform.position, Quaternion.identity, parentObjectOfOffspring);
    // give the newly detemrined trait values to the offspring
    offspring.GetComponent<PredatorHermaphrodite>().size = offspringSize;
    offspring.GetComponent<PredatorHermaphrodite>().speed = offspringSpeed;
    offspring.GetComponent<PredatorHermaphrodite>().senseRadius = offspringSenseRadius;
    offspring.GetComponent<PredatorHermaphrodite>().energy = 5000;

    // take away the energy that it takes to reproduce
    energy -= energyForRep;
    // esure the state doesn't become stuck in GetMate
    currentState = "Wander";
}

// COLLISION
void OnCollisionEnter2D(Collision2D col)
{
    // If Predator hits another alike Predator and Is in the GetMate State
    if (col.gameObject.CompareTag("PredatorHermaphrodite") && currentState == "GetMate")
    {
        // Can Predator Reproduce?
        if (readyForRep == true)
        {
            reproduce(col.gameObject.GetComponent<PredatorHermaphrodite>());
            timeBtwRep = setTimeBtwRep;
        }
    }

    // Collision with food
    if (food == (food | (1 << col.gameObject.layer)))
    {
        // check size of predator compared to food size
        // this ensures predators only wat food that is smaller than them
        if (size >= col.gameObject.transform.localScale.x)
        {
            // catch prey or not
            float sizeDifference = (size - col.gameObject.transform.localScale.x);
            float chanceToCatch = Random.Range(0f, sizeDifference);

            // if chanceToCatch is greater than 50% of sizeDifference then eat food
            if (chanceToCatch > sizeDifference / 2)
            {
                // Add energy
                energy += energyInFood;
                // destroy the food
```

```
                Destroy(col.gameObject);
            }
            else // Forced Rest
            {
                rest = true;
            }
        }
    }

    // Collision with periodic bounds
    // Deals with the layers
    // Does NOT deal with moving of the predator
    if (col.gameObject.CompareTag("Periodic"))
    {
        // Set Predator to a layer that can't be interacted with by periodicBounds layer
        gameObject.layer = 11;
        // Start timer to move back to original layer
        swapSides = true;
    }
}

// When Destroy() is called on this object
private void OnDestroy()
{
    // Update Statistics
    // Population
    PredatorStatistics.predatorHermaphroditeCount -= 1;
    // Update Trait Stats
    // size
    PredatorStatistics.sizeDivisionTracker[sizeDiv] -= 1;
    // speed
    PredatorStatistics.speedDivisionTracker[speedDiv] -= 1;
    // sense radius
    PredatorStatistics.senseRadiusDivisionTracker[senseRadiusDiv] -= 1;
}
}
```

Sexual Male Predator: (385 total lines, 358 sloc)

```
using UnityEngine;

public class PredatorSexualMale : MonoBehaviour
{
    // TRAITS
    [Header("Trait Values")]
    public float speed;
    public float size;
    public float senseRadius;
    public string[] chromosomes;
    // tracking of traits
    private PredatorStatistics predatorStatistics;
    // trait divisions
    public int speedDiv;
    public int sizeDiv;
    public int senseRadiusDiv;

    // ENERGY SETTINGS
    [Header("Energy Settings")]
    public float minEnergyForRep;
    public float minEnergyToBeHungry;
    public float energyInFood;
    public float energyToRep;

    // CATCHING PREY
    [Header("Catching Prey")]
    public float setRestTime;
    public bool rest;

    // RUN TIME EFFECTED VARIABLES
    [Header("Simulated Variables")]
    public float energy;
    public bool readyForRep;
    private float timeBtwRep;
    private float restTime;
    // MOVEMENT
    public bool move;
    private Vector2 targetPos;
    private float timeBtwDecision;
    public float setTimeBtwDecision;
```

```
// PERIODIC BOUNDS
public float setHitBoundsTime;
private float hitBoundsTime;
public bool swapSides;
public int initialLayer;

// MOVEMENT SETTINGS
[Header("Movement Values")]
public float maxX;
public float maxY;
public float minX;
public float minY;

// STATE
[Header("State Variables")]
public string currentState;

// SENSORY
[Header("Sensory Variables")]
// check for food
public bool foodClose;
public LayerMask food;
private Vector2 foodObjectPos;
// check for mates
public bool matesClose;
public LayerMask mates;
private Vector2 mateObjectPos;

// REPRODUCTION
public float setTimeBtwRep;

// Start is called before the first frame update
void Start()
{
    //Initialize values
    // Set size based on the size gene
    transform.localScale = new Vector3(size, size, size);
    readyForRep = false;
    // allow movement
    move = true;
    // reset timer of decisions
    timeBtwDecision = 0;
    // reset swap side bool
    swapSides = false;
    // Set initialLayer
    initialLayer = gameObject.layer;
    // Get statistics reference
    predatorStatistics = FindObjectOfType<PredatorStatistics>();

    // Set chromosomes
    chromosomes = new string[2];
    chromosomes[0] = "X";
    chromosomes[1] = "Y";
    // Set timeBtwRep
    timeBtwRep = setTimeBtwRep;
    // Set restTime
    restTime = setRestTime;
    // Set rest
    rest = false;
    // Update statistics
    // POPULATIONS ones
    PredatorStatistics.predatorSexualCount += 1;
    PredatorStatistics.allTimePredatorSexualCount += 1;
    // TRAIT ones
    // Get Divisions
    // size
    sizeDiv = predatorStatistics.getSizeDivision(size);
    // speed
    speedDiv = predatorStatistics.getSpeedDivision(speed);
    // sense radius
    senseRadiusDiv = predatorStatistics.getSenseRadiusDivision(senseRadius);
    // Update Trait Stats
    // size
    CreatureStatistics.sizeDivisionTracker[sizeDiv] += 1;
    // speed
    CreatureStatistics.speedDivisionTracker[speedDiv] += 1;
    // sense radius
    CreatureStatistics.senseRadiusDivisionTracker[senseRadiusDiv] += 1;
}
```

```
// Update is called once per frame
void Update()
{
    // WANDER STATE
    // this state has the predator wander around the world.
    // In this state the predator can: Move, Rest.
    if (currentState == "Wander")
    {
        // Move or Not
        timeBtwDecision -= Time.deltaTime;
        if (timeBtwDecision <= 0)
        {
            // Reset Timer
            timeBtwDecision = setTimeBtwDecision;
            // Randomly Decide What To Do
            int randint = Random.Range(0, 2);
            if (randint == 0)
            {
                generateRandTargetPos();
                move = true;
            }
            else
                move = false;
        }
    }

    // GETFOOD STATE
    // this state has the predator move to the food piece it sensed.
    // In this state the predator can: Move.
    if (currentState == "GetFood")
    {
        // Set targetPos to the food object.
        targetPos = foodObjectPos;
        // move
        move = true;
    }

    // GETMATE STATE
    // this state has the predator move to a potential mate, and reporoduce
    // In this state the predator can: Move, Reproduce.
    if (currentState == "GetMate")
    {
        // Set targetPos to potential mate
        targetPos = mateObjectPos;
        // move
        move = true;
        // IMPORTANT
        // reproduction only occurs on touch.
        // see OnCollisionEnter2D() for the calling of Reproduce()
    }

    // MOVING / ENERGY CONSUMPTION
    // When move is true AND rest is false, Move towards the targetPos
    if (move == true && rest == false)
    {
        // Move to targetPos
        transform.position = Vector2.MoveTowards(transform.position, targetPos, speed * Time.deltaTime);
        // This deals with the constant loss of energy from moving
        // size is now factored in because size effects how much energy is needed to move
        energy -= speed / (1f + size) * Time.timeScale;
    }
    else
    {
        // This deals with the constant loss of energy from resting
        energy -= (1 * Time.deltaTime);
    }

    // Can Predator Reproduce?
    timeBtwRep -= Time.deltaTime;
    if (energy >= minEnergyForRep && timeBtwRep <= 0)
        readyForRep = true;
    else
        readyForRep = false;

    // PERIODIC BOUNDS
    // when the swapSides bool is true then start a timer that will wait for the predator to swap sides
    // this allows the predator to never become stuck in an endless state of swapping sides while also allowing normal sensing
    if (swapSides)
    {
        // Inititalize the count down timer
```

```
            hitBoundsTime = setHitBoundsTime;
            // Count down
            hitBoundsTime -= Time.deltaTime;
            // If hitBoundsTime hits 0
            if (hitBoundsTime <= 0)
            {
                // Set Predator layer to its original layer
                gameObject.layer = initialLayer;
                // Stop the countdown by setting swapSides to false
                swapSides = false;
            }
        }

        // DEATH FROM LACK OF ENERGY
        // this will run if the energy of the predator ever reaches 0
        if (energy <= 0)
        {
            Destroy(gameObject);
        }

        // FORCED REST
        // this only happens when this predator fails to catch prey (food)
        if (rest == true)
        {
            // start countdown of rest
            restTime -= Time.deltaTime;
            if (restTime <= 0)
            {
                rest = false;
                // reset rest time
                restTime = setRestTime;
            }
        }
    }

    private void FixedUpdate()
    {
        // SENSORY
        // This happens all of the time no matter what, and controls the states.
        // check for food
        foodClose = Physics2D.OverlapCircle(transform.position, senseRadius, food, 0);
        // check for mates
        matesClose = Physics2D.OverlapCircle(transform.position, senseRadius, mates, 0);

        // DECISION MAKING
        // priorities: #1 Keep energy up, #2 Mate
        // If food or mate is close compare the options
        if (foodClose == true || matesClose == true)
        {
            // Check if Mating is a viable option or if energy is needed.
            if (energy < minEnergyForRep || energy < minEnergyToBeHungry || !matesClose)
            {
                if (foodClose == true)
                {
                    if (foodClose == true)
                    {
                        GameObject potentialPrey = Physics2D.OverlapCircle(transform.position, senseRadius, food, 0).gameObject;
                        // check size of predator compared to food size
                        // this ensures predators only eat food that is smaller than them
                        // if the sensed prey is smaller then or the same size as the predator (itself) then chase the prey
                        if (potentialPrey.transform.localScale.x <= size)
                        {
                            foodObjectPos = potentialPrey.transform.position;
                            currentState = "GetFood";
                        }
                        else // if not then wander
                        {
                            currentState = "Wander";
                        }
                    }
                }
                else
                {
                    currentState = "Wander";
                }
            } // If Mating is viable than go to mate.
            else
            {
                if (matesClose == true && readyForRep)
                {
```

```
                currentState = "GetMate";
                GameObject potentialMate = Physics2D.OverlapCircle(transform.position, senseRadius, mates, 0).gameObject;
                // If the sensed potential mate can also mate then move to the mate.
                if (potentialMate.GetComponent<PredatorSexualFemale>().currentState == "GetMate")
                {
                    mateObjectPos = Physics2D.OverlapCircle(transform.position, senseRadius, mates, 0).transform.position;
                }
            }
            else
            {
                currentState = "Wander";
            }
        }
    } // If not then wander
    else
    {
        currentState = "Wander";
    }
}

// Generate new position
void generateRandTargetPos()
{
    // Generate Random Position
    float randX = Random.Range(minX, maxX);
    float randY = Random.Range(minY, maxY);
    targetPos = new Vector2(randX, randY);
}

// Reproduce
void reproduce()
{
    energy -= energyToRep;
    currentState = "Wander";
    // IMPORTANT
    // actual sharing of traits, and instantiating of new predators is done in the female predator's reproduce() function.
}

// COLLISION
void OnCollisionEnter2D(Collision2D col)
{
    // If Predator hits another alike Predator and Is in the GetMate State
    if (col.gameObject.CompareTag("PredatorFemale") && currentState == "GetMate")
    {
        // Can Predator Reproduce?
        if (readyForRep == true)
        {
            reproduce();
            timeBtwRep = setTimeBtwRep;
        }
    }

    // Collision with food
    if (food == (food | (1 << col.gameObject.layer)))
    {
        // check size of predator compared to food size
        // this ensures predators only wat food that is smaller than them
        if (size >= col.gameObject.transform.localScale.x)
        {
            // catch prey or not
            float sizeDifference = (size -= col.gameObject.transform.localScale.x);
            float chanceToCatch = Random.Range(0f, sizeDifference);

            // if chanceToCatch is greater than 50% of sizeDifference then eat food
            if (chanceToCatch > sizeDifference/2)
            {
                // Add energy
                energy += energyInFood;
                // destroy the food
                Destroy(col.gameObject);
            }
            else // Forced Rest
            {
                rest = true;
            }
        }
    }

    // Collision with periodic bounds
    // Deals with the layers
```

```
        // Does NOT deal with moving of the predator
        if (col.gameObject.CompareTag("Periodic"))
        {
            // Set Predator to a layer that can't be interacted with by periodicBounds layer
            gameObject.layer = 11;
            // Start timer to move back to original layer
            swapSides = true;
        }
    }

    // When Destroy() is called on this object
    private void OnDestroy()
    {
        // Update Statistics
        // Population
        PredatorStatistics.predatorSexualCount -= 1;
        // Update Trait Stats
        // size
        PredatorStatistics.sizeDivisionTracker[sizeDiv] -= 1;
        // speed
        PredatorStatistics.speedDivisionTracker[speedDiv] -= 1;
        // sense radius
        PredatorStatistics.senseRadiusDivisionTracker[senseRadiusDiv] -= 1;
    }
}
```

Sexual Female Predator: (493 total lines, 459 sloc)

```
using UnityEngine;

public class PredatorSexualFemale : MonoBehaviour
{
    // TRAITS
    [Header("Trait Values")]
    public float speed;
    public float size;
    public float senseRadius;
    public string[] chromosomes;
    // tracking of traits
    private PredatorStatistics predatorStatistics;
    // trait divisions
    public int speedDiv;
    public int sizeDiv;
    public int senseRadiusDiv;

    // ENERGY SETTINGS
    [Header("Energy Settings")]
    public float minEnergyForRep;
    public float minEnergyToBeHungry;
    public float energyInFood;
    public float energyForRep;

    // CATCHING PREY
    [Header("Catching Prey")]
    public float setRestTime;
    public bool rest;

    // RUN TIME EFFECTED VARIABLES
    [Header("Simulated Variables")]
    public float energy;
    public bool readyForRep;
    private float timeBtwRep;
    private float restTime;
    // MOVEMENT
    public bool move;
    private Vector2 targetPos;
    private float timeBtwDecision;
    public float setTimeBtwDecision;

    // PERIODIC BOUNDS
    public float setHitBoundsTime;
    private float hitBoundsTime;
    public bool swapSides;
    public int initialLayer;

    // MOVEMENT SETTINGS
    [Header("Movement Values")]
    public float maxX;
    public float maxY;
    public float minX;
```

```
public float minY;

// STATE
[Header("State Variables")]
public string currentState;

// SENSORY
[Header("Sensory Variables")]
// check for food
public bool foodClose;
public LayerMask food;
private Vector2 foodObjectPos;
// check for mates
public bool matesClose;
public LayerMask mates;
private Vector2 mateObjectPos;

// REPRODUCTION
[Header("Reproduction")]
public GameObject SexualMale;
public GameObject SexualFemale;
public float setTimeBtwRep;
// parent object holding all spawned offspring
private Transform parentObjectOfOffspring;
// Mutations
[Header("Mutations")]
public int mutationRate;
public float minSize;
public float maxSize;
public float minSpeed;
public float maxSpeed;
public float minSenseRadius;
public float maxSenseRadius;

// Start is called before the first frame update
void Start()
{
    //Initialize values
    // Set size based on the size gene
    transform.localScale = new Vector3(size, size, size);
    readyForRep = false;
    // allow movement
    move = true;
    // reset timer of decisions
    timeBtwDecision = 0;
    // reset swap side bool
    swapSides = false;
    // Set initialLayer
    initialLayer = gameObject.layer;
    // Get statistics reference
    predatorStatistics = FindObjectOfType<PredatorStatistics>();

    // Set chromosomes
    chromosomes = new string[2];
    chromosomes[0] = "X";
    chromosomes[1] = "X";
    // Set timeBtwRep
    timeBtwRep = setTimeBtwRep;
    // Set restTime
    restTime = setRestTime;
    // Set rest
    rest = false;
    // Update statistics
    // POPULATIONS ones
    PredatorStatistics.predatorSexualCount += 1;
    PredatorStatistics.allTimePredatorSexualCount += 1;
    // TRAIT ones
    // Get Divisions
    // size
    sizeDiv = predatorStatistics.getSizeDivision(size);
    // speed
    speedDiv = predatorStatistics.getSpeedDivision(speed);
    // sense radius
    senseRadiusDiv = predatorStatistics.getSenseRadiusDivision(senseRadius);
    // Update Trait Stats
    // size
    CreatureStatistics.sizeDivisionTracker[sizeDiv] += 1;
    // speed
    CreatureStatistics.speedDivisionTracker[speedDiv] += 1;
    // sense radius
```

```
    CreatureStatistics.senseRadiusDivisionTracker[senseRadiusDiv] += 1;

    // Set parentObjectOfOffspring to the object holding all PREDATOR creatures
    parentObjectOfOffspring = GameObject.FindGameObjectWithTag("PredatorHolder").transform;
}

// Update is called once per frame
void Update()
{
    // WANDER STATE
    // this state has the predator wander around the world.
    // In this state the predator can: Move, Rest.
    if (currentState == "Wander")
    {
        // Move or Not
        timeBtwDecision -= Time.deltaTime;
        if (timeBtwDecision <= 0)
        {
            // Reset Timer
            timeBtwDecision = setTimeBtwDecision;
            // Randomly Decide What To Do
            int randint = Random.Range(0, 2);
            if (randint == 0)
            {
                generateRandTargetPos();
                move = true;
            }
            else
                move = false;
        }
    }

    // GETFOOD STATE
    // this state has the predator move to the food piece it sensed.
    // In this state the predator can: Move.
    if (currentState == "GetFood")
    {
        // Set targetPos to the food object.
        targetPos = foodObjectPos;
        // move
        move = true;
    }

    // GETMATE STATE
    // this state has the predator move to a potential mate, and reporoduce
    // In this state the predator can: Move, Reproduce.
    if (currentState == "GetMate")
    {
        // Set targetPos to potential mate
        targetPos = mateObjectPos;
        // move
        move = true;
        // IMPORTANT
        // reproduction only occurs on touch.
        // see OnCollisionEnter2D() for the calling of Reproduce()
    }

    // MOVING / ENERGY CONSUMPTION
    // When move is true AND rest is false, Move towards the targetPos
    if (move == true && rest == false)
    {
        // Move to targetPos
        transform.position = Vector2.MoveTowards(transform.position, targetPos, speed * Time.deltaTime);
        // This deals with the constant loss of energy from moving
        // size is now factored in because size effects how much energy is needed to move
        energy -= speed / (1f + size) * Time.timeScale;
    }
    else
    {
        // This deals with the constant loss of energy from resting
        energy -= (1 * Time.deltaTime);
    }

    // Can Predator Reproduce?
    timeBtwRep -= Time.deltaTime;
    if (energy >= minEnergyForRep && timeBtwRep <= 0)
        readyForRep = true;
    else
        readyForRep = false;
```

```
    // PERIODIC BOUNDS
    // when the swapSides bool is true then start a timer that will wait for the predator to swap sides
    // this allows the predator to never become stuck in an endless state of swapping sides while also allowing normal sensing
    if (swapSides)
    {
        // Inititalize the count down timer
        hitBoundsTime = setHitBoundsTime;
        // Count down
        hitBoundsTime -= Time.deltaTime;
        // If hitBoundsTime hits 0
        if (hitBoundsTime <= 0)
        {
            // Set Predator layer to its original layer
            gameObject.layer = initialLayer;
            // Stop the countdown by setting swapSides to false
            swapSides = false;
        }
    }

    // DEATH FROM LACK OF ENERGY
    // this will run if the energy of the predator ever reaches 0
    if (energy <= 0)
    {
        Destroy(gameObject);
    }

    // FORCED REST
    // this only happens when this predator fails to catch prey (food)
    if (rest == true)
    {
        // start countdown of rest
        restTime -= Time.deltaTime;
        if (restTime <= 0)
        {
            rest = false;
            // reset rest time
            restTime = setRestTime;
        }
    }
}

private void FixedUpdate()
{
    // SENSORY
    // This happens all of the time no matter what, and controls the states.
    // check for food
    foodClose = Physics2D.OverlapCircle(transform.position, senseRadius, food, 0);
    // check for mates
    matesClose = Physics2D.OverlapCircle(transform.position, senseRadius, mates, 0);

    // DECISION MAKING
    // priorities: #1 Keep energy up, #2 Mate
    // If food or mate is close compare the options
    if (foodClose == true || matesClose == true)
    {
        // Check if Mating is a viable option or if energy is needed.
        if (energy < minEnergyForRep || energy < minEnergyToBeHungry || !matesClose)
        {
            if (foodClose == true)
            {
                if (foodClose == true)
                {
                    GameObject potentialPrey = Physics2D.OverlapCircle(transform.position, senseRadius, food, 0).gameObject;
                    // check size of predator compared to food size
                    // this ensures predators only eat food that is smaller than them
                    // if the sensed prey is smaller then or the same size as the predator (itself) then chase the prey
                    if (potentialPrey.transform.localScale.x <= size)
                    {
                        foodObjectPos = potentialPrey.transform.position;
                        currentState = "GetFood";
                    }
                    else // if not then wander
                    {
                        currentState = "Wander";
                    }
                }
            }
            else
            {
                currentState = "Wander";
```

```
            }
        } // If Mating is viable than go to mate.
        else
        {
            if (matesClose == true && readyForRep)
            {
                currentState = "GetMate";
                GameObject potentialMate = Physics2D.OverlapCircle(transform.position, senseRadius, mates, 0).gameObject;
                // If the sensed potential mate can also mate then move to the mate.
                if (potentialMate.GetComponent<PredatorSexualMale>().currentState == "GetMate")
                {
                    mateObjectPos = Physics2D.OverlapCircle(transform.position, senseRadius, mates, 0).transform.position;
                }
            }
            else
            {
                currentState = "Wander";
            }
        }
    } // If not then wander
    else
    {
        currentState = "Wander";
    }
}

// Generate new position
void generateRandTargetPos()
{
    // Generate Random Position
    float randX = Random.Range(minX, maxX);
    float randY = Random.Range(minY, maxY);
    targetPos = new Vector2(randX, randY);
}

// Reproduce
void reproduce(PredatorSexualMale mateTraits)
{
    // Decide Sex Of Offspring
    int randChromosome = Random.Range(0, 2);
    string donatedChromosome = mateTraits.chromosomes[randChromosome];

    // FINAL SEX OF OFFSPRING
    string[] offspringChromosomes = new string[2];
    offspringChromosomes[0] = "X";
    offspringChromosomes[1] = donatedChromosome;

    // Determine Speed Of Offspring
    float offspringSpeed = (mateTraits.speed + speed) / 2;
    // determine variation
    float speedVariation = Random.Range((offspringSpeed * -0.2f), (offspringSpeed * 0.2f));
    // apply variation
    offspringSpeed += speedVariation;
    // mutations of speed
    int mutateSpeed = Random.Range(0, mutationRate);
    if (mutateSpeed == 1)
        // then mutate speed
        offspringSpeed = Random.Range(minSpeed, maxSpeed);
    // final check
    // this makes sure the final traits don't go over set maximum or minimum values
    if (offspringSpeed < minSpeed)
        offspringSpeed = minSpeed;
    if (offspringSpeed > maxSpeed)
        offspringSpeed = maxSpeed;

    // Determine Size Of Offspring
    float offspringSize = (mateTraits.size + size) / 2;
    // determine variation
    float sizeVariation = Random.Range((offspringSize * -0.2f), (offspringSize * 0.2f));
    // apply variation
    offspringSize += sizeVariation;
    // mutations of size
    int mutateSize = Random.Range(0, mutationRate);
    if (mutateSize == 1)
        // then mutate size
        offspringSize = Random.Range(minSize, maxSize);
    // final check
    // this makes sure the final traits don't go over set maximum or minimum values
    if (offspringSize < minSize)
        offspringSize = minSize;
```

```
    if (offspringSize > maxSize)
        offspringSize = maxSize;

    // Determine Sense Radius Of Offspring
    float offspringSenseRadius = (mateTraits.senseRadius + senseRadius) / 2;
    // determine variation
    float senseRadiusVariation = Random.Range((offspringSenseRadius * -0.2f), (offspringSenseRadius * 0.2f));
    // apply variation
    offspringSenseRadius += senseRadiusVariation;
    // mutations of sense radius
    int mutateSenseRadius = Random.Range(0, mutationRate);
    if (mutateSenseRadius == 1)
        // then mutate sense radius
        offspringSenseRadius = Random.Range(minSenseRadius, maxSenseRadius);
    // final check
    // this makes sure the final traits don't go over set maximum or minimum values
    if (offspringSenseRadius < minSenseRadius)
        offspringSenseRadius = minSenseRadius;
    if (offspringSenseRadius > maxSenseRadius)
        offspringSenseRadius = maxSenseRadius;

    // SPAWNING OF NEW CREATURE (offSpring)
    GameObject offspring;
    if (offspringChromosomes[1] == "Y")
    {
        // the offspring should be a MALE
        offspring = Instantiate(SexualMale, transform.position, Quaternion.identity, parentObjectOfOffspring);
        // give the newly detemrined trait values to the offspring
        offspring.GetComponent<PredatorSexualMale>().size = offspringSize;
        offspring.GetComponent<PredatorSexualMale>().speed = offspringSpeed;
        offspring.GetComponent<PredatorSexualMale>().senseRadius = offspringSenseRadius;
        offspring.GetComponent<PredatorSexualMale>().energy = 5000;
    }
    else if (offspringChromosomes[1] == "X")
    {
        // the offspring should be a FEMALE
        offspring = Instantiate(SexualFemale, transform.position, Quaternion.identity, parentObjectOfOffspring);
        // give the newly detemrined trait values to the offspring
        offspring.GetComponent<PredatorSexualFemale>().size = offspringSize;
        offspring.GetComponent<PredatorSexualFemale>().speed = offspringSpeed;
        offspring.GetComponent<PredatorSexualFemale>().senseRadius = offspringSenseRadius;
        offspring.GetComponent<PredatorSexualFemale>().energy = 5000;
    }
    // take away the energy that it takes to reproduce
    energy -= energyForRep;
    // esure the state doesn't become stuck in GetMate
    currentState = "Wander";
}

// COLLISION
void OnCollisionEnter2D(Collision2D col)
{
    // If Predator hits another alike Predator and Is in the GetMate State
    if (col.gameObject.CompareTag("PredatorMale") && currentState == "GetMate")
    {
        // Can Predator Reproduce?
        if (readyForRep == true)
        {
            reproduce(col.gameObject.GetComponent<PredatorSexualMale>());
            timeBtwRep = setTimeBtwRep;
        }
    }

    // Collision with food
    if (food == (food | (1 << col.gameObject.layer)))
    {
        // check size of predator compared to food size
        // this ensures predators only wat food that is smaller than them
        if (size >= col.gameObject.transform.localScale.x)
        {
            // check size of predator compared to food size
            // this ensures predators only wat food that is smaller than them
            if (size >= col.gameObject.transform.localScale.x)
            {
                // catch prey or not
                float sizeDifference = (size - col.gameObject.transform.localScale.x);
                float chanceToCatch = Random.Range(0f, sizeDifference);

                // if chanceToCatch is greater than 50% of sizeDifference then eat food
                if (chanceToCatch > sizeDifference / 2)
```

```
                {
                    // Add energy
                    energy += energyInFood;
                    // destroy the food
                    Destroy(col.gameObject);
                }
                else // Forced Rest
                {
                    rest = true;
                }
            }
        }
    }

    // Collision with periodic bounds
    // Deals with the layers
    // Does NOT deal with moving of the predator
    if (col.gameObject.CompareTag("Periodic"))
    {
        // Set Predator to a layer that can't be interacted with by periodicBounds layer
        gameObject.layer = 11;
        // Start timer to move back to original layer
        swapSides = true;
    }
}

// When Destroy() is called on this object
private void OnDestroy()
{
    // Update Statistics
    // Population
    PredatorStatistics.predatorSexualCount -= 1;
    // Update Trait Stats
    // size
    PredatorStatistics.sizeDivisionTracker[sizeDiv] -= 1;
    // speed
    PredatorStatistics.speedDivisionTracker[speedDiv] -= 1;
    // sense radius
    PredatorStatistics.senseRadiusDivisionTracker[senseRadiusDiv] -= 1;
}
}
```

Prey Statistics: (264 total lines, 251 sloc)

```
using UnityEngine;
using UnityEngine.UI;
using System;

public class CreatureStatistics : MonoBehaviour
{
    // STATIC VARIABLES FOR TRACKING
    // POPULATIONS
    // Seuxal Creature Counts
    public static int femaleSexualCreatureCount;
    public static int maleSexualCreatureCount;
    public static int sexualCreatureCount;
    public static int allTimeFemaleCreatureCount;
    public static int allTimeMaleCreatureCount;
    public static int allTimeSexualCreatureCount;
    // Aseuxal Creature Counts
    public static int asexualCreatureCount;
    public static int allTimeAsexualCreatureCount;
    // Hermaphrodite Creature Counts
    public static int hermaphroditeCreatureCount;
    public static int allTimeHermaphroditeCreatureCount;
    // TRAITS
    [Header("Trait Tracking")]
    // Size
    public float[] sizeDivisions;
    public static int[] sizeDivisionTracker;
    // Speed
    public float[] speedDivisions;
    public static int[] speedDivisionTracker;
    // Sense Radius
    public float[] senseRadiusDivisions;
    public static int[] senseRadiusDivisionTracker;

    // TEXT OBJECTS FOR DISPLAYING STATISTICS
    // Sexual
    [Header("Sexual Prey Stat Text")]
```

```
    public Text sexualCreatureCountText;
    public Text maleSexualCreatureCountText;
    public Text femaleSexualCreatureCountText;
    public Text allTimeSexualCreatureCountText;
    public Text allTimeMaleSexualCreatureCountText;
    public Text allTimeFemaleSexualCreatureCountText;
    // Asexual
    [Header("Asexual Prey Stat Text")]
    public Text asexualCreatureCountText;
    public Text allTimeAsexualCreatureCountText;
    // Hermaphrodite
    [Header("Hermaphrodite Prey Stat Text")]
    public Text hermaphroditeCreatureCountText;
    public Text allTimeHermaphroditeCreatureCountText;

    // COLLECTING DATA
    [Header("Collecting Data")]
    public float setRecordInterval;
    // run time effected countdown
    private float recordInterval;
    // reference to the SaveData script allowing for writing to logs
    public SaveData saveData;
    // reference to the Timer script allowing for access to it's values
    public Timer timer;
    [Header("Formatting Data")]
    // this stuff is for formatting of data in the file "Log.txt"
    public string[] activeCreatureName;
    public string[] activePredatorName;

    private void Start()
    {
        // Make the simlation run at a stable frame rate
        Application.targetFrameRate = 60;
        // RESET ALL VALUES
        // this ensures that when the functions used to "reload" the scene static statistic values start at 0
        // Reset all time stats
        allTimeAsexualCreatureCount = 0;
        allTimeSexualCreatureCount = 0;
        allTimeFemaleCreatureCount = 0;
        allTimeMaleCreatureCount = 0;
        allTimeHermaphroditeCreatureCount = 0;
        // Reset current stats
        asexualCreatureCount = 0;
        sexualCreatureCount = 0;
        maleSexualCreatureCount = 0;
        femaleSexualCreatureCount = 0;
        hermaphroditeCreatureCount = 0;

        // Initialize trait division tracker arrays
        // Size
        sizeDivisionTracker = new int[5];
        // Speed
        speedDivisionTracker = new int[5];
        // Sense Radius
        senseRadiusDivisionTracker = new int[5];
    }

    // INITIALIZE / FORMAT NEW SET OF DATA
    // only called in the RunSimulation() function of MenuManager
    // this ensures that all values, and settings chosen by the user have been set.
    public void initializeNewLog()
    {
        // VERY IMPORTANT
        // create the new folder, and sub folder for this simulation's logs
        saveData.CreateNewLogFolder(MenuManager.logFolderName);
        // Set recordingInterval
        setRecordInterval = MenuManager.setDataRecordingInterval;
        // Create a description of the circumstances on the log
        saveData.CreatePreyPopulationLog("\n\n New Data Set \n" + activeCreatureName[MenuManager.activeCreatureIndex] + " creatures with " +
activePredatorName[MenuManager.activePredatorIndex] + " predators\n");
        // Determine and format time limit into a string
        TimeSpan timeSpan = TimeSpan.FromSeconds(0 + timer.timeLimits[timer.activeTimeLimitIndex]);
        string timeLimit = string.Format("{0:D2}:{1:D2}:{2:D2}", timeSpan.Hours, timeSpan.Minutes, timeSpan.Seconds);
        // Add all presets, and inital values to the log
        saveData.CreatePreyPopulationLog("Time Limit: " + timeLimit + " Inital Creature Count: " + MenuManager.initialCreatureCount.ToString() + "
Initial Food Count: " + MenuManager.initalFood + " Food Spawn Rate: " + MenuManager.foodSpawnRate + "\n");
    }

    // Update is called once per frame
    void Update()
```

```
        {
            // Update relevant statistics of PREY
            if (MenuManager.activeCreatures[0])
                SexualCreatureTextUpdate();
            if (MenuManager.activeCreatures[1])
                AsexualCreatureTextUpdate();
            if (MenuManager.activeCreatures[2])
                HermaphroditeCreatureTextUpdate();

            // DATA COLLECTION
            // countdown from the set recording interval
            recordInterval -= Time.deltaTime;
            // once the countdown reaches zero record the proper data, and reset the timer
            if (recordInterval <= 0)
            {
                // restart countdown
                recordInterval = setRecordInterval;
                // Log relevant statistics
                if (MenuManager.activeCreatures[0])
                    LogSexual();
                if (MenuManager.activeCreatures[1])
                    LogAsexual();
                if (MenuManager.activeCreatures[2])
                    LogHermaphrodite();
                // Log trait stats
                LogSize();
                LogSpeed();
                LogSenseRadius();
            }
        }

        // LOGGING FUNCTIONS
        // POPULATION STATS
        // Asexual
        void LogAsexual()
        {
            saveData.CreatePreyPopulationLog("Time: " + Mathf.FloorToInt(Timer.time).ToString() + " Creatures: " + asexualCreatureCount.ToString() + " All
Time Creatures: " + allTimeAsexualCreatureCount.ToString());
        }
        // Sexual
        void LogSexual()
        {
            saveData.CreatePreyPopulationLog("Time: " + Mathf.FloorToInt(Timer.time).ToString() + " Creatures: " + sexualCreatureCount.ToString() + " All
Time Creatures: " + allTimeSexualCreatureCount.ToString());
        }
        // Hermaphrodite
        void LogHermaphrodite()
        {
            saveData.CreatePreyPopulationLog("Time: " + Mathf.FloorToInt(Timer.time).ToString() + " Creatures: " + hermaphroditeCreatureCount.ToString()
+ " All Time Creatures: " + allTimeHermaphroditeCreatureCount.ToString());
        }
        // TRAIT STATS
        // Size
        void LogSize()
        {
            saveData.CreatePreySizeLog("Size: Time: " + Mathf.FloorToInt(Timer.time).ToString() + " Division 1: " + sizeDivisionTracker[0].ToString() + "
Division 2: " + sizeDivisionTracker[1].ToString() + " Division 3: " + sizeDivisionTracker[2].ToString() + " Division 4: " + sizeDivisionTracker[3].ToString() + "
Division 5: " + sizeDivisionTracker[4].ToString());
        }
        // Speed
        void LogSpeed()
        {
            saveData.CreatePreySpeedLog("Speed: Time: " + Mathf.FloorToInt(Timer.time).ToString() + " Division 1: " + speedDivisionTracker[0].ToString()
+ " Division 2: " + speedDivisionTracker[1].ToString() + " Division 3: " + speedDivisionTracker[2].ToString() + " Division 4: " + speedDivisionTracker[3].ToString()
+ " Division 5: " + speedDivisionTracker[4].ToString());
        }
        // Sense Radius
        void LogSenseRadius()
        {
            saveData.CreatePreySenseRadiusLog(" Sense Radius: Time: " + Mathf.FloorToInt(Timer.time).ToString() + " Division 1: " +
senseRadiusDivisionTracker[0].ToString() + " Division 2: " + senseRadiusDivisionTracker[1].ToString() + " Division 3: " +
senseRadiusDivisionTracker[2].ToString() + " Division 4: " + senseRadiusDivisionTracker[3].ToString() + " Division 5: " +
senseRadiusDivisionTracker[4].ToString());
        }

        // Update sexual statistics text
        void SexualCreatureTextUpdate()
        {
            // SEXUAL STATS
            // Update the total sexual creature count text
```

```
        sexualCreatureCountText.text = "Total Sexual Creatures: " + sexualCreatureCount.ToString();
        // Update the total male sexual creature count text
        maleSexualCreatureCountText.text = "Total Male Creatures: " + maleSexualCreatureCount.ToString();
        // Update the total female sexual creature count text
        femaleSexualCreatureCountText.text = "Total Female Creatures: " + femaleSexualCreatureCount.ToString();
        // Update the all time total sexual creature count text
        allTimeSexualCreatureCountText.text = "Total Sexual Creatures Born: " + allTimeSexualCreatureCount.ToString();
        // Update the all time total sexual creature count text
        allTimeMaleSexualCreatureCountText.text = "Total Male Creatures Born: " + allTimeMaleCreatureCount.ToString();
        // Update the all time total female sexual creature count text
        allTimeFemaleSexualCreatureCountText.text = "Total Female Creatures Born: " + allTimeFemaleCreatureCount.ToString();
    }

    // Update asexual statistics text
    void AsexualCreatureTextUpdate()
    {
        // ASEXUAL STATS
        // Update the total asexual creature count text
        asexualCreatureCountText.text = "Total Asexual Creatures: " + asexualCreatureCount.ToString();
        // Update the all time total asexual creature count text
        allTimeAsexualCreatureCountText.text = "Total Asexual Creatures Born: " + allTimeAsexualCreatureCount.ToString();
    }

    // Update hermaphrodite statistics text
    void HermaphroditeCreatureTextUpdate()
    {
        // HERMAPHRODITE STATS
        // Update the total hermaphrodite creature count text
        hermaphroditeCreatureCountText.text = "Total Hermpahrodite Creatures: " + hermaphroditeCreatureCount.ToString();
        // Update the all time total hermaphrodite creature count text
        allTimeHermaphroditeCreatureCountText.text = "Total Hermaphrodite Creatures Born: " + allTimeHermaphroditeCreatureCount.ToString();
    }

    // DETERMINE DIVISION OF TRAITS
    // Size
    public int getSizeDivision(float size)
    {
        int division = 0;
        // compare size to divisions
        for (int i = 0; i < sizeDivisions.Length; i++)
        {
            // check if the size is smaller than or equal to the division
            if (size >= sizeDivisions[i])
                division = i;
            // if not then continue
        }
        // return the division for later use in
        return division;
    }
    // Speed
    public int getSpeedDivision(float speed)
    {
        int division = 0;
        // compare speed to divisions
        for (int i = 0; i < speedDivisions.Length; i++)
        {
            // check if the speed is smaller than or equal to the division
            if (speed >= speedDivisions[i])
                division = i;
            // if not then continue
        }
        // return the division for later use in
        return division;
    }
    // Sense Radius
    public int getSenseRadiusDivision(float senseRadius)
    {
        int division = 0;
        // compare sense radius to divisions
        for (int i = 0; i < senseRadiusDivisions.Length; i++)
        {
            // check if the sense radius is smaller than or equal to the division
            if (senseRadius >= senseRadiusDivisions[i])
                division = i;
            // if not then continue
        }
        // return the division for later use in
        return division;
    }
}
```

Predator Statistics: (225 total lines, 211 sloc)

```
using UnityEngine;
using UnityEngine.UI;

public class PredatorStatistics : MonoBehaviour
{
    // STATIC VARIABLES FOR TRACKING
    // Sexual
    public static int predatorSexualCount;
    public static int allTimePredatorSexualCount;
    // Asexual
    public static int predatorAsexualCount;
    public static int allTimePredatorAsexualCount;
    // Hermaphrodite
    public static int predatorHermaphroditeCount;
    public static int allTimePredatorHermaphroditeCount;
    // TRAITS
    [Header("Trait Tracking")]
    // Size
    public float[] sizeDivisions;
    public static int[] sizeDivisionTracker;
    // Speed
    public float[] speedDivisions;
    public static int[] speedDivisionTracker;
    // Sense Radius
    public float[] senseRadiusDivisions;
    public static int[] senseRadiusDivisionTracker;

    // TEXT OBJECTS FOR DISPLAYING STATISTICS
    // Sexual
    [Header("Pred. Sexual Stat Text")]
    public Text predatorSexualCountText;
    public Text allTimePredatorSexualCountText;
    // Asexual
    [Header("Pred. Asexual Stat Text")]
    public Text predatorAsexualCountText;
    public Text allTimePredatorAsexualCountText;
    // Hermaphrodite
    [Header("Pred. Hermaphrodite Stat Text")]
    public Text predatorHermaphroditeCountText;
    public Text allTimePredatorHermaphroditeCountText;

    // COLLECTING DATA
    [Header("Collecting Data")]
    public float setRecordInterval;
    // run time effected countdown
    private float recordInterval;
    // reference to the SaveData script allowing for writing to logs
    public SaveData saveData;
    // reference to the Timer script allowing for access to it's values
    public Timer timer;

    private void Start()
    {
        // RESET ALL VALUES
        // this ensures that when the functions used to "reload" the scene static statistic values start at 0
        // Reset all time stats
        allTimePredatorSexualCount = 0;
        allTimePredatorAsexualCount = 0;
        allTimePredatorHermaphroditeCount = 0;
        // Reset current stats
        predatorSexualCount = 0;
        predatorAsexualCount = 0;
        predatorHermaphroditeCount = 0;

        // Initialize trait division tracker arrays
        // Size
        sizeDivisionTracker = new int[5];
        // Speed
        speedDivisionTracker = new int[5];
        // Sense Radius
        senseRadiusDivisionTracker = new int[5];
    }

    // Set RecordingInterval
    public void SetRecordingInterval()
    {
```

```
        // Set recordingInterval
        setRecordInterval = MenuManager.setDataRecordingInterval;
    }

    // Update is called once per frame
    void Update()
    {
        // Update relevant statistics of PREY
        if (MenuManager.activePredator[0])
            SexualCreatureTextUpdate();
        if (MenuManager.activePredator[1])
            AsexualCreatureTextUpdate();
        if (MenuManager.activePredator[2])
            HermaphroditeCreatureTextUpdate();

        // DATA COLLECTION
        // countdown from the set recording interval
        recordInterval -= Time.deltaTime;
        // once the countdown reaches zero record the proper data, and reset the timer
        if (recordInterval <= 0)
        {
            // restart countdown
            recordInterval = setRecordInterval;
            // log relevant statistics
            if (MenuManager.activePredator[0])
                LogSexual();
            if (MenuManager.activePredator[1])
                LogAsexual();
            if (MenuManager.activePredator[2])
                LogHermaphrodite();
            // Log trait stats
            LogSize();
            LogSpeed();
            LogSenseRadius();
        }
    }

    // LOGGING FUNCTIONS
    // the following functions tell saveData what new content to add to the file "Log.txt"
    // Asexual
    void LogAsexual()
    {
        saveData.CreatePredatorPopulationLog("Time: " + Mathf.FloorToInt(Timer.time).ToString() + " Predators: " + predatorAsexualCount.ToString() + "
All Time Predators:  " + allTimePredatorAsexualCount.ToString());
    }
    // Sexual
    void LogSexual()
    {
        saveData.CreatePredatorPopulationLog("Time: " + Mathf.FloorToInt(Timer.time).ToString() + " Predators: " + predatorSexualCount.ToString() + "
All Time Predators: " + allTimePredatorSexualCount.ToString());
    }
    // Hermaphrodite
    void LogHermaphrodite()
    {
        saveData.CreatePredatorPopulationLog("Time: " + Mathf.FloorToInt(Timer.time).ToString() + " Predators: " +
predatorHermaphroditeCount.ToString() + " All Time Predators: " + allTimePredatorHermaphroditeCount.ToString());
    }
    // TRAIT STATS
    // Size
    void LogSize()
    {
        saveData.CreatePredatorSizeLog("Size: Time: " + Mathf.FloorToInt(Timer.time).ToString() + " Division 1: " + sizeDivisionTracker[0].ToString() +
" Division 2: " + sizeDivisionTracker[1].ToString() + " Division 3: " + sizeDivisionTracker[2].ToString() + " Division 4: " + sizeDivisionTracker[3].ToString() + "
Division 5: " + sizeDivisionTracker[4].ToString());
    }
    // Speed
    void LogSpeed()
    {
        saveData.CreatePredatorSpeedLog("Speed: Time: " + Mathf.FloorToInt(Timer.time).ToString() + " Division 1: " +
speedDivisionTracker[0].ToString() + " Division 2: " + speedDivisionTracker[1].ToString() + " Division 3: " + speedDivisionTracker[2].ToString() + " Division 4: " +
speedDivisionTracker[3].ToString() + " Division 5: " + speedDivisionTracker[4].ToString());
    }
    // Sense Radius
    void LogSenseRadius()
    {
        saveData.CreatePredatorSenseRadiusLog(" Sense Radius: Time: " + Mathf.FloorToInt(Timer.time).ToString() + " Division 1: " +
senseRadiusDivisionTracker[0].ToString() + " Division 2: " + senseRadiusDivisionTracker[1].ToString() + " Division 3: " +
senseRadiusDivisionTracker[2].ToString() + " Division 4: " + senseRadiusDivisionTracker[3].ToString() + " Division 5: " +
senseRadiusDivisionTracker[4].ToString());
    }
```

```
    // Update sexual statistics text
    void SexualCreatureTextUpdate()
    {
        // SEXUAL STATS
        // Update the total sexual creature count text
        predatorSexualCountText.text = "Total Pred. Sexual: " + predatorSexualCount.ToString();
        // Update the all time total sexual creature count text
        allTimePredatorSexualCountText.text = "Total Pred. Sexual Born: " + allTimePredatorSexualCount.ToString();
    }

    // Update asexual statistics text
    void AsexualCreatureTextUpdate()
    {
        // ASEXUAL STATS
        // Update the total asexual creature count text
        predatorAsexualCountText.text = "Total Pred. Asexual: " + predatorAsexualCount.ToString();
        // Update the all time total asexual creature count text
        allTimePredatorAsexualCountText.text = "Total Pred. Asexual Born: " + allTimePredatorAsexualCount.ToString();
    }

    // Update hermaphrodite statistics text
    void HermaphroditeCreatureTextUpdate()
    {
        // HERMAPHRODITE STATS
        // Update the total hermaphrodite creature count text
        predatorHermaphroditeCountText.text = "Total Pred. Hermaphrodite: " + predatorHermaphroditeCount.ToString();
        // Update the all time total hermaphrodite creature count text
        allTimePredatorHermaphroditeCountText.text = "Total Pred. Hermaphrodite Born: " + allTimePredatorHermaphroditeCount.ToString();
    }

    // DETERMINE DIVISION OF TRAITS
    // Size
    public int getSizeDivision(float size)
    {
        int division = 0;
        // compare size to divisions
        for (int i = 0; i < sizeDivisions.Length; i++)
        {
            // check if the size is smaller than or equal to the division
            if (size >= sizeDivisions[i])
                division = i;
            // if not then continue
        }
        // return the division for later use in
        return division;
    }
    // Speed
    public int getSpeedDivision(float speed)
    {
        int division = 0;
        // compare speed to divisions
        for (int i = 0; i < speedDivisions.Length; i++)
        {
            // check if the speed is smaller than or equal to the division
            if (speed >= speedDivisions[i])
                division = i;
            // if not then continue
        }
        // return the division for later use in
        return division;
    }
    // Sense Radius
    public int getSenseRadiusDivision(float senseRadius)
    {
        int division = 0;
        // compare sense radius to divisions
        for (int i = 0; i < senseRadiusDivisions.Length; i++)
        {
            // check if the sense radius is smaller than or equal to the division
            if (senseRadius >= senseRadiusDivisions[i])
                division = i;
            // if not then continue
        }
        // return the division for later use in
        return division;
    }

}
```

Log Statistics: (145 total lines, 138 sloc)

```
using UnityEngine;
using System.IO;

public class SaveData : MonoBehaviour
{
   string fullPath;

   // Create New Folder
   public void CreateNewLogFolder(string folderName)
   {
      // path of the file
      string path = Application.dataPath + "/" + folderName;
      // create directory if it doesnt exist
      if (!Directory.Exists(path))
      {
         var folder = Directory.CreateDirectory(path);
      }
      // set the path for other functions to reference
      fullPath = path;

      // Create sub folders for prey, and predator data
      string prey = fullPath + "/prey";
      // create directory if it doesnt exist
      if (!Directory.Exists(prey))
      {
         var folder = Directory.CreateDirectory(prey);
      }

      string predators = fullPath + "/predators";
      // create directory if it doesnt exist
      if (!Directory.Exists(predators))
      {
         var folder = Directory.CreateDirectory(predators);
      }
   }

   // LOGGING OF PREY
   // Population tracking
   public void CreatePreyPopulationLog(string content)
   {
      // path of the file
      string path = fullPath + "/prey/PopulationsLog.txt";
      // create file if it doesnt exist
      if (!File.Exists(path))
      {
         File.WriteAllText(path, "Population Logs:\n");
      }
      // add the content
      File.AppendAllText(path, content + "\n");
   }

   // size
   public void CreatePreySizeLog(string content)
   {
      // path of the file
      string path = fullPath + "/prey/SizeLog.txt";
      // create file if it doesnt exist
      if (!File.Exists(path))
      {
         File.WriteAllText(path, "Prey Size Logs:\n");
      }
      // add the content
      File.AppendAllText(path, content + "\n");
   }
   // speed
   public void CreatePreySpeedLog(string content)
   {
      // path of the file
      string path = fullPath + "/prey/SpeedLog.txt";
      // create file if it doesnt exist
      if (!File.Exists(path))
      {
         File.WriteAllText(path, "Prey Speed Logs:\n");
      }
      // add the content
      File.AppendAllText(path, content + "\n");
   }
```

```
    // sense radius
    public void CreatePreySenseRadiusLog(string content)
    {
        // path of the file
        string path = fullPath + "/prey/SenseRadiusLog.txt";
        // create file if it doesnt exist
        if (!File.Exists(path))
        {
            File.WriteAllText(path, "Prey SenseRadius Logs:\n");
        }
        // add the content
        File.AppendAllText(path, content + "\n");
    }

    // LOGGING OF PREDATORS
    // Population tracking
    public void CreatePredatorPopulationLog(string content)
    {
        // path of the file
        string path = fullPath + "/predators/PopulationsLog.txt";
        // create file if it doesnt exist
        if (!File.Exists(path))
        {
            File.WriteAllText(path, "Predator Population Logs:\n");
        }
        // add the content
        File.AppendAllText(path, content + "\n");
    }
    // size
    public void CreatePredatorSizeLog(string content)
    {
        // path of the file
        string path = fullPath + "/predators/SizeLog.txt";
        // create file if it doesnt exist
        if (!File.Exists(path))
        {
            File.WriteAllText(path, "Predator Size Logs:\n");
        }
        // add the content
        File.AppendAllText(path, content + "\n");
    }
    // speed
    public void CreatePredatorSpeedLog(string content)
    {
        // path of the file
        string path = fullPath + "/predators/SpeedLog.txt";
        // create file if it doesnt exist
        if (!File.Exists(path))
        {
            File.WriteAllText(path, "Predator Speed Logs:\n");
        }
        // add the content
        File.AppendAllText(path, content + "\n");
    }
    // sense radius
    public void CreatePredatorSenseRadiusLog(string content)
    {
        // path of the file
        string path = fullPath + "/predators/SenseRadiusLog.txt";
        // create file if it doesnt exist
        if (!File.Exists(path))
        {
            File.WriteAllText(path, "Predator SenseRadius Logs:\n");
        }
        // add the content
        File.AppendAllText(path, content + "\n");
    }
}
```

Menu Management: (281 total lines, 246 sloc)

```
using UnityEngine;
using UnityEngine.UI;
using System;
using UnityEngine.SceneManagement;

public class MenuManager : MonoBehaviour
{
    // MENUS
    // Statistics
```

```
public GameObject statisticsMenu;
// Main Menu
public GameObject mainMenu;
// Setup Menu
public GameObject setupMenu;
// Pause Menu
public GameObject pauseMenu;

// PAUSING
public bool paused;
public bool simStarted;

// TIME LIMIT
public Timer timer;
// time
public float setTimeScale;
public static float timeScale;

// CREATURE TYPE
// For interactivity between other scripts
public static bool[] activeCreatures = new bool[3];
public static int activeCreatureIndex;

// CREATURE SPAWNING
public GameObject[] creatures;
public static int initialCreatureCount;
public CreatureSpawning creatureSpawning;

// PREDATOR TYPE
// For interactivity between other scripts
public static bool[] activePredator = new bool[4];
public static int activePredatorIndex;

// PREDATOR SPAWNING
public GameObject[] predators;
public static int initialPredatorCount;
public PredatorSpawning predatorSpawning;


// CREATURE STATISTICS
public GameObject[] creatureStats;

// PREDATOR STATISTICS
public GameObject[] predatorStats;

// FOOD STUFF
// Food Manager
public GameObject foodManager;
// Initial Food Amount
public static int initalFood;
// Food Spawn Rate
public static float foodSpawnRate;

// DATA COLLECTION / LOGGING
// New log sections / setting recording interval
public CreatureStatistics creatureStatistics;
// Setting recording interval
public PredatorStatistics predatorStatistics;
// general logging
public static float setDataRecordingInterval;
public static string logFolderName;


// Start is called before the first frame update
void Start()
{
    // set menus active or inactive
    statisticsMenu.SetActive(false);
    mainMenu.SetActive(true);
    setupMenu.SetActive(false);
    pauseMenu.SetActive(false);
    // set food manager inactive
    foodManager.SetActive(false);
    // make sure nothing is simulated
    Time.timeScale = 0;
    // initialize pausing, and simulation state management bools
    paused = false;
    simStarted = false;
    // reset active predator and creature (prey) bools
    // predators
```

```
    for (int i = 0; i < activePredator.Length; i++)
    {
        activePredator[i] = false;
    }
    // prey
    for (int i = 0; i < activeCreatures.Length; i++)
    {
        activeCreatures[i] = false;
    }

    // set timeScale
    timeScale = setTimeScale;
}

private void Update()
{
    // PAUSE
    // if "space" key is pressed, the simulation is not paused, and the simulation has started then pause the simulation
    if (Input.GetKeyDown(KeyCode.Space) && !paused && simStarted)
    {
        // set true to make this statement can only run when not paused
        paused = true;
        // stop time
        Time.timeScale = 0;
        // show the pause menu
        pauseMenu.SetActive(true);
    }
}

// Generate Random FolderName
string RandomFolderNameGenerator(int length)
{
    string characters = "ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890";
    string generated_string = "";

    for (int i = 0; i < length; i++)
        generated_string += characters[UnityEngine.Random.Range(0, length)];

    return generated_string;
}

// THE FOLLOWING FUNCTIONS ARE CALLED WHEN BUTTONS ON THE "Pause Menu" ARE PRESSED

// END SIMULATION
public void EndSimulation()
{
    SceneManager.LoadScene(0);
}

// RESUME SIMULATION
public void ResumeSimulation()
{
    // stop showing the pause menu
    pauseMenu.SetActive(false);
    // set false to allow pausing to happen again
    paused = false;
    // start time
    Time.timeScale = timeScale;
}

// THE FOLLOWING FUNCTIONS ARE CALLED WHEN THE BUTTON "Run Simulation Button" IS PRESSED

// RUN A NEW SIMULATION
// this will close the setup menu, and begin running the simlation with the applied settings
public void RunSimulation()
{
    // set statistics menu active
    statisticsMenu.SetActive(true);
    // set food spawner active
    foodManager.SetActive(true);
    // begin simulating
    simStarted = true;
    Time.timeScale = timeScale;
    // begin logging
    creatureStatistics.initializeNewLog();
    predatorStatistics.SetRecordingInterval();
}

// SET CREATURE
public void SetCreature(Dropdown creatureType)
```

```
{
    // set creature type active
    creatures[creatureType.value].SetActive(true);
    activeCreatureIndex = creatureType.value;
    statisticsMenu.SetActive(true);
    // set creature statistics active
    creatureStats[creatureType.value].SetActive(true);
    activeCreatures[creatureType.value] = true;
}

// SET PREDATOR
public void SetPredator(Dropdown predatorType)
{
    // check if predatorType value is not 3 (none)
    if(predatorType.value != 3)
    {
        // set predator type active
        predators[predatorType.value].SetActive(true);
        activePredator[predatorType.value] = true;
        // set predator statistics active
        predatorStats[predatorType.value].SetActive(true);
    }
    // set activePredatorIndex for other scripts to act upon
    activePredatorIndex = predatorType.value;
}

// SET TIME LIMIT
public void SetTimeLimit(Dropdown timeLimit)
{
    // set time limit
    timer.activeTimeLimitIndex = timeLimit.value;
}

// SET INITIAL FOOD COUNT
public void SetInitialFood(InputField foodCount)
{
    // set the inital food count to the value of foodCount if the text is not effectively null
    if(foodCount.text != "")
        initalFood = Convert.ToInt32(foodCount.text);
}

// SET FOOD SPAWN RATE
public void SetFoodSpawnRate(InputField spawnRate)
{
    // set the food spawn rate to value of spawnRate if the text is not effectively null
    if (spawnRate.text != "")
        foodSpawnRate = float.Parse(spawnRate.text, System.Globalization.CultureInfo.InvariantCulture);
    else // if not set then default to 1 second
        foodSpawnRate = 1;
}

// SET DATA COLLECTION INTERVAL
public void SetDataCollectionRate(InputField collectionRate)
{
    // set the collection rate to value of collectionRate if the text is not effectively null
    if (collectionRate.text != "")
        setDataRecordingInterval = float.Parse(collectionRate.text, System.Globalization.CultureInfo.InvariantCulture);
    else // the value was not set so default to 2 seconds
        setDataRecordingInterval = 2f;
}

// SET NEW LOG FOLDER NAME
public void SetLogFolderName(InputField folderName)
{
    // set the folder name to the value of folderName if the text is not effectively null
    if (folderName.text != "")
        logFolderName = folderName.text;
    else // the value was not set so default to a random string
        logFolderName = RandomFolderNameGenerator(10);
}

// SET START AMOUNT OF CREATURES (prey)
public void SetInitialCreatureCount(InputField initialAmount)
{
    // set the inital creature count to value of initialAmount if the text is not effectively null
    if (initialAmount.text != "")
        initialCreatureCount = Convert.ToInt16(initialAmount.text, System.Globalization.CultureInfo.InvariantCulture);
    else // default to four creatures (prey)
        initialCreatureCount = 4;
    // start spawning
```

```
        SpawnInitialCreatures();
    }

    // START SPAWNING OF PREY
    void SpawnInitialCreatures()
    {
        // start spawning of inital creatures
        creatureSpawning.spawnInitialCreatures(initialCreatureCount, activeCreatureIndex);
    }

    // SET START AMOUNT OF PREDATORS
    public void SetInitalPredatorCount(InputField initialAmount)
    {
        // set the inital predator count to value of initialAmount if the text is not effectively null
        if (initialAmount.text != "")
            initialPredatorCount = Convert.ToInt16(initialAmount.text, System.Globalization.CultureInfo.InvariantCulture);
        else // default to four predators
            initialPredatorCount = 4;
        if (activePredatorIndex != 3)
        {
            // start spawning
            SpawnInitalPredators();
        }
    }

    // START SPAWNING OF PREDATORS
    void SpawnInitalPredators()
    {
        // start spawning of inital predators
        predatorSpawning.spawnInitialPredators(initialPredatorCount, activePredatorIndex);
    }
}
```

Timer: (41 total lines, 36 sloc)

```
using System;
using UnityEngine;
using UnityEngine.UI;

public class Timer : MonoBehaviour
{
    // TEXT OBJECTS FOR DISPLAYING TIME
    [Header("General")]
    public Text timerText;
    public static float time = 0;

    // VALUES OF TIME FOR THE TIME LIMIT
    [Header("Time Limit")]
    public float[] timeLimits;
    public int activeTimeLimitIndex;
    // reference to menu manager for resetting
    public MenuManager menuManager;

    private void Start()
    {
        // INITIALIZE VALUES
        time = 0;
    }

    // Update is called once per frame
    void Update()
    {
        // Update time value
        time += Time.deltaTime;
        TimeSpan timeSpan = TimeSpan.FromSeconds(0 + time);
        // Format the time
        string timeText = string.Format("{0:D2}:{1:D2}:{2:D2}", timeSpan.Hours, timeSpan.Minutes, timeSpan.Seconds);
        timerText.text = timeText;

        // Time Limit
        // if time limit is reached and there is a finite time limit then end if the simulation.
        if (time >= timeLimits[activeTimeLimitIndex] && activeTimeLimitIndex != 8)
            // end simulation
            menuManager.EndSimulation();
    }
}
```

Food Manager: (72 total lines, 63 sloc)

```
using UnityEngine;

public class FoodManager : MonoBehaviour
{
   // FOOD SPAWN SETTINGS
   [Header("Food Spawn Settings (Start)")]
   public int foodSourceAmount;
   // min and max (x,y) values for food spawn locations
   public float maxX;
   public float maxY;
   public float minX;
   public float minY;
   // food spawn timing
   private float timeBtwSpawn;
   public float setTimeBtwSpawn;
   // the good source gameObject
   public GameObject foodSource;

   private Transform foodSourcesHolder;

   // Start is called before the first frame update
   void Start()
   {
      // Set inital values from settings selected by user
      foodSourceAmount = MenuManager.initalFood;
      setTimeBtwSpawn = MenuManager.foodSpawnRate;
      // Initialize food
      foodSourcesHolder = gameObject.transform.GetChild(0);
      generateInitialFood();
      timeBtwSpawn = setTimeBtwSpawn;
   }


   private void Update()
   {
      timeBtwSpawn -= Time.deltaTime;
      if (timeBtwSpawn <= 0)
      {
         // spawn a new piece of food
         generateNewFood();
         // reset the timer
         timeBtwSpawn = setTimeBtwSpawn;
      }
   }

   // Generate Food
   void generateInitialFood()
   {
      for (int i = 0; i < foodSourceAmount; i++)
      {
         // generate random position
         float x = Random.Range(minX, maxX);
         float y = Random.Range(minY, maxY);
         Vector3 spawnPos = new Vector3(x, y, 1);

         // spawn food source
         Instantiate(foodSource, spawnPos, Quaternion.identity, foodSourcesHolder);
      }
   }

   // Generate a single new piece of food
   void generateNewFood()
   {
      // generate random position
      float x = Random.Range(minX, maxX);
      float y = Random.Range(minY, maxY);
      Vector3 spawnPos = new Vector3(x, y, 1);

      // spawn food source
      Instantiate(foodSource, spawnPos, Quaternion.identity, foodSourcesHolder);
   }
}
```

Boundary Manager: (37 total lines, 33 sloc)

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

public class BorderManagement : MonoBehaviour
```

```
{
    // SETTINGS
    [Header("Settings")]
    // variable checked by creatures
    public static bool staticBounds;
    // Variable used to interact with staticBounds from the inspector.
    public bool setStaticBounds;

    // BORDERS
    [Header("Borders")]
    public GameObject staticBorder;
    public GameObject periodicBorder;

    // Awake()
    private void Awake()
    {
        // initialize border type
        staticBounds = setStaticBounds;
        // deactivate static border by default.
        staticBorder.SetActive(false);
        // deactivate periodic border by default.
        periodicBorder.SetActive(false);

        // Check if it has static bounds
        if (staticBounds)
            // if it does have static bounds then enable the border gameObject
            staticBorder.SetActive(true);
        else
            // if it doesnt then creatures will have periodic bounds, and will essentially have a looping border.
            periodicBorder.SetActive(true);
    }
}
```