**Project Description**

EcoSim is an application that aims to simulate basic ecosystems. The process starts by generating a randomized terrain. From there, animals will be placed in the simulation. Animal types can be sheep, wolves, bunnies, foxes, and more. The user can choose to let the simulation run on its own, resulting in a completely randomized simulation cycle. The user also has the choice to modify specific aspects of the simulation, such as offspring rate, food growth rate, average lifespan, and much more. The simulation aims to cover a large range of real-world aspects, such as population levels, prey vs. predator, mutations, and much more.

**Competitive Analysis**

**Structural Plan**

Animals: Every animal type will share the same core attributes (i.e. each animal will have hunger levels, thirst levels, health levels, offspring rate, mutation rate, etc.). Each animal will inherit these attributes from the parent class, Animal. Each animal type, however, will have their own specific starting attribute values. For example, a sheep’s starting strength levels will be far less than a wolf’s starting strength levels. Each animal type’s class code will also be stored in its own file for organization.

Assets: Each asset for every animal type will be stored in the ‘assets’ directory. Some animal types will have multiple assets, such as one asset for a blue sheep, one asset for a sheep with a specific mutation, one asset for a sheep who’s sick, etc. This is the goal, but I may not be able to reach it in time.

General Code: Most of the code will be stored in main.py. As of right now, the plan is to have all of the code pertaining to the simulation is main.py in what seems like the most intuitive, chronologically organized way possible.

**Algorithmic Plan**

Animals Interacting in the Ecosystem (very hard)

This will be the hardest, most algorithmically complex feature to implement by far. There are many parts to this system, of which I will list in the order that they must be implemented:

1. Dynamically create instances of animal objects
2. Dynamically display images of the animal objects in the terrain
3. Use pathfinding-like technology that will allow animals to chase after one another, water, food, etc.
4. Create more instances of animal objects when animals have offspring
5. Delete instances of animal objects when they die, and also delete their image from the screen
6. Modify animal attributes on initialization to reflect when an animal has mutated
7. While doing all of this, accurately keep track of each instance’s attributes

Terrain Generation (hard)

This feature will also be algorithmically complex. There will be four main parts to the terrain generation:

1. Grass (which animals can eat)
2. Bare land (when there is nothing there to eat)
3. Flowers (extra-nutritious food)
4. Water (which animals can drink and possibly swim in)

There are currently two possible algorithms for the terrain generation, each one looking something similar to the following pseudocode:

1. Option 1 (the less likely candidate)
   1. Generate a 2D list with dimensions equal to the dimensions of the grid we will be generating. Randomly populate this list with the colors ‘green’ (grass), ‘tan’ (no grass), ‘pink/red/purple’ (flowers), ‘blue’ (water).
   2. Draw a grid by drawing the cells one by one (like we did in the case studies). Each cell will correspond to an element in the 2D list we generated. That way each cell will also have a color associated to it.
      1. At this stage, we have a grid with a completely randomized terrain. Nothing looks good or makes sense.
   3. To make it look nicer, we have to group the water together into bodies of water. This is how we do that: while drawing each cell, if we run into a blue cell, we look at all of the surrounding cells a few times, and if they are valid placements, we also make those cells blue. That we have a cohesive body of water.
      1. We would only do this 2-3 times, or else most of the grid would be water
   4. The random pink/red/purple cells would represent flowers
2. Option 2 (the much more likely candidate)
   1. Generate a grid with all grass (green cells)
   2. Complete this process 2-3 times
      1. Pick a random cell on the grid
      2. Make it blue
      3. Look at all surrounding cells a random number of layers (this will make the size of the body of water vary each time the simulation is run). If it’s a valid cell, make it blue.
         1. We now have bodies of water
   3. Complete this process a random number of times (0-20 inclusive)
      1. Pick a random cell on the grid
      2. Pick a random color (either ‘pink’, ‘red’, or ‘purple’)
      3. Make that cell that color
         1. We now have flowers
   4. Complete this process a random number of times
      1. Pick a random cell on the grid
      2. If it is green, make it tan
         1. We now have patches of grassless land

Graphing (Average/Easy)

Time allowing, I plan on creating my own graphing system to graph the current population of each animal type. To do this, I would use cmu\_112\_graphics to plot lines based off of the current data of each animal type (this is average in difficulty). If I run out of time, I will use matplotlib (easy in difficulty).

User Interaction (sliders impact the simulation) (Easy)

This feature shouldn’t be hard to implement, but it could be time consuming. As the user adjusts sliders on the right-hand side of the screen, the simulation should also change. For example, if the user changes ‘grass growth rate’ to 1, then the grass will grow back much slower than if the slider were at 9. The sliders will be coded from scratch, rather than using a module that has sliders built in. This will somewhat help increase algorithmic complexity. This process will work as so:

1. Create text that will act as the label for each slider
2. Create a line that the slider will ‘slide’ on
3. Create a rectangle that will rest on the line and act as the slider
4. If the user drags the slider, it will slide along the same x-axis and it will not go off of the line

There are currently two possible approaches on how to implement the feature where the slider directly impacts the simulation:

1. Option 1 (the less likely candidate)
   1. Each slider will be linked to a variable. That variable will change the simulation live
2. Option 2 (the more likely candidate, strictly due to time constraints)
   1. Each slider will be linked to a variable. The sliders can only be changed *before* the simulation is run. The sliders cannot be modified during the simulation, and therefore the simulation cannot be changed live.

**Timeline Plan**

Complete by 8:00 PM on November 18, 2021

* Terrain Generation Part 1 (Bodies of Water)
* Basic slider implementation
* Dynamically initialize animals
* Animals follow each other

Complete by 8:00 PM on November 23, 2021

* Any features I did not finish from the last deadline (hopefully this won’t happen)
* Users can change the simulation by changing the sliders
* Animals are ‘alive’ (i.e., they move, eat, drink, hunt, die, etc.)
* Terrain Generation Part 2 (Flowers)
* Implement all sliders
* Mutation
* Graphing

Complete by 5:00 PM on December 1, 2021

* Any features I did not finish from the last deadline (hopefully this won’t happen)
* TP Video
* Fun ideas for after MVP is reached
  + More mutations
  + More animal types
  + Climate Change
  + Humans cutscene-like feature

**Version Control Plan**

All files are being backed up to GitHub. I already have a repo (currently private so that my mentor, Asad, can look at the ideas I've come up with) that is currently storing all versions of my code.

A screenshot of a computer

Description automatically generated with medium confidence

**Modules**

I will be using cmu\_112\_graphics. One feature I would like to implement requires graphing. Time allowing, I will use cmu\_112\_graphics and code my own graphing software. If I run out of time, I will use matplotlib. I won’t know whether I will use matplotlib until I decide whether I will be implementing this feature or not.