Evolution Simulator

H446-03 - Programming Project

Benedict Cameron John Vlasto

Candidate no: 6503

Centre, centre no: Eton College, 51537

Analysis

Problem identification

Evolution is not intuitive, and as a topic it was largely brushed over during GCSE biology and never covered before that. As such, a visual and interactive simulation of evolution – especially for subtle behaviours like altruism – would make the topic more interesting and much easier to grasp. As a learning tool, I would have loved something like this when formally covering the topic as it would help clarify the impact of changing environmental factors on a population's behaviour and traits. The software could therefore help teachers explain the concept of evolution to their students and ensure they truly understand it.

The features of a computer system that would be required for this solution would be a computer with hardware capable of accepting and processing user inputs, light data processing, and a suitable operating system for the software to run on. The problem lends itself to a computational solution because simulations of this kind can only practically be run on a computer, especially given the program must adapt in real time based on user inputs. Described in its simplest form, the solution would take inputs from a user to design a virtual environment, populate the environment with creatures designed by the user, and then simulate the development of the species as it 'evolves'.

Stakeholders

The clients and demographic for this software would be primarily biology students and teachers. To ensure both these demographics are represented, one stakeholder will be a biology teacher and the other a GCSE student studying biology.

Biology teachers must teach evolution as part of the GCSE syllabus, but it is often a misunderstood topic among students – my own class spent several lessons ironing out misunderstandings. My solution would help biology teachers to explain the concept to their students, as well as make it intuitive so the students' understanding goes beyond rote learning. A teacher could use the software as an interactive display of evolution in action, and even allow the students to experiment with the software themselves. The stakeholder for the teacher demographic is Kerri Hicks, an experienced biology teacher who has taught evolution to many years of GCSE students.

The other demographic for the project is current GCSE biology students. With GCSE exams coming up, they could use my solution to ensure they understand the topic of evolution, both for their own curiosity and to achieve the highest possible mark in the exam. The stakeholder for this demographic is G. L., a GCSE biology student who has previously studied evolution in class.

How stakeholders will use proposed solution

The biology teacher will use this program to help teach evolution. She could use it inside the classroom to make a lesson more engaging and to help the students understand how a change in the

environment changes the creatures within it. She could also use the program as part of homework, for instance by asking students to guess how a certain creature will evolve, then ask the students to actually see what happens using the program and compare the results.

The student could use the program both to make learning about evolution more fun, and as a tool to help solidify their understanding of evolution by seeing how small changes in an environment affect the creatures within it. This would help them understand the cause-and-effect relationship between environment and creature even when the creatures are changing randomly.

Why it is suited to a computational solution

This problem lends itself to computational methods of finding and implementing a solution for a variety of reasons. The solution will be software that runs a simulation which is iterative by nature, in which each iteration is based on the outcome of the one previous. Only computers can perform this kind of iteration very accurately and very fast, which means this kind of simulation can only practically be implemented with a computer. Moreover, computers can store previous iterations, allowing the change over each iteration to be recorded and studied, which again is only possible on a computer. There is no alternative that is not a computational solution.

The aspects of the solution controlled by the software is the iterative simulation, after the initial conditions have been set by the user using either a mouse and keyboard or a touchscreen.

Computational methods that the solution lends itself to

Problem recognition

The overall problem is helping students studying biology to better understand the topic of evolution. However, the actual underlying problem is creating a simulation of evolution that is simplified far enough that the process becomes intuitive, whilst retaining enough accuracy to real life that it's a useful learning tool. When this is overcome the rest of the solution is simply implementing the evolution of different traits, and allowing the user to change the environment (e.g. number of resources) in which the virtual creatures will live.

Problem decomposition

This problem can be decomposed into a set of much smaller steps. This is an outline of what those steps are:

1. Present a menu of options to the user so they can select the trait they wish to observe evolving.
2. Allow the user to create an environment for the virtual creatures.
3. Run iterations of the simulated environment until the user stops it.
4. Display the results of the simulation (e.g. how competing populations sizes changed over time) in a user-friendly interface.

When these steps are done, the process is completed. This means the problem can be clearly broken down, and hence is a perfect candidate for problem decomposition.

Divide and conquer

Although these steps are challenging on their own, they're each manageable. Solving these steps on their own and then combining them into a modular program makes use of the divide and conquer method of problem solving. Because the problem can be broken down into the above steps, the divide and conquer method is applied here.

Abstraction

Evolution is an enormously complex process that takes millions of years and involves minute, subtle changes. As a learning tool, an evolution simulator doesn't need to accurately reflect this, so lots of the detail can be entirely removed whilst keeping the essence of the process the same. The simulation should be only as complex as the simplest version of evolution, which involves abstracting the process to its fundamentals.

The complexities of the software itself will be abstracted from the user so that the interface they see is kept as simple as possible, and the details removed from real evolution will only be revealed as much as is needed for the software to be a useful learning tool.

Interview

Interview questions

I will outline some of the key questions that I will ask each stakeholder, but after the interview I may ask follow up questions or ask them to elaborate on certain answers. The questions will find their opinion on the software and how they would like to use it. I asked the same questions of both the teacher and student demographic, because the questions are equally relevant to both. This interview was conducted virtually, via survey.

My questions to the student and teacher are as follows:

1. Have you ever used a biology simulation to teach/learn in the past?
2. If yes, was it useful? What did you like about it?
3. Do you think an evolution simulator would be useful to you, as a teaching/learning tool?
4. How do you envision an evolution simulator? What trait(s) would you like to see evolving?
5. How would you use this software?
6. Do you have anything else to add?

Questions one and two establish their history with biology simulations and what they took from the simulations they used. This is important because it's useful to know if their opinion is informed by previous experience.

Questions three and four investigate whether or not they would use an evolution simulator, and what they want from it – this will make up a large part of what informs the success criteria, because it will tell me what my stakeholders are looking for in my solution.

Question five asks how they would use the simulator, which will also inform the success criteria.

The questions conclude with asking if they have anything to say that I may have missed.

Answers

G. L. (student)

1. **Have you ever used a biology simulation to teach/learn in the past?**

   "Yes."

2. **If yes, was it useful? What did you like about it?**

   "It was very useful in visualising information for the more difficult concepts to understand at first like osmosis. I didn't have a simulator for evolution so a software like this might have been useful in understanding at the time."

3. **Do you think an evolution simulator would be useful to you, as a teaching/learning tool?**

   "Yes."

4. **How do you envision an evolution simulator? What trait(s) would you like to see evolving?**

   "It would be fascinating to see how creatures and organisms develop and evolve over the years and to see their gradual features improve such as capability, intelligence, size, ability to carry out tasks."

5. **How would you use this software?**

   "I wouldn't personally find much use for the software besides leisurely as I didn't struggle with the concept myself. Evolution would still be fascinating to learn about through the software."

6. **Do you have anything else to add?**

   No response.

Kerri Hicks (teacher)

1. **Have you ever used a biology simulation to teach/learn in the past?**

   "Yes."

2. **If yes, was it useful? What did you like about it?**

   "Yes. This simulation: http://www.kscience.co.uk/animations/anim_2.htm (you need to scroll down to the bottom). I use it every time I teach enzyme kinetics. Very useful as helps students visualise a process they can't usually see (because too small)."

3. **Do you think an evolution simulator would be useful to you, as a teaching/learning tool?**

"Yes."

4. **How do you envision an evolution simulator? What trait(s) would you like to see evolving?**

"If you want to tailor to the classic case studies you could do 'Darwin's Finches', Peppered Moths and/or Antibiotic Resistance in Bacteria. Otherwise any other interesting examples would be great. I anticipate the difficulty would be in building enough complexity for you to be able to change certain parameters (e.g. the selective pressures and selective advantages). How would you build in the fact that mutation is a random event? I suppose you would have a starting point that assumed the mutation did indeed happen then the simulation starts from there?"

5. **How would you use this software?**

"Depending on the level of the simulation's functionality I could use it simply as a visualisation of the process of evolution by natural selection, or even better have students investigate the possible outcomes of evolution by changing different parameters and seeing what the result is (which is what I do with the enzyme animation mentioned above)."

6. **Do you have anything else to add?**

"Nothing for now, but happy to answer any questions you have along the way!"

Analysis

Both stakeholders have used biology simulations in the past and would find them useful, suggesting my product is something they might want. It seemed that in both cases, the simulation was useful for less intuitive processes that they couldn't see or visualise – G. L. gave the example of osmosis, and Ms Hicks gave the example of enzyme kinetics. G. L. also said that an evolution simulator would have been useful when he was learning the topic, another indication that my product would be used. Also, evolution comes under processes that can't be easily seen or visualised.

G. L. said he would find the simulator "fascinating", and gave the specific traits of intelligence and size, as well as the much more vague "capability" and "ability to carry out tasks". Size is a very attainable trait to simulate the evolution of as it is a single trait with simple consequences: faster and stronger, but uses more energy. Intelligence, on the other hand, isn't possible in all its complexity but may be possible as a greatly simplified version; perhaps from intelligence to the ability to make simple decisions. He closed by saying that he would only use it as entertainment, as he doesn't personally need help with the topic.

Ms Hicks envisioned the simulator as a simulator of classic examples of natural selection: Darwin's finches, peppered moths, or antibiotic resistance. Peppered moths would work very

well with my simulator because their evolution was from white to black to better camouflage themselves during the Industrial Revolution. Not only is this change simple, it is by nature very visual, so would be a good first goal for simulation. She worries that the random nature of mutation will be lost, so I will make sure to emphasise the randomness of the process in the program. Her ideal use would be to allow students to "investigate the possible outcomes of evolution by changing different parameters" themselves, so customisation of the environment should be a heavy focus in my solution.

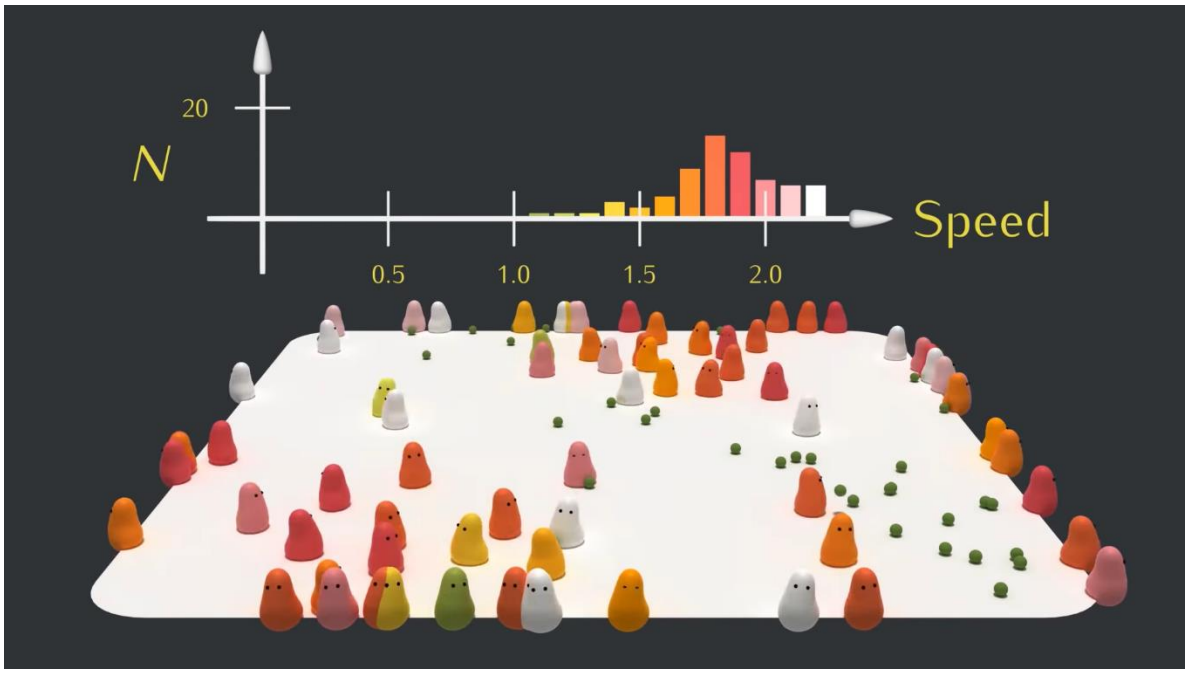
She closed by saying that she had nothing to add now.

Research existing solutions

Existing similar solutions

Primer

Overview:



Primer is an educational YouTube channel that creates videos exploring the evolution of different traits using animations created in Blender, a 3D modelling software. The animations are very colourful because traits are visually described in colour, and simple graphs are often displayed near the animation that summarise the changing population as shown above. Because the format is a video, however, there is no element of interactivity, and it's impossible to experiment with the software yourself to change either the creatures or the environment.

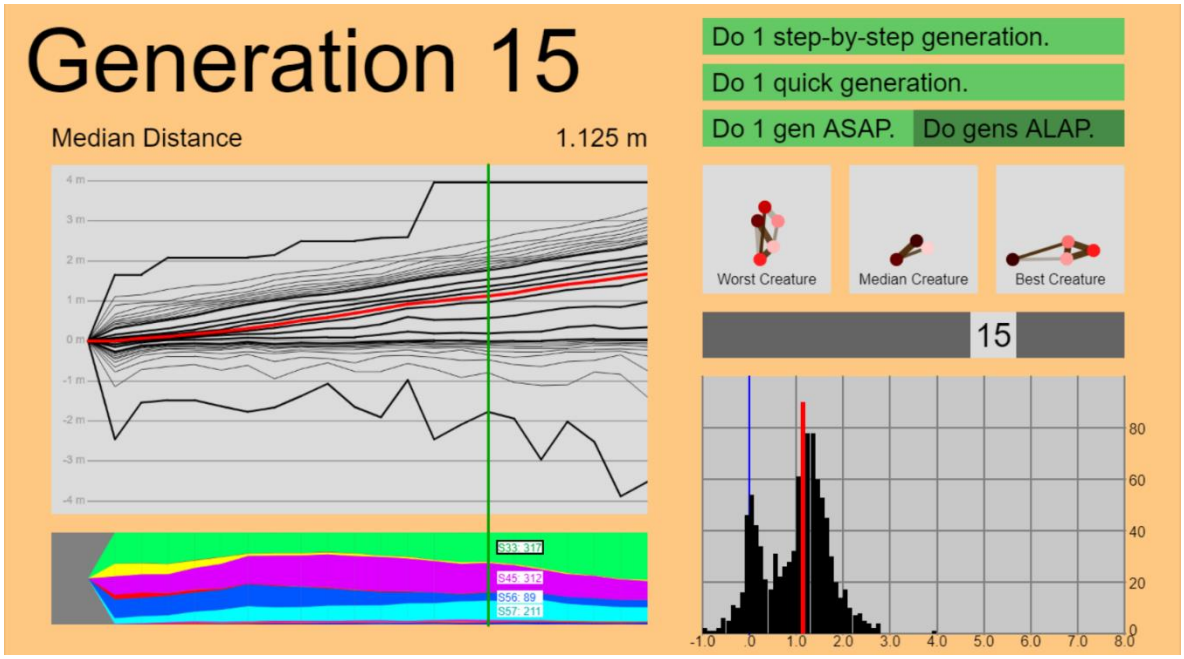
Parts that I can apply to my solution:

The representation of traits using a colour gradient (e.g. a slower creature is more green, and a faster create is more red) is a very effective, visual way of conveying how a population is

changing, so is something I would like to carry forward to my solution. The environment is also quite simple, and in each video the process of evolution is abstracted to only one or two traits for the sake of simplicity, which is also something I intend to implement. However, my solution will be visually simpler so that it can be displayed without putting a heavy load on the device's GPU – I intend my software to be lightweight enough that it could be run in real time on a phone.

Improved Evolution Simulator

Overview:



The "Improved Evolution Simulator" simulates the evolution of animal movement, abstracting an animal into only a set of nodes and 'muscles'. The goal of these creatures is to travel as far as possible in 15 seconds, and after each generation only the faster creatures are randomly slightly altered to produce the next generation. This is an example of reinforcement machine learning, and the difference between each generation is reflected in lots of different data plots: a histogram and line chart of fitness, as well as a summary of the different kinds of creature and the best, median, and worst creature is available for *every* generation, which the user can navigate through using a slider shown above on the middle right.

I found the user interface very easy to navigate and the instructions were very clear, and there was the option of looking at each creature as it evolved or evolving entire generations in a click to accelerate the process.

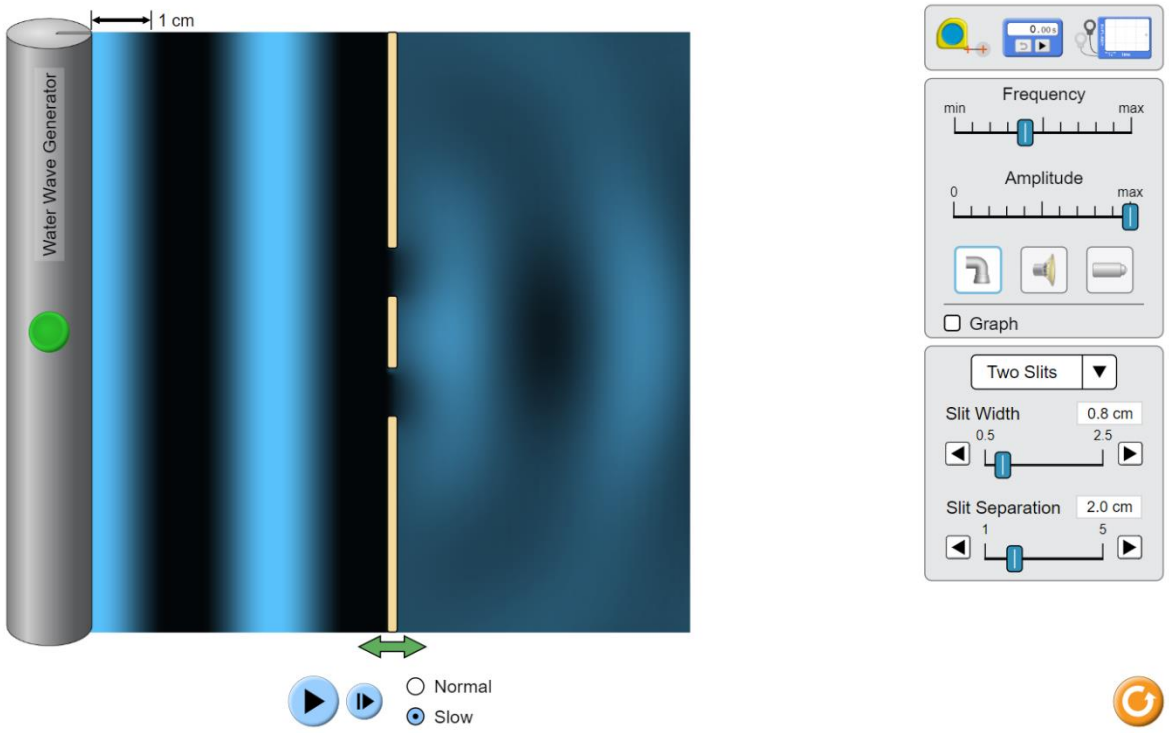
Parts that I can apply to my solution:

This program only simulates the evolution of one ability but does it in lots of detail. My solution will have more of an emphasis on the evolution of a variety of traits and will not simulate the evolution of physical features. However, a variety of data plots is a feature I

would like to include in my solution, as it shows at a glance the state of a population. Another very useful feature from this software is the slider for different generations, which allows for immediate analysis of how a population has changed over time, and so is a feature I might include in my solution. Moreover, the simplicity of the user interface makes it useable with no briefing or practise, which is definitely a feature I would like – my solution should be easy to use.

Wave Interference Simulation

Overview:



The University of Colorado created this computer visualisation of wave interference as a tool for teachers to demonstrate how waves behave when passing through a slit, or two slits. The user can change the number of slits the waves pass through, as well as every other relevant parameter such as slit width and separation, and the waves themselves. The controls are very simple, being either a slider, dropdown menu, or button, and the simulation changes in real time as the parameters change – there is no need to restart the visualisation to change the environment. These animations are frequently used by my own teachers to help with students' understanding.

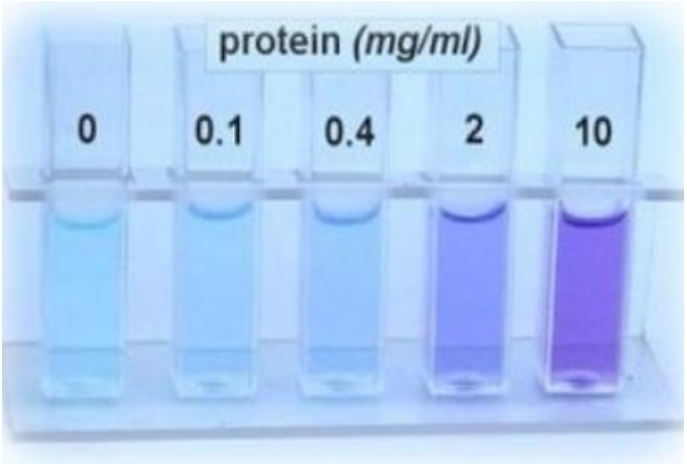
Parts that I can apply to my solution:

The simple controls – mostly sliders and dropdown menus – are perfect for a teaching tool like I'm trying to build, because they make the simulator very intuitive to use. The pictures used as buttons for the different kinds of wave (water, sound, and light) is another feature that makes the simulator easy to use, so in my solution I would like to use similarly intuitive controls that are clearly labelled, as they are here. Moreover, the simulation changing in real

time as the parameters are changed makes it immediately obvious what each parameter does, and has an immediate impact that I would like to carry over to my solution.

Biuret test (biology practical)

Overview:



The Biuret test is a biology practical that tests for the presence of proteins in a sample, in which the contents of a test tube change colour depending on the concentration of protein in the sample. It's intended to introduce biology students to science practicals, as it is easy to follow and has immediate, easily distinguishable results. It's also fast, so under time pressure a class can still perform this practical to understand how it works.

Parts that I can apply to my solution:

My software will not be a physical practical of this sort, but the colour coded results are easy to distinguish between and the practical is quick and easy to perform. Both of these traits – colour coded results and being fast to use – are traits I would like my software to exhibit, as they would make it more useful in a time-pressured environment such as a busy classroom. Simple instructions are particularly crucial if the students themselves can use the software, because otherwise it would take too long to get each student using it.

Features of the proposed solution

Initial concept of my solution considering this research

My solution will be an application that when started will bring up a menu of options, including an introduction to evolution and the evolution of several different traits, along with an "instructions" option to get familiar with the software. These are essential features because the tool is designed to be education and easy to use. On selecting a trait, an entirely new screen will appear, showing the initial conditions of an environment and sliders with which to change them, as well as a button to begin the simulation. This is an essential trait because the simulation being customisable is what makes this a useful learning tool. After the simulation starts, the software will iteratively change the population based on the conditions selected by the user and the laws of evolution. This is an essential feature because this is the process of simulation. Once the user stops the simulation, they will have the option to see the graphical results (a useful learning tool), or to return to the home menu.

Limitations of my solution

The main limitation of my software is that it can't be used by people with certain visual or physical disabilities. Blind people, and those with very poor eyesight, will not be able to interact with the software because it is designed to be very visual, and those with mobility and other physical disabilities will not be able to use the touchscreen that the software is designed to run on. In theory, some physical disabilities could be overcome using a different input method such as a footmouse, which would overcome the issue.

Another limitation of this solution is its complexity: it isn't a useful tool beyond the basics of evolution. Students learning the details of evolution as it works in reality couldn't use this software to help their learning, as the software will be specifically designed to ignore the finer details in favour of a high-level overview. They could use the software to secure their basic understanding, but any details of the biology or chemistry will not be covered.

Limitation	Justification
Visual impairment can render my software unusable	The software is by nature very visual, so blind people and those with poor eyesight will not be able to use the software. Also, colour blind people may not be able to tell apart the colours used to describe the traits
The controls are not designed to work with certain physical disabilities	The method of using the software – touchscreen – is not accessible to people with certain physical disabilities. A different input method such as a footmouse could overcome this problem
Complexity	The software is designed with GCSE and equivalent level students in mind. As such, the specific details of the biology and chemistry is out of scope, so the software will not be useful to those studying at A level and equivalent, or beyond
Flexibility	The program must run on a device with a screen size substantially larger than a smartphone because the software is very visual. Each student at my school was recently issued with an iPad, so I will design my software to run on an iPad. This is a major limitation because the majority of people do not own an iPad, and I do not have the resources to make it run on other devices

Requirements

Software and hardware requirements

Hardware

A computer capable of developing the software, with standard IO peripherals – The software will run and display a simulation, so for development the device must have a minimum processing power. However, this processing power is available in all modern-day devices, so this will not be a problem. To navigate the IDE, I will use standard IO peripherals including a mouse and keyboard.

iPad (6th generation), software version 13.4.1 – the iPad I will use to test my software on.

Software

macOS Catalina operating system – Because the software will be written in XCode, I will need access to the Mac operating system that supports XCode, which is macOS Catalina.

XCode 11 – The code will be written in Swift, so I will use XCode, an integrated development environment for developing software for Apple products. XCode 11 is the latest version of XCode.

Jump Desktop – Because I'm using a Windows device, I do not have direct access to XCode. I will therefore use Jump Desktop (remote desktop access software) to access a Mac with macOS Catalina.

Stakeholder requirements

Design

Requirement	Importance	Justification
Simple instructions	Desirable	If the software's instructions aren't easy to follow, it will not work as a learning tool – the pace of lessons are too quick to allow for wasted time
Simple and uncluttered user interface	Desirable	The Wave Interference Simulation has only as many variables as is needed to explore the topic fully. A cluttered interface would make it too complex to use in a classroom environment
Lightweight design	Desirable	The design needs to be simple to use and not confusing
An obvious button for instructions, and one for an overview of evolution if implemented	Nice to have	If the process doesn't make sense to the user, both instructions for the software and a high-level explanation of evolution should be available

Functionality features

Requirement	Importance	Justification
Simulate a changing population according to initial conditions and simple rules	Essential	This is the main function of the program, simulating a simplified version of evolution to convey how it works
Emphasise randomness of process	Desirable	Specifically requested as a feature by Ms Hicks – the randomness of the process is a key part of evolution
Have parameters for initial conditions the user can change in order to investigate impact	Desirable	Ms Hicks said the software would be "better" with customisable parameters, and this feature is what makes it a more effective learning tool than otherwise
Simulate classic examples of evolution such as peppered moths, or Darwin's finches	Nice to have	This was suggested as a possible target of simulation by Ms Hicks, as it would cover the examples traditionally taught in classrooms, so would complement traditional teaching well
A menu option for an overview of evolution	Nice to have	This would complement the simulation and make it a more effective learning tool, similar to the in-depth explanations in Primer's videos
Visual description of traits such as speed	Desirable	As in Primer's videos, a visual description of how a population is changing makes it immediately clear what has changed

Graphs near the simulation that describe the population at a glance	Essential	A vital part of Primer's videos is a graphical representation of the simulated population, and is a vital part of making evolution intuitive

Hardware and software

Development

Requirement	Justification
Standard peripherals: Computer with a keyboard, mouse, and monitor	In order to develop and test my software, I need access to a computer that can run the simulation and IDE as well as a keyboard, mouse, and monitor with which to navigate the computer
iPad (6th generation), software version 13.4.1	The iPad I will test my software on, because this is the iPad distributed to the students at my school
macOS Catalina operating system	Because the software will be written in XCode, I will need access to the Mac operating system that supports XCode, which is macOS Catalina
XCode 11	The code will be written in Swift, so I will use XCode, an integrated development environment for developing software for Apple products. XCode 11 is the latest version of XCode
Jump Desktop	Because I'm using a Windows device at home, I do not have direct access to XCode. I will therefore use Jump Desktop (remote desktop access software) to access a Mac with macOS Catalina

Deployment

Requirement	Explanation
iPad with iOS 12 or later	94.7% of all iPad users have adopted iOS 12 or later, so it isn't worth the extra time to develop the software for iOS versions earlier than that. Furthermore, the iPads distributed to the students at my school have iOS 13.4.1
At least 1GB of free storage	The software will be way under 1GB in size, but to ensure it will not cause any issues the user should have 1GB of free storage to locally store the software

Success Criteria

Reference	Criteria	Justification
1	A series of options to see the evolution of different traits	There should be a number of different traits available for study so that the user gets a rounded view of how evolution works
2	A clearly labelled option in the menu for instructions	If the user doesn't understand the purpose of the app or how it works, they need a clearly labelled 'instructions' button
3	An option in the menu for an explanation of evolution	This software is supposed to be a learning tool for evolution, so it would be helpful to that end to have an explanation of how evolution works built in

4	After the user clicks on a trait, show a popup describing the circumstances of each trait's development to provide context	To tie in the evolution of these virtual creatures to the real world, there should be an example of where this trait actually evolved to make the software more relevant
5	Emphasise randomness of process in 'explanation of evolution' page and on the popup after selecting a trait	Ms Hicks specifically requested that the randomness of the process should be highlighted to students who will be using the app, as that is often misunderstood
6	Be able to change at least one parameter of each simulation's environment	The software is designed to be interactive, so each trait's simulation must have an element of interactivity in the form of variable parameters
7	Traits described visually using colour in animation	Part of what makes Primer's videos effective is their use of colour to describe traits as they evolve, so I will too
8	Graphs visually near the simulation summarising the state of the population	Another key part of Primer's videos is at least one graph near the simulation simplifying the entire population into an empirical form
9	Uncluttered user interface	Lessons are fast paced, so the software must be uncluttered and easy to use so it doesn't take too much of a lesson to use. This is a key part of the 'Wave Interference Simulator'
10	Starting the simulation changes the population according to the selected parameters	This is the main feature of the software
11	Graph updates in real time as simulation runs	Instead of creating a graph after the simulation is finished, update the graph in real time to help the user understand how the population is changing
12	A clearly labelled button to stop the simulation at any time	In order to stop the simulation and return to the main menu, there must be a clearly labelled 'stop' button
13	Interactive post-simulation window summarising the evolution of the selected trait	The software would be more useful as a learning tool if there was an interactive summary of how the trait evolved, similar to the 'Improved Evolution Simulator' displayed after each new generation

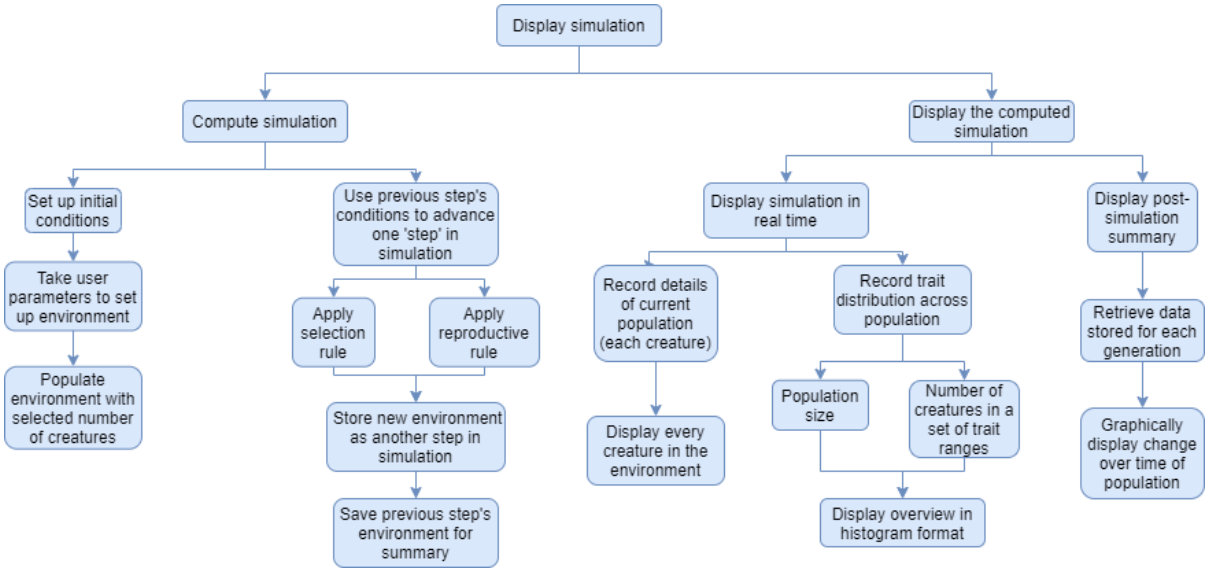
Design

Decomposition of Solution

Apart from the user interface parts of this solution, the main challenge is in the simulation technique. The overall objective of this is:

Take parameters of environment --> Display simulation as it progresses

This can be broken down into smaller problems that illustrate how this process will work:



The process can be largely separated into three sections, the setup of the environment, the 'natural' selection of each generation, and the displaying of the data produced. The setup only needs to be done once per simulation, but the selection and displaying of data must be done each 'step', or generation, of the simulation. Breaking this down into separate subroutines shows the following process:

Setup:

- Choose trait to select against
- Take user input for environment parameters
- Create environment and creatures

Computation:

- Apply selection pressure (remove creatures)
- Apply reproduction rules (generate slightly different population)
- Store current state in array of states so they can be accessed post-simulation

Display data:

- Display each creature of new population each generation
- Colour each creature to represent its trait (e.g. faster creature is redder)
- Create histogram describing the population
- After simulation is finished, display graph of average trait of each generation

Each step in more detail:

Setup

Choose trait to select against:

From the main menu, the user will select the evolution of a trait they're interested in, for instance camouflage. The trait in this case would be colour and is the example I will use for the rest of the problem decomposition.

Link to success criteria: 4

Take user input for environment parameters:

The user will be allowed to customise the environment and creatures. The user could (for instance) change the environment's colour, the reproduction rate of the creatures, the initial number of creatures, and the selection pressure (how well camouflaged the creatures must be). This allows the user to see for themselves the impact of changing the different parameters.

Link to success criteria: 7

Create environment and creatures:

Given the trait being selected against and the environment parameters, the population and environment can be created inside the program. This allows the population to be changed and the evolution to occur.

Link to success criteria: 11

Computation

Apply selection pressure:

Each generation, the first thing to do is kill off the creatures that are not well suited to their environment. The less similarly coloured they are to the environment, the more likely they are to be eaten – how aggressive this pressure is can be changed by the user. However, by random chance some poorly adapted creatures should live, and some well adapted creatures should die. This randomness is a crucial part of evolution and should be highlighted.

Link to success criteria: 11

Apply reproduction rules:

After some of the creatures are killed, the remaining creatures should have a chance to reproduce. Those creatures that do reproduce will create another creature similar to them, but slightly different, which represents the random mutation that characterises asexual reproduction. A new creature could, for instance, have a colour ±20 on each of the RGB inputs.

Link to success criteria: 11

Store current state in array of states:

The data summarising the new population should be stored in an array of similar states so they can be displayed post-simulation.

Link to success criteria: 14

Display data

Display each creature of each new generation:

After selection and reproduction, the new population should be displayed to the user so they can intuitively see the change in population size.

Link to success criteria: 11

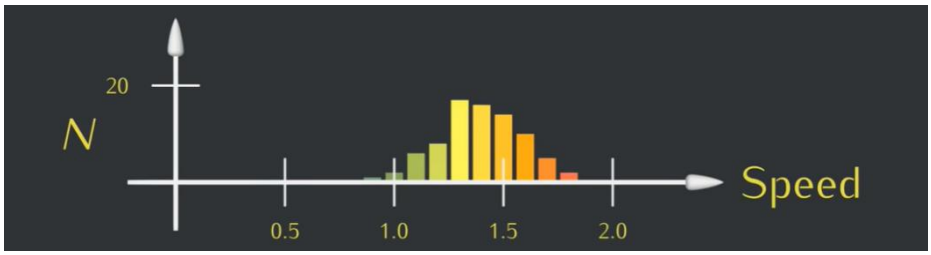
Colour each creature to represent its trait:

For each creature in each generation, colour the creature according to its trait: for instance, a slower creature is more green, and a faster creature is more red. This allows the user to immediately see the change in population after the selection and reproduction.

Link to success criteria: 8

Create histogram describing the population:

Each generation, the spread of traits should be summarised in a histogram like the one here:

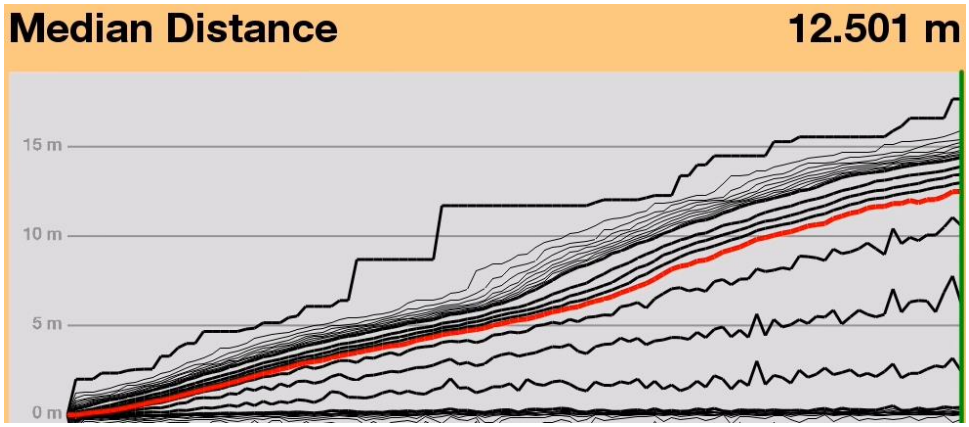


Note that the histogram itself is colour coded the same way as the creatures, to make the change to each generation even more intuitive. This histogram should be created and displayed every generation until the simulation is stopped.

Link to success criteria: 9, 12

Display graph of average trait of each generation after simulation is finished:

The change in the population over time can be summarised using a line graph of the average trait of each generation:



The red line represents the median speed of each generation over all previous generations and gives the user an immediate insight about the rate of evolution.

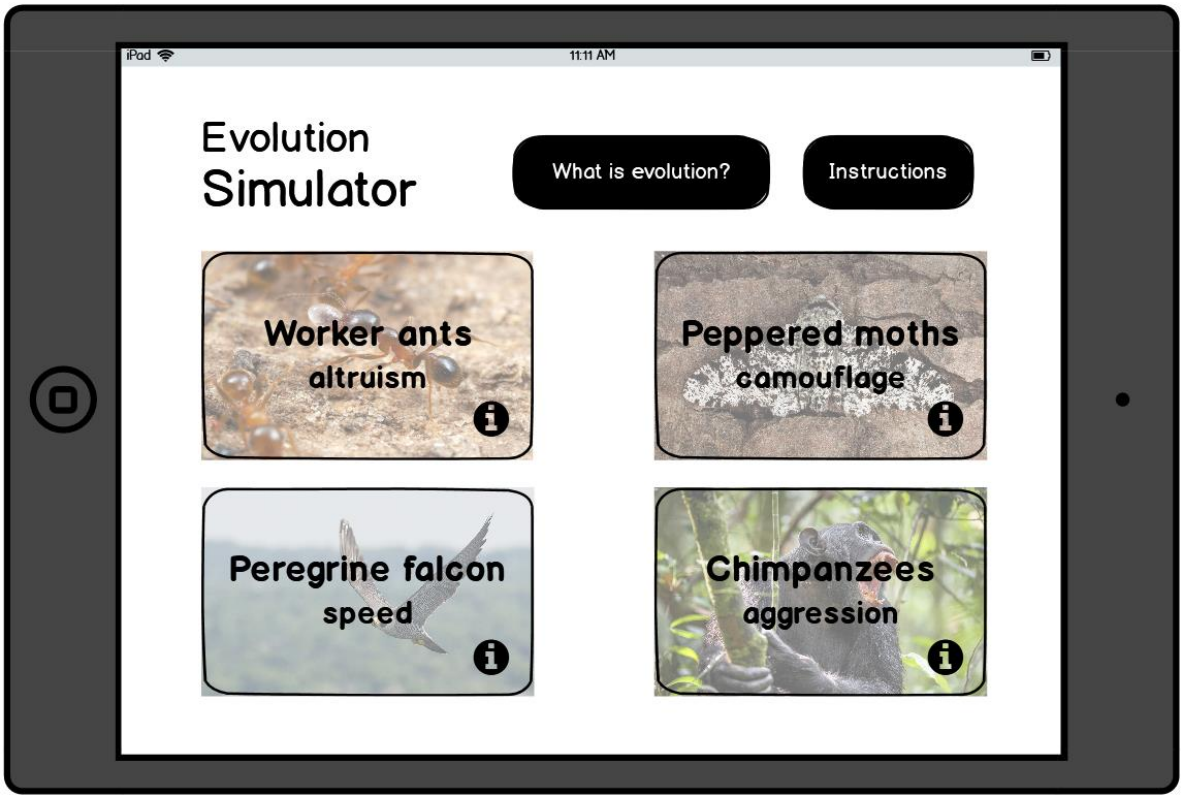
Link to success criteria: 14

Usability features

The usability features that I have considered make sure the program is easy to use for any type of user. All the buttons and sliders in the software will be reasonably large so that they're easy to see and use, which ensures people with visual impairments can still use the software. To that end, all the text in the program will also be large and easily legible, so people with visual impairments or who are using text-to-speech programs can understand it.

For people using screen readers, it is also important that all the text in the program is actually text, as opposed to an image containing text: this allows the text to be directly interpreted, rather than converted from an image to text.

Main menu



This wireframe shows the large buttons and clear text. Although the final version may have a different colour scheme, the text will remain clearly visible. This menu is linked to success criterion 1.

The program must be easy to use. Part of making the program easy to use is having a simple menu structure: if the app includes nested menus six deep, it is confusing at best and unusable at worst, so each menu will have its own button to avoid the possibility of the user getting lost in the menus.

The setup of each simulation has the possibility of becoming quite complicated. The instructions (which explain how the setup process works) must therefore be easy to access: as per success criterion 2, the instructions **must** be a single click from the main screen as demonstrated in the above wireframe.

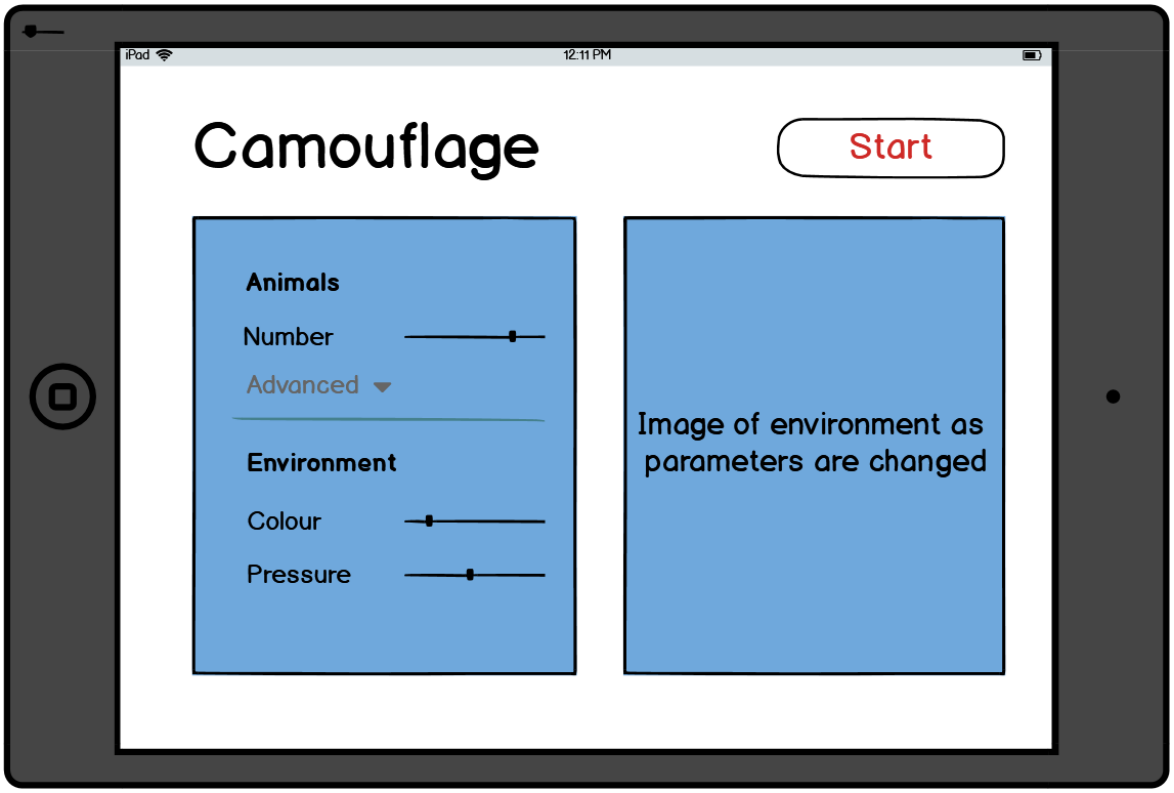
The option next to instructions for an explanation of evolution – also one click from the main menu for easy access – matches success criterion 3.

Success criterion 5 dictates that each trait must be linked to the real world to provide real-life context in which that trait evolved. To achieve this criterion, each simulation in the main menu has bold, easily legible text naming the animal in which the trait evolved. The trait itself is a sub-heading under the name of the animal, which is also bold to make it easily legible. This makes it very clear to the user what menu they are entering if they select that option, which helps to keep the menu structure intuitive for the user.

Furthermore, there is an information button in the bottom right of each simulation option:

Once clicked on, summary information on the creature and trait will appear, making it as clear as possible what each simulation will do. The “What is evolution?” button and the “Instructions” button have heavy, black designs to visually differentiate them from the simulations and to immediately draw attention to them on program launch – these should generally be the first buttons visited by the user.

Parameter input

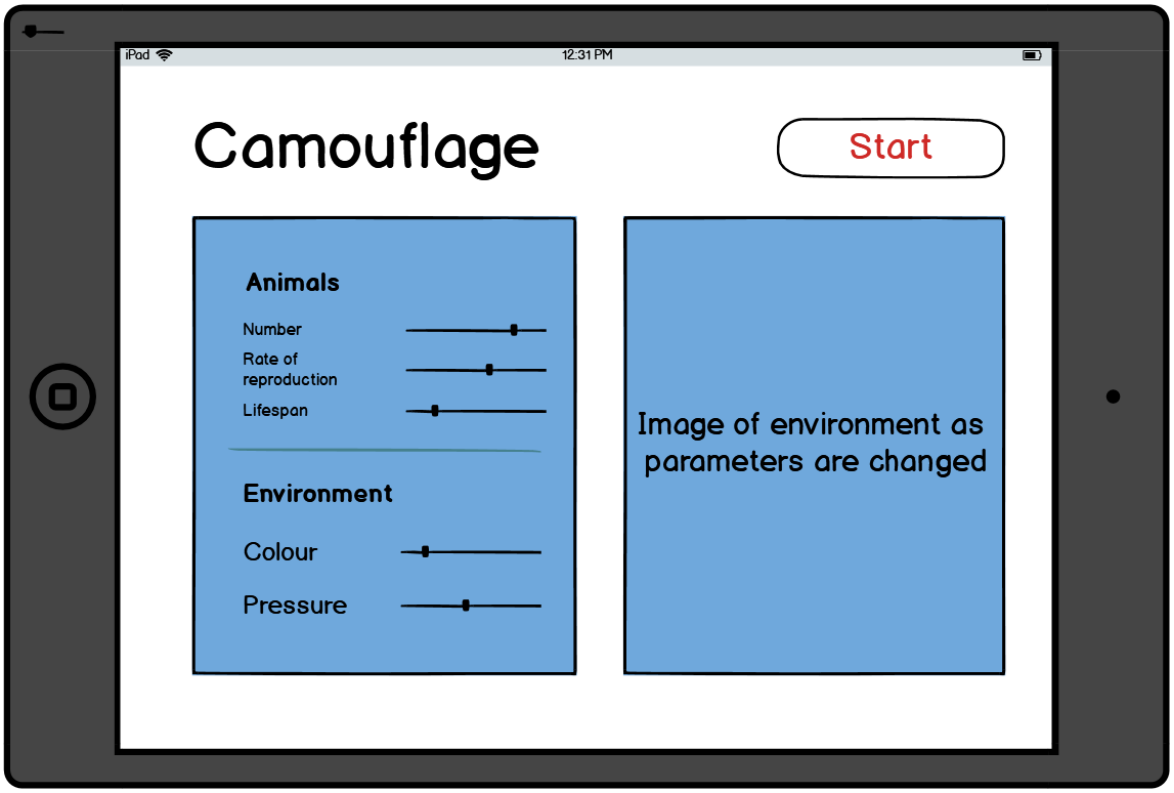


After selecting a trait, this page (or its equivalent for other traits) will open. The large title makes it clear what menu the user has entered, making the program easy to navigate. The “Start” button in the top right is coloured red because red has a cultural association, at least in the west, with a warning, and in this case suggests starting will lock in the parameters selected by the user. It is placed in the top right because buttons for continuing through a process should be placed on the right of the screen, similar to how the right page of an English book is what must be turned to get to the next page.

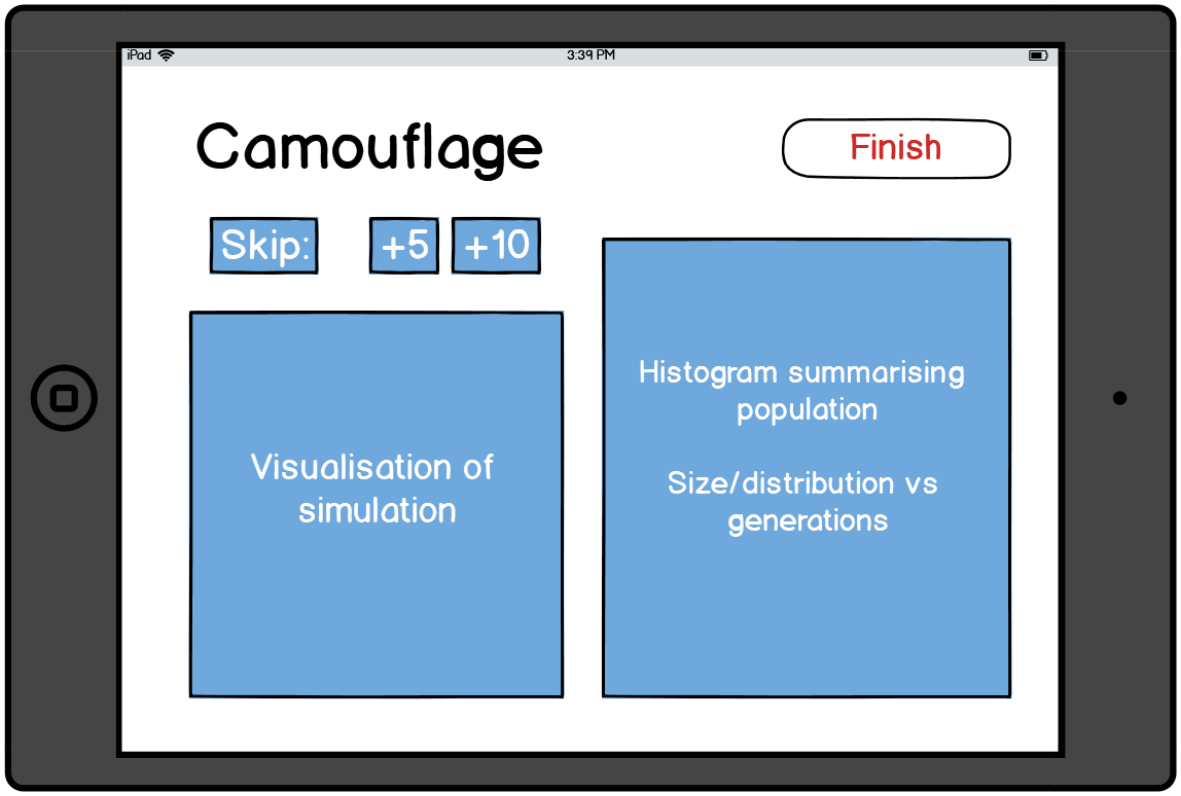
The sliders are used to select the parameters as a form of data validation: the user can only input valid data, given the data type and range are fixed. Using sliders ensures the user’s

experience of the program will be smooth, without error messages popping up. This option to customise parameters is linked to success criterion 7. When the page first opens, it will probably display a relatively limited number of options to keep the program simple for users, so that it's as easy to use as possible for users who don't care about extra options. The "Advanced" dropdown menu will display a new parameter input page with more options, but it's greyed out to start with to convey to the user it's optional, and not vital to the functionality of the program.

Upon clicking the dropdown, this menu will appear:



Simulation

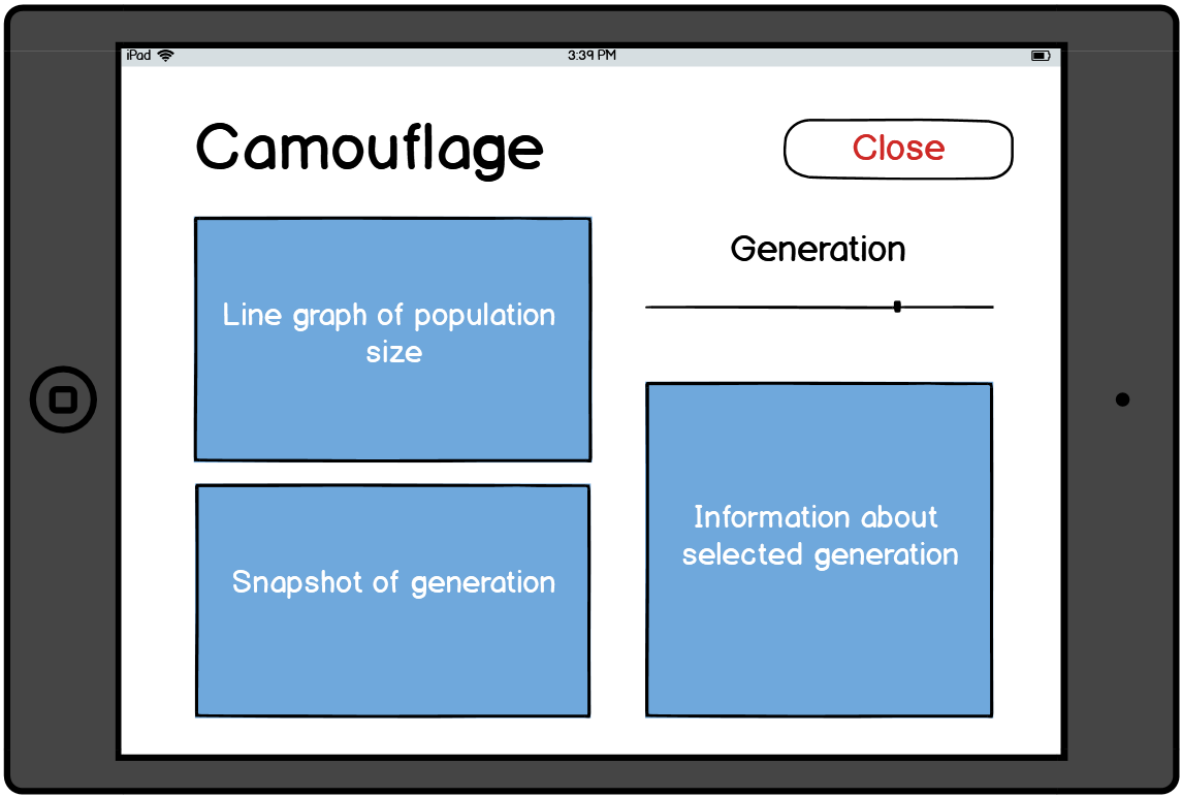


The trait being evolved is in a large font in the top left of the screen, where English-speaking readers (who read left --> right and top --> bottom) will generally look first. This makes it very clear to the user what menu they have just entered, making the menu navigation as easy as possible. If the user wants to leave the simulation at any time, there is a large "Finish" button in the top right, where a user's eye is drawn after the title. It is coloured red to indicate a destructive action as per Apple's Human Interface Guidelines. Red is used because it's associated with danger and is considered a warning colour in the western world. The clearly labelled "Finish" button meets success criterion 13 and makes navigating the software easier.

To enhance the user's experience of the simulation, there are two buttons to skip either 5 or 10 generations, allowing the user to see the evolution in less time, which also makes it more suitable for use in lessons in which time is limited.

The layout of the simulation page is as simple as I could make it without losing functionality, ensuring the user can use the software on first launch without much time spent getting familiar with it.

Post-simulation summary



Once again, the button that exits the simulation is coloured red to signify it as a destructive action. Also, the input (generation number selection) is a slider, forcing the user to select a valid data type and within a valid range. The purpose of the slider is clearly marked in large font. The use of a slider simplifies the use of the software for the user as they will never see an "Invalid input" screen.

Visual information about the simulation is collected on the left of the screen to keep the screen from becoming too cluttered, which is linked to success criterion 10. Summary information of the selected generation is displayed directly below the slider to create an intuitive link between the two, which clarifies the purpose of the information box without making it clunkily explicit, which also keeps the user interface uncluttered.

Stakeholder input

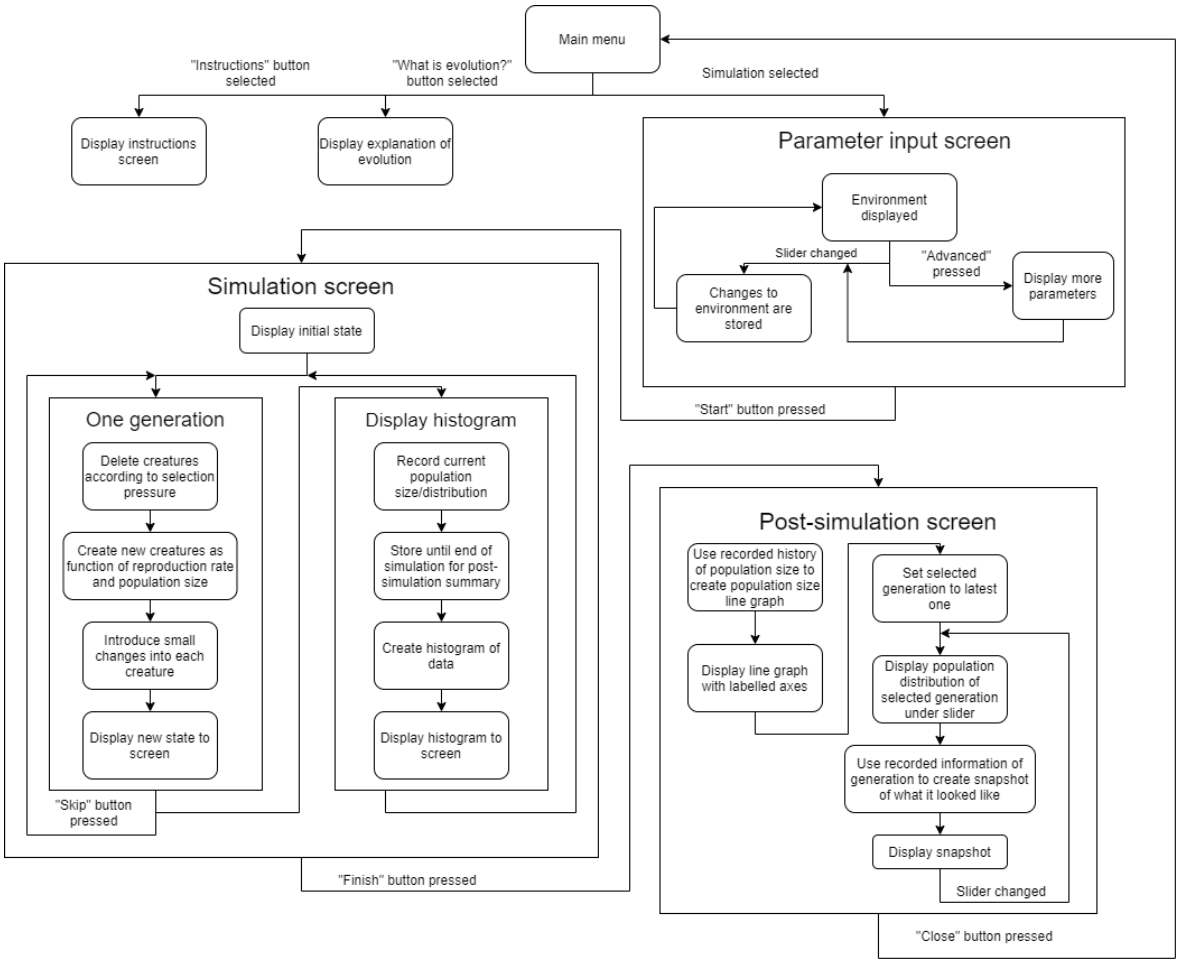
Having created my wireframes, I thought it would be useful to get input from my stakeholders and see if they had any immediate suggestions for how the wireframes could be improved, or if there were features missing that they were hoping would be there.

G.L. never responded.

Ms. Hicks simply responded with "Looks good Ben. I like the friendly format. Without seeing exactly how it works it's hard to comment more."

I will therefore ask her for input after I have created a minimum viable product.

Structure of the solution

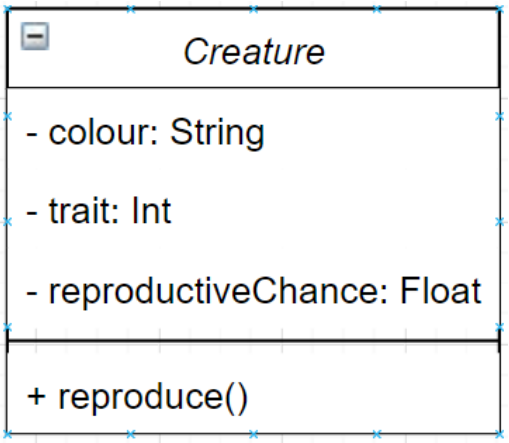


Key variables / data structures / classes

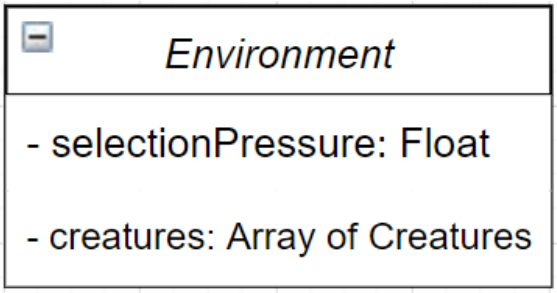
Name	Data type	Justification
populationHistory	An array of every generation's environment from the simulation.	This allows the post-simulation summary graphs to be created and displayed, relating to success criterion 14.
creatures	An array of Creatures that contains every creature from a given generation.	This allows the populationHistory array to be populated, the current generation's histogram to be made, and the next generation to be generated. Related to success criteria 11 and 12.
chartData	An array of just the "trait" integers from a given generation's "creatures".	Used to populate the BarChartData of each generation's histogram, related to success criterion 12.
selectionPressure	A float between 0 and 1 that defines how punishing an environment is, i.e. how likely a poorly adapted creature is to die.	Selection pressure is a key part of evolution in that it drives the process forward, dictating how fast it must happen. Linked to success criterion 11.

Key classes

Two key classes are the “Creature” class and the “Environment” class.



The



“Creature” class is necessary because the objects created from this class will populate the environment of each simulation. The “colour” attribute will be a string (hexadecimal) which will be used to colour each creature, which is a key part of the simulator: this is linked to success criterion 8. The “trait” attribute will describe how ‘much’ of a trait a creature has: for example, the speed of a creature would be stored as an integer in this attribute. This differentiates the creatures so that natural selection can remove unfit creatures. The “reproductiveChance” attribute can be dictated by the user in the “Advanced” parameters and will control the likelihood of any creature reproducing in a single generation. A chance of 1 means each creature will reproduce every generation, and a chance of 0.1 means 1 in 10 creatures will reproduce each generation, on average. The “reproduce” method will create another Creature in the “creatures” array in the “Environment” class.

The “Environment” class will be populated with creatures. To store each creature, the “Environment” class has the attribute “Creatures” to store an array of every creature currently in the environment, which must be updated each generation. This class also has the “selectionPressure” attribute which controls how punishing this environment is. A higher pressure means a poorly adapted creature is more likely to die each generation.

Necessary validation

My app will include no explicit validation of user inputs. This is because I have chosen to use sliders in my app instead of text input, and the format of a slider means data entered is always within a valid range. This decision was taken because text input provides no advantages over sliders here, and using sliders allows for cleaner code and automatic validation.

Algorithms

Environment creation

After the parameters have been set by the user and they press “Start”, the first instance of the environment must be created. The following function creates an instance of “Environment”, taking in the parameters as inputs and returning nothing. It fills the environments “creatures” array with the number of creatures specified by the user.

```
FUNCTION CreateEnvironment(numberOfCreatures, selectionPressure, reproductiveChance):
```

```
        let environment = Environment()
        environment.selectionPressure = selectionPressure
        environment.creatures = []
        FOR i = 1 TO numberOfCreatures:
            environment.creatures.append(createCreature())
        END FOR
END FUNCTION
```

Creature creation

The "CreateEnvironment" function calls "createCreature" as many times as the number of creatures specified by the user. This function must therefore create a creature with a random trait value, a reproductive chance specified by the user, and a colour based on the trait value. It must then return this creature.

```
FUNCTION createCreature():
        let creature = Creature()
        creature.traitValue = random.randint(1, 100)
        creature.colour = Int(creature.traitValue * 2.55) # Converts trait value to single range of red, green, or blue values
        RETURN creature
END FUNCTION
```

Stepping forward one generation

Each generation of the simulation, some creatures must be killed and the surviving creatures must each have a chance to reproduce. The function "createNewGeneration" therefore loops through each creature in the environment, killing those creatures less well suited and allowing the surviving creatures a chance to reproduce. It takes no parameters, and returns nothing, but updates the simulation display after the new generation has been created.

```
FUNCTION createNewGeneration():
        FOR creature IN environment.creatures:
            adaptationMetric = MOD(creature.traitValue - environment.traitValue) / environment.traitValue # Higher number means creature matches environment less well
            IF adaptationMetric > selectionPressure - 1:
                REMOVE creature FROM environment.creatures # This is a simplified version of how creatures will actually be 'killed'
            ELSE:
                reproductionNumber = random.random(0, 1)
```

```
                        IF reproductionNumber < creature.reproductiveChance:
                                environment.creatures.append(creature.reproduce(creature.traitValue))
                        END IF
                END IF
        END FOR
        populationHistory.append(environment)
        UpdateSimulation(environment)
END FUNCTION
```

Creature reproduction

Creature reproduction is different to creature creation. The key difference is that creature reproduction means the child creature is similar to the parent creature instead of being randomly generated. The "reproduce" function therefore takes the parent's trait value as a parameter and returns the child creature.

```
FUNCTION reproduce(traitValue):
        let creature = Creature()
        traitDifference = random.randint(-20, 20)
        creature.traitValue = traitValue + traitDifference
        creature.colour = Int(creature.traitValue * 2.55)
        RETURN creature
END FUNCTION
```

After completing the above steps, the simulation has been set up and can advance one generation – these are the fundamental parts of the complete solution. The variables needed for the other parts of the solution (such as creating the various graphs) have also been included so that incorporating them into the overall solution will be very simple. The other parts of the solution cannot be written as pseudocode algorithms here because the complexity of those parts is as much in the detail of their implementation in Swift as their algorithmic complexity, and because I don't yet *know* exactly which algorithms I will need. More algorithms I need will become clear as I develop the solution.

Test data – iterative development

Given the only user inputs in this program are buttons and sliders, data entered can only be normal and boundary data. As such, setting the sliders to the minimum and maximum – and thereby testing the smallest and largest simulations – is the most strenuous possible test.

Smallest possible simulation

Data field (sliders)	Value	Justification
Number of creatures	Minimum: 1	The minimum number of creatures in an environment can start with is 1. If this creature dies in the first round, the simulation immediately ends, testing how well the program copes with a single generation
Rate of reproduction	Minimum: 0.1	In trying to kill the population in the first generation update, a minimum rate of reproduction is ideal.
Selection pressure	Maximum: 0.95	This gives a randomly generated creature a 95% chance of dying, and will test how well the program copes with the immediate death of the entire population.
Environment colour	Minimum: 0	The lowest chance of a randomly generated creature being near to the colour of the environment (and surviving) is if the environment is pure white or pure black, so to kill the population as fast as possible I have set the environment colour to black.

Largest possible population

Data field (sliders)	Value	Justification
Number of creatures	Maximum: 20	When testing how the program copes with huge population sizes, I will test the case in which the user enters the maximum possible population size.
Rate of reproduction	Maximum: 1	In trying to test the upper capacity of the program, I will test the simulation when the creatures reproduce as fast as possible. A value of 1 roughly translates to each creature reproducing once each generation.
Selection pressure	Minimum: 0.05	We want the least possible creatures to be naturally selected from the population each generation.
Environment colour	Mean: 127	Given the creatures are randomly generated, we want the environment colour to be the mean creature colour in order to keep as many of them alive as possible.

UI tests

The UI tests I will implement will mostly include navigating the menus (simply checking the buttons work and recording their user) and creating simulations using boundary data to make sure nothing breaks. I will document what I am testing and the inputs for the test, as well as the result of each test during development.

Test data – post development

The iterative nature of the Agile methodology means that the test data I use in post development testing entirely depends on the direction my app takes in response to my

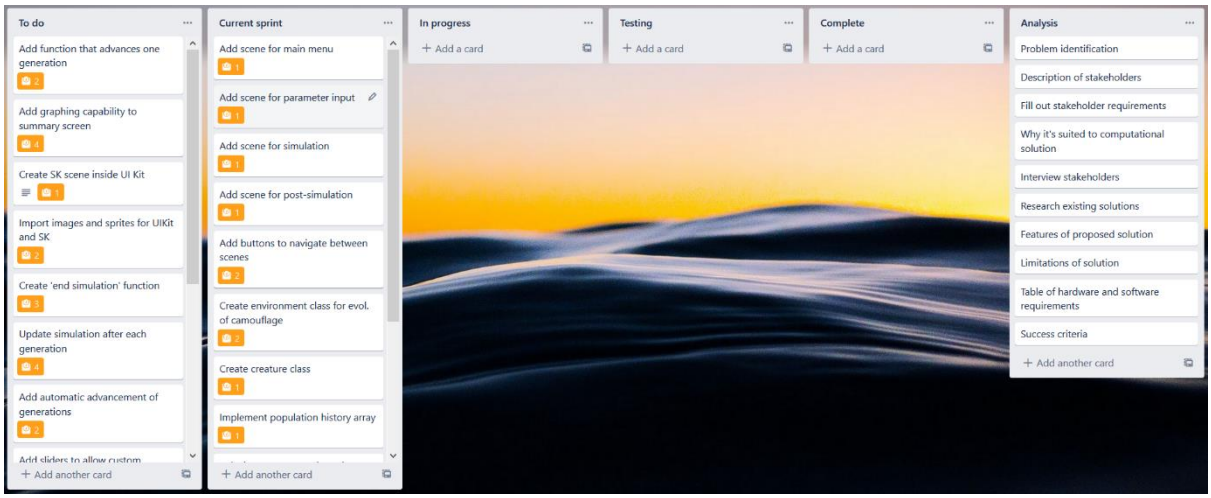
stakeholder feedback. It is therefore impossible to know exactly what data I will use in post development testing, but the tests (and test data) I use will be carefully documented at the time.

That being said, the above tests will all apply in post development testing.

Development

Introduction

During development of the product, I will use an Agile methodology that splits the 10 weeks of development into 5 two-week 'sprints'. At the start of each of the 5 sprints, I will select the tasks I will aim to complete over the course of the sprint from the backlog of tasks I created. Once every task in the backlog is complete, I will have finished my product. The backlog of tasks, including the tasks already completed during analysis and design, is shown below:

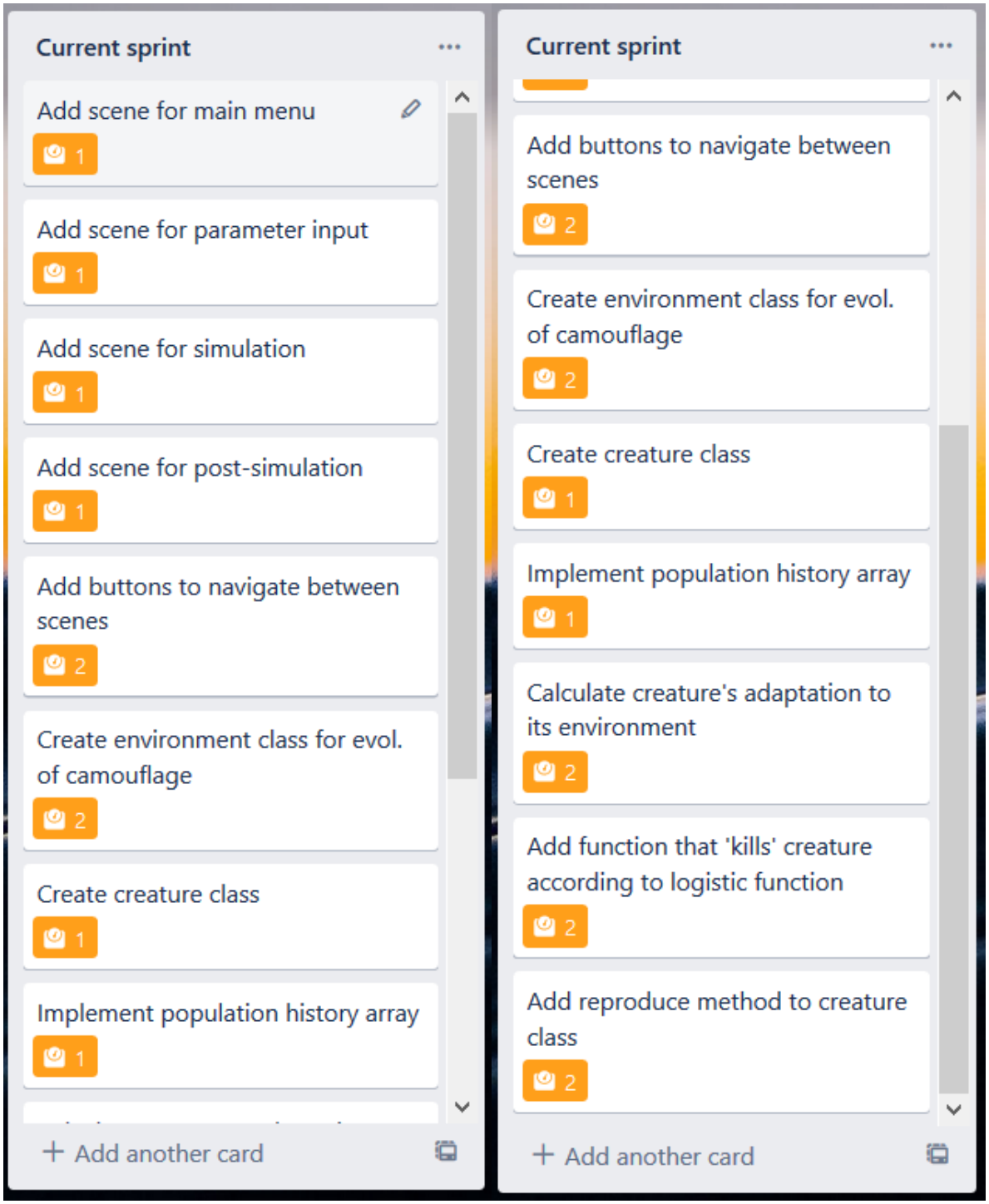


Because I'm developing my product iteratively, it is important that after building each class and function I test the code works. I will do this by building automated unit tests which I will document, and use to improve my product. If a test fails, I will go back to the code I wrote, attempt to fix the bug, then run the test again until the code works as intended. For those features that cannot be tested using automated unit tests, I will provide test data and results inside the ticket.

Not every task in my backlog is displayed in the image above, but I will display the tasks selected for the relevant sprint at the start of each one. If I finish every task chosen for a sprint, I will select more tasks to add to the sprint. If I do not finish every task chosen for a sprint, I will carry over the unfinished tasks to the next sprint and prioritise them above the other tasks. As development continues, some tasks may be split into several smaller ones once it is clear to me how best to solve a particular task. For instance, I do not yet know the best way of graphing in Swift, but once I do I will split the 'graphing capability' task into several smaller, more detailed ones.

Sprint 1

Below, the tasks selected for the first sprint are displayed. The numbers in yellow are estimates of the difficulty of each task, allowing me to better manage my time. The precise amount of time needed to complete 1 'difficulty point' is hard to estimate; instead, the numbers serve as a guide rather than an exact value.



Creating creature and environment class

The two most fundamental classes are the ‘creature’ and ‘environment’ class, so I will begin by creating then testing them.

S1 T1: Creature class

I created the creature class first. This ticket is linked to success criteria 7 and 10. For the evolution of camouflage, the trait of each creature is colour, so the 'traitValue' of each creature represents its colour. When the creatures are first created, each creature has a random traitValue, but a creature produced by reproduction will have a traitValue similar to its parent – this represents the genetic similarity between parent and child. In other words, the 'reproduce' function should initialise another instance of the Creature class with a traitValue close to that of the Creature on which the function was called.

UML class diagram and pseudocode

Class diagram: see page 24.

Pseudocode: see page 25-26.

Code

```swift
//
//  Creature.swift
//  Evolution simulator
//
//  Created by Vlasto, Benedict (JDN) on 19/09/2020.
//

import Foundation

class Creature {

    // Trait value is an integer between 1 and 100 that represents how much of the trait being evolved a creature has, such as how fast/slow a creature is.
    var traitValue: Int

    init(traitValue: Int?) {
        if let creatureTraitValue = traitValue {
            self.traitValue = creatureTraitValue
        } else {
            self.traitValue = Int.random(in: 1...100)
        }
    }

    func reproduce() -> Creature {

        let mutationConstant = Int.random(in: -10...10)
        let childTraitValue = self.traitValue + mutationConstant
        let newCreature = Creature(traitValue: childTraitValue)
        return newCreature

    }

}

```

Test

```
34
35          // Arrange
36          let creature = Creature(traitValue: nil)
37
38          // Act
39          let newCreature = creature.reproduce()
40          let expected = 11
41          var actual = creature.traitValue - newCreature.traitValue
42          if actual < 0 {
43              actual = -actual
44          }
45
46          // Assert
47          XCTAssertLessThan(actual, expected)
48
49      }
50
51  }
```

The reproduce function and the initialiser can be tested in one unit test, as the reproduce function calls the initialiser. The test above creates a creature, calls the reproduce function on that creature, then checks whether the traitValue of the child is within 10 of the parent's. The green tick in the top left shows that the test passed. This test passed first time.

It was now I realised I'd made a mistake. Although the test passed, I'd forgotten to test whether the traitValue of a child is within the acceptable bounds of 1 to 100. I wrote another unit test specifically to test this.

Test whether child's traitValue is within acceptable bounds

```
57
58      for _ in 1...5 {
59
60          // Arrange
61          let creature = Creature(traitValue: 1)
62
63          // Act
64          let newCreature = creature.reproduce()
65
66          // Assert
67          XCTAssertGreaterThan(newCreature.traitValue, 0)        4  XCTAssertGreaterThan failed: ("-2") is not greater than ("0")
68
69      }
70
71  }
```

The test failed, as shown by the red X in the top left and the error message, which tells me *why* the test failed. In this case, the child's traitValue is less than zero, which is outside the acceptable bounds.

I therefore revised my Creature class so that when the reproduce function is called on a creature, the child's traitValue will definitely be within 1 and 100 inclusive. The logic I implemented to ensure this calculates the 'overflow' over 100 or under 1, and moves that far back into the acceptable range. If a creature's traitValue is 110, it will be set to 100 - (110 – 100) = 90. If a creature's traitValue is –4, it will be set to 1 + (1 - (-4)) = 6.

Updated code

```
if childTraitValue > 100 {
    childTraitValue = 200 - (mutationVariable + self.traitValue)
} else if childTraitValue < 1 {
    childTraitValue = 2 - (mutationVariable + self.traitValue)
}
```

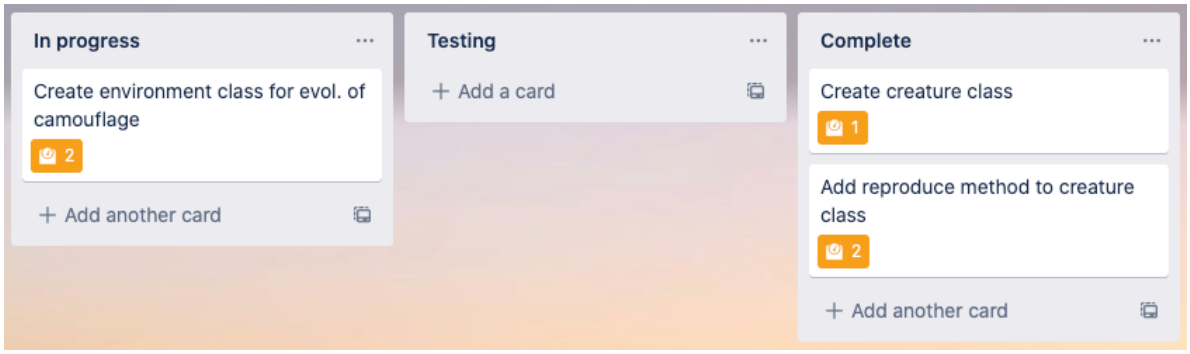
Updated tests

```
57      func testChildTraitValueLessThanOneIsHandledCorrectly() {
58
59          for _ in 1...5 {
60
61              // Arrange
62              let creature = Creature(traitValue: 1)
63
64              // Act
65              let newCreature = creature.reproduce()
66
67              // Assert
68              XCTAssertGreaterThan(newCreature.traitValue, 0)
69
70          }
71
72      }
73
74      func testChildTraitValueGreaterThanOneHundredIsHandledCorrectly() {
75
76          for _ in 1...5 {
77
78              // Arrange
79              let creature = Creature(traitValue: 100)
80
81              // Act
82              let newCreature = creature.reproduce()
83
84              // Assert
85              XCTAssertLessThan(newCreature.traitValue, 101)
86
87          }
88
      }
```

I wrote two different tests to test numbers both above and below the range. Each test passed first time after implementing the new logic. This tells me the logic works, and I can therefore move on to the development of the environment class.

S1 T2: Environment class

Having written the required tests of the creature class and had the tests pass, I can move the relevant ticket to 'Complete'. I also implemented and tested the 'reproduce' method inside the creature class, so I can move that ticket to 'Complete' too. Now, I can add the environment class ticket to 'In progress'. This ticket is working towards success criterion 10.



The environment class will store the population of creatures in a simulation in one array called 'creatures', which will be an array of Creatures. This class will also store values that are constant throughout the simulation, such as the chance a creature reproduces each generation, and the pressure against a creature that is poorly adapted to its environment. To start with, I will only write, then test, the initialiser of the class.

UML class diagram

Class diagram: see page 24.

Pseudocode: see pages 24-25.

Code

```swift
import Foundation

class Environment {

    // Reproductive chance is a float between 0 and 1 that represents the
        chance a creature will reproducee each generation. It is a constant
        per simulation.
    var reproductiveChance: Float
    var selectionPressure: Float
    var creatures: [Creature]

    init(reproductiveChance: Float, selectionPressure: Float,
        populationSize: Int) {
        self.reproductiveChance = reproductiveChance
        self.selectionPressure = selectionPressure
        self.creatures = []
        for _ in 1...populationSize {
            let creature = Creature(traitValue: nil)
            self.creatures.append(creature)
        }
    }

}
```

Test

```swift
    func testEnvironmentClassInitialiserCreatesCorrectNumberOfCreatures() {

        // Arrange
        let environment = Environment(reproductiveChance: 0.5, selectionPressure: 0.1, populationSize: 20)

        // Act
        let expected = 20
        let actual = environment.creatures.count

        // Assert
        XCTAssertEqual(actual, expected)

    }

}
```

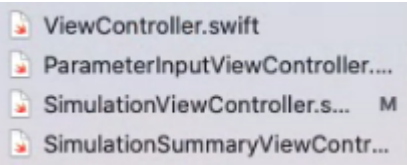
This test ensures that an environment initialised with 20 creatures correctly creates an array of 20 creatures. This test passed first time, which tells me the initialiser works as intended. The ‘create environment class’ ticket can be moved to complete, so I will start work on the UI.

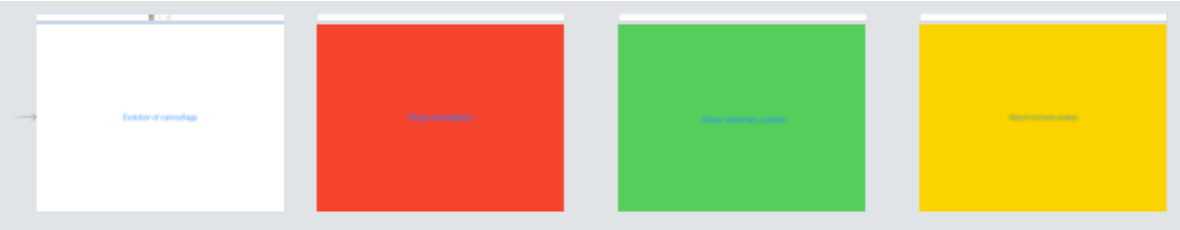
S1 T3-6: Adding scenes

As I am in only the first sprint, I will create the most basic possible UI on which I can build my app. This is related to all of my success criteria because this is the structure on which the other features will be built. The first things I need to create in the UI are the different ‘view controllers’ so that the main menu, the parameter input screen, the simulation, and the summary screen are all controlled by different classes (of type UIViewController). Using separate classes in this way makes the code **much** cleaner, as it is easier to distinguish between the different aspects of the UI such as the many buttons and sliders I will use.

Creating the different scenes was relatively easy as XCode has inbuilt functionality to add scenes. I added four scenes for the different sections of the simulation (main menu, parameter input, simulation, and summary), and colour coded them so the difference was clear. I used the default "ViewController.swift" file for the main menu.

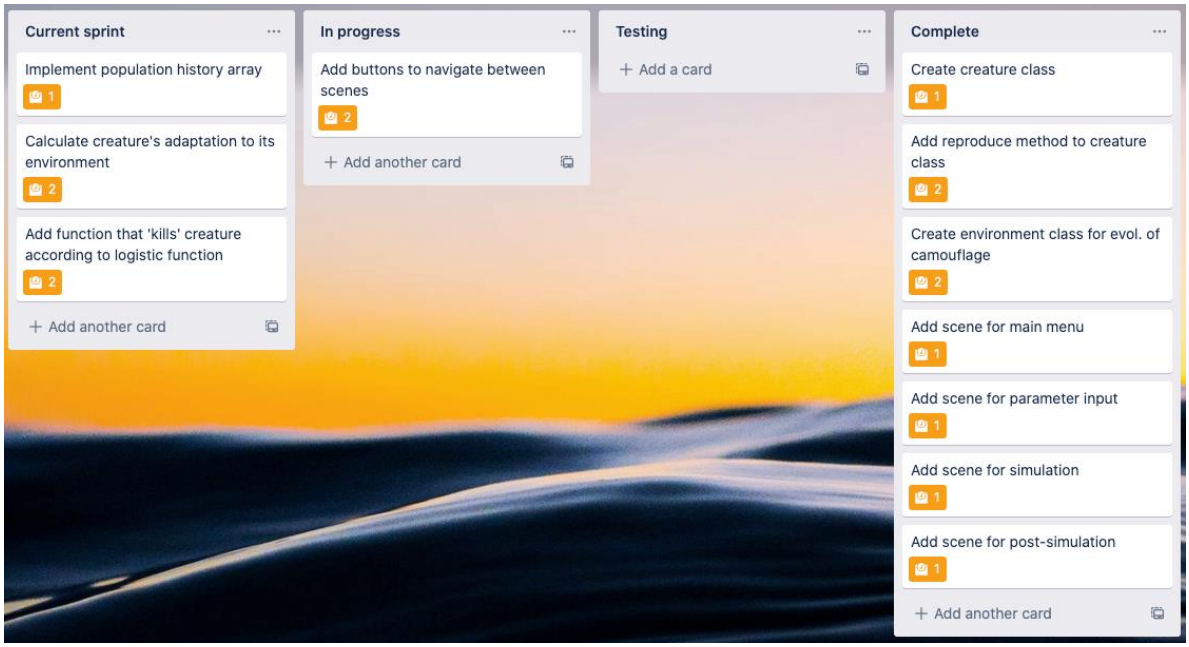
Code and UI





This cannot usefully be tested using automated UI tests when simply running the app is as effective, so I moved the 'create scenes' tickets directly into 'Completed'.

S1 T7: Adding buttons to navigate app



I decided to add buttons to transition between scenes now (relatively early on) because the user must be able to navigate the app for it to be remotely functional. This absolutely fundamental functionality means this ticket is related to almost all of the success criteria. Creating the buttons to navigate between these scenes should have been easy, but as often happens with programming it took far longer than expected. It should simply have been a case of adding a button to each scene using

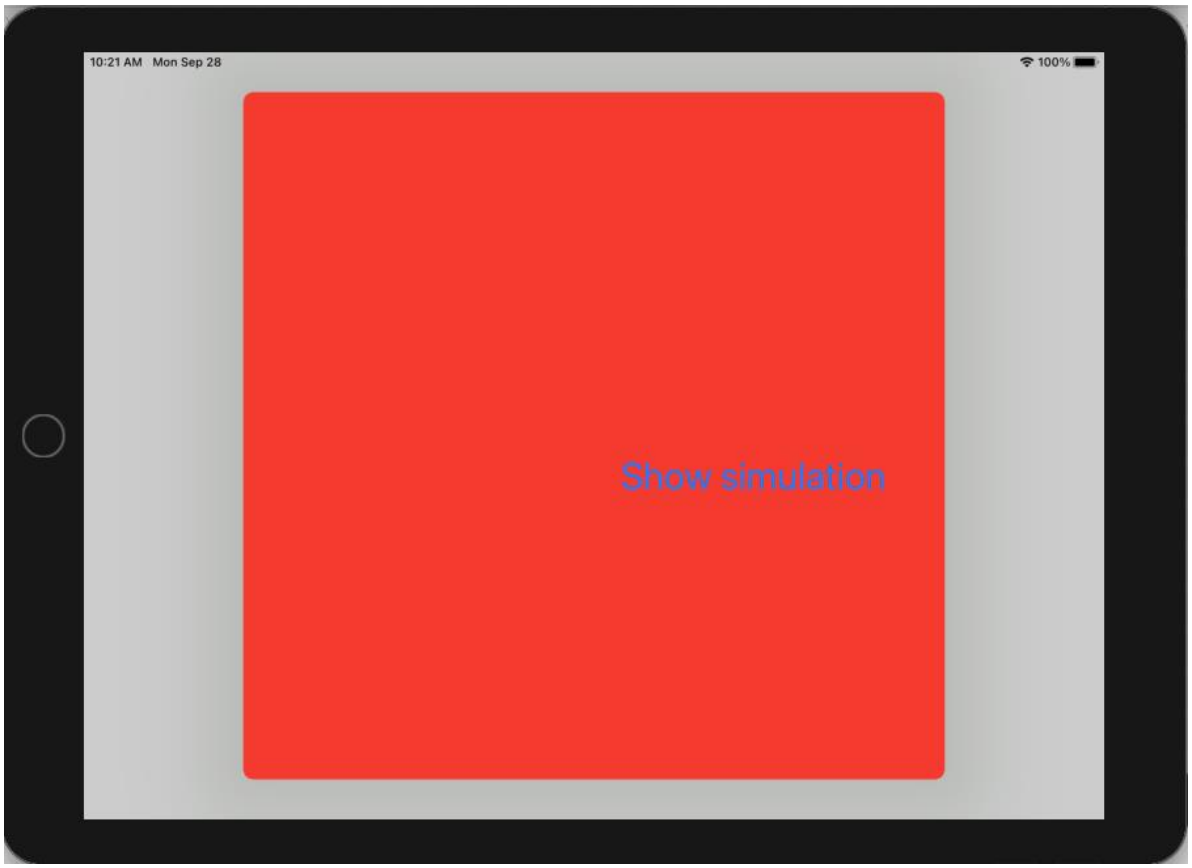
XCode's inbuilt UI editor, linking each button to the relevant class, then adding code to present the next scene when the button is pressed.

Code

```
@IBAction func showParameterInputVC(_ sender: Any) {
    if let vc = storyboard?.instantiateViewController(identifier: "parameterInputVC") as? ParameterInputViewController {
        present(vc, animated: true, completion: nil)
    }
}
```

Test

I manually tested the app by running it in XCode's device simulator, but the next scene opened in a popup window, like this:



I spent half an hour researching this, then worked out I had to specify I wanted the new scene to present itself as full screen, like this:

```
vc.modalPresentationStyle = .fullScreen
present(vc, animated: true, completion: nil)
```

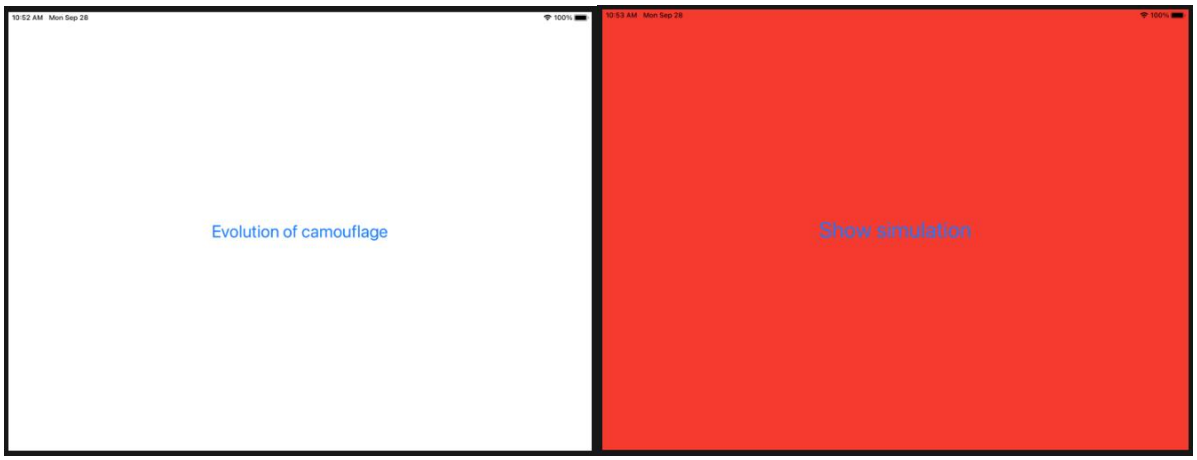
Now the new scene presented itself as expected, and I could navigate from one scene to the next, then back to the main menu from the simulation summary screen. These buttons *can* be tested using automated UI tests.

Code

```
@IBAction func showSimulationVC(_ sender: Any) {
    if let vc = storyboard?.instantiateViewController(identifier: "simulationVC") as?
        SimulationViewController {
        vc.modalPresentationStyle = .fullScreen
        present(vc, animated: true, completion: nil)
    }
}
```

Test

I ran the app myself to ensure the buttons and scenes present to the user as expected, as this sort of extremely simple test does not lend itself to automated testing.



The different scenes have brightly coloured backgrounds to clearly differentiate them during development, but I will change these colours later so the app is useable.

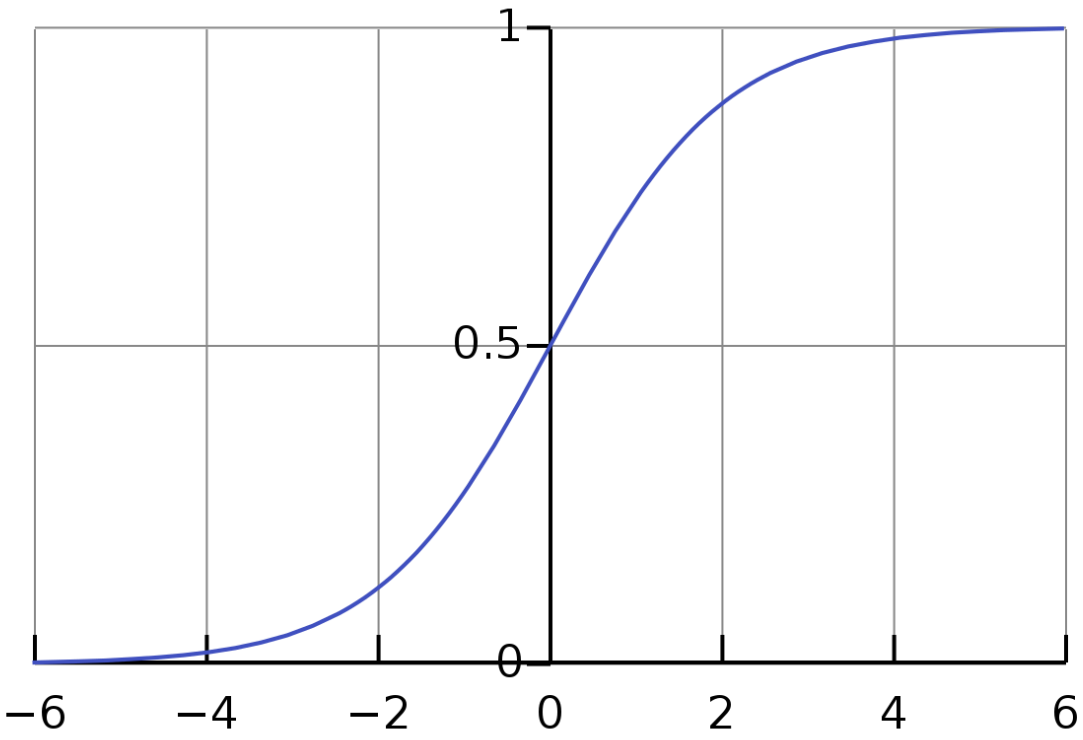
As the buttons to navigate between the scenes work as expected, I will add this ticket to the 'Completed' list and begin work on calculating a creature's adaptation to its environment.

S1 T8: Calculate creature's adaptation

For this app to work, I must be able to calculate how likely a creature is to die each generation as per success criterion 10 in my analysis section. In the case of camouflage, a creature with a traitValue (i.e.

colour) close to that of the environment is less likely to die than a creature with a very different traitValue. However, to incorporate the random aspect of evolution – that a poorly adapted creature can live while a well-adapted creature can die – there must be a non-zero probability of survival for each creature, each generation.

This probability function will look something like a logistic function, like the one shown here:



The y-value, which can only take values between 0 and 1, will represent the probability of living. The x-value, which ranges from -∞ to +∞, will represent the creature's adaptation to its environment. In this case, a creature's adaptation to its environment can only range between 100 (perfect adaptation) and 0 (worst possible adaptation), so I will have to scale the logistic function to return a useful chance of death. Its adaptation is easy to calculate: 100 – MOD(environment.traitValue - creature.traitValue). I added this function to the environment class, as the environment's traitValue is easily accessible within the environment class itself.

Pseudocode:

See pages 25-26.

Code

```
func calculateCreaturesAdaptation(creature: Creature) -> Int {
    var difference = creature.traitValue - self.environmentColour
    if difference < 0 {
        difference = -difference
    }
    let adaptation = 100 - difference
    return adaptation
}
```

Test

```
105
106     // Arrange
107     let environment = Environment(environmentColour: 50, reproductiveChance: 0.5, selectionPressure: 0.1,
            populationSize: 20)
108
109     // Act
110     for creature in environment.creatures {
111         let actual = environment.calculateCreaturesAdaptation(creature: creature)
112         var expected = environment.environmentColour - creature.traitValue
113         if expected < 0 {
114             expected = -expected
115         }
116         XCTAssertEqual(actual, expected)        17  XCTAssertEqual failed: ("54") is not equal to ("46")
117     }
118
119 }
```

This test failed, because the value I expected to get (“46”) did not equal the value I got from the function I wrote (“54”). However, I realised this was not a problem with the function, but with the test: the test did not subtract the number from 100.

Corrected test

```
105
106     // Arrange
107     let environment = Environment(environmentColour: 50, reproductiveChance: 0.5, selectionPressure: 0.1,
            populationSize: 20)
108
109     // Act
110     for creature in environment.creatures {
111         let actual = environment.calculateCreaturesAdaptation(creature: creature)
112         var expected = environment.environmentColour - creature.traitValue
113         if expected < 0 {
114             expected = -expected
115         }
116         expected = 100 - expected
117         XCTAssertEqual(actual, expected)
118     }
119
120 }
```

After correcting the test, it passed first time. This means the function worked as expected, so I can move the relevant ticket to “Completed”.

End of sprint review

Having spent two weeks on the project, I have finished my first sprint. I had completed most of the tickets in my sprint backlog, but I didn’t manage to implement the function that kills a creature. I also didn’t manage to implement the population history array, but it isn’t possible to implement this array

until later tickets are completed such as the ‘Advance one generation’ ticket. I have therefore moved the kill function to the top of the next sprint’s backlog, and the population history array later in the sprint.

I didn’t complete all the tickets in my backlog, but instead of selecting fewer tickets in the next sprint I will dedicate more time to the project, as otherwise I will not make fast enough progress.

Stakeholder feedback

Questions:

- What did you like about the app?
- What would you like to see changed?
- What should I focus on moving forward?

G.L.: The app is still in very early stages of development which makes sense. The user interface doesn’t do anything yet so I can’t tell you what you should change, but I can answer definitely these questions when the app has more features.

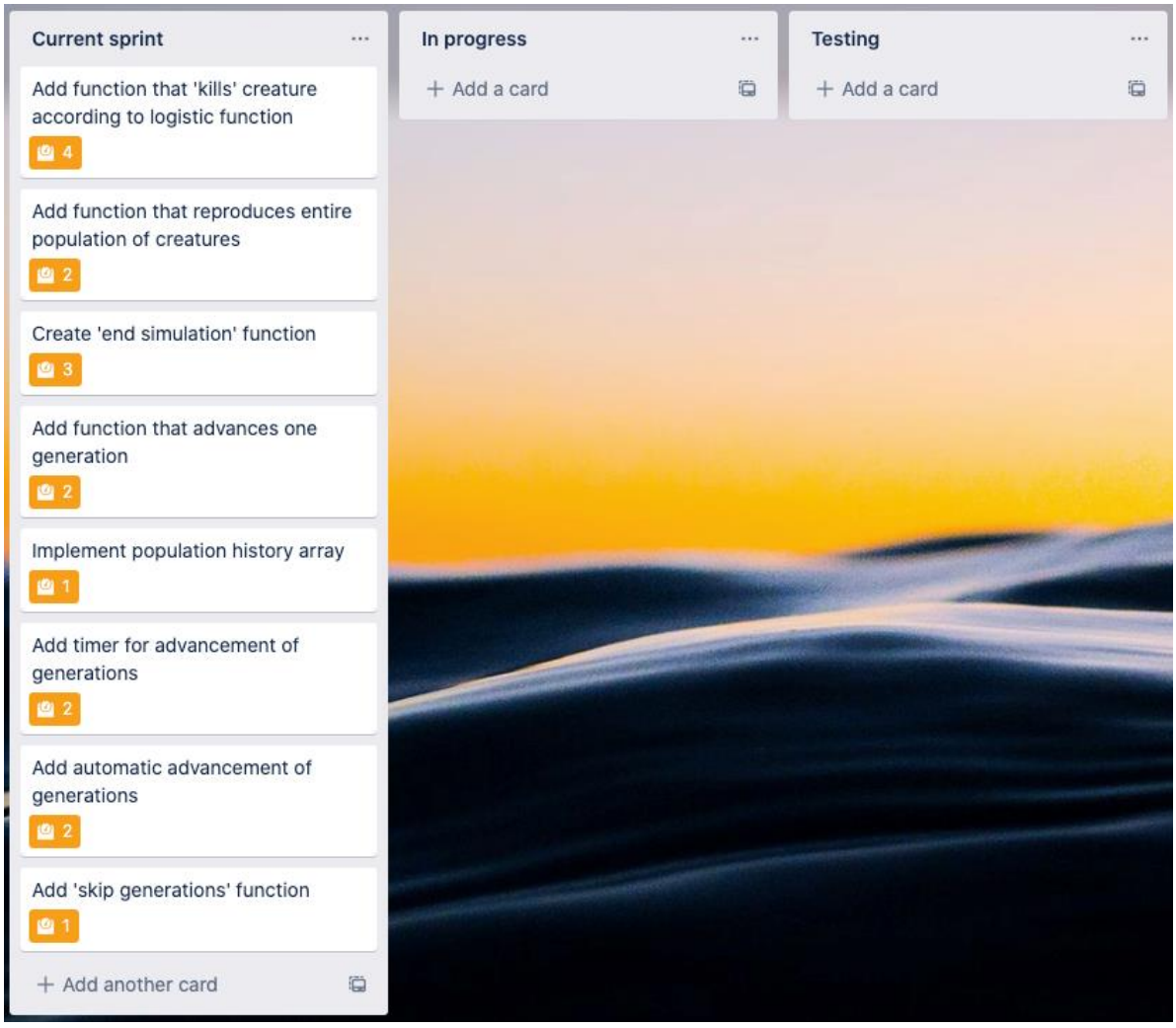
My biology teacher stakeholder is unavailable due to maternity leave.

Conclusion

The app is in too early a stage of development for the stakeholder to give any useful feedback on its progress, so in the next sprint I will work towards a usable app that the stakeholder *can* give feedback on.

Sprint 2

Below is the sprint backlog for this sprint:



If all of these tickets are completed, the code underlying the simulation will be complete, and there will only be the visuals left to complete to produce the minimum viable product (the simplest version of my product that will meet the essential success criteria).

S2 T1: Kill creature by logistic function

As stated above, I will use a logistic function as the probability of a creature dying to meet success criterion 10. To figure out how to implement this maths, I first have to understand the equation of the logistic function. The general equation of this function is

$$f(x) = \frac{L}{1 + e^{-k(x - x_0)}},$$

where

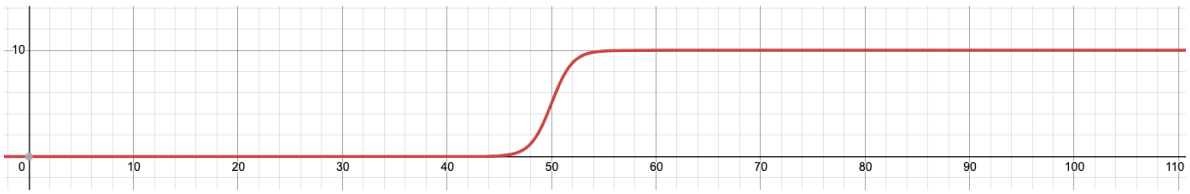
- L is the curve's maximum value,
- x_0 is the curve's midpoint and
- k is the steepness of the curve.

For my simulation, the highest probability of a creature living is 1, so L = 1. I can take a shortcut in the calculation of this function by directly taking the adaptation value I calculated earlier, and I want an adaptation of 50 (approximately the median value of adaptation) to give a ½ chance of living. x_0 will

therefore take a value of 50. I will approximate the value needed for k using the Desmos graphing calculator, and plugging in different values of k until it looks approximately correct.

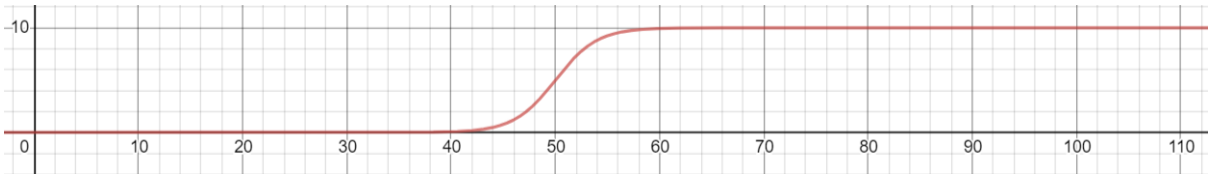
In the following graphs, I have set L = 10 to make the graph easier to see, but in the app it will take a value of 1.

k = 1:



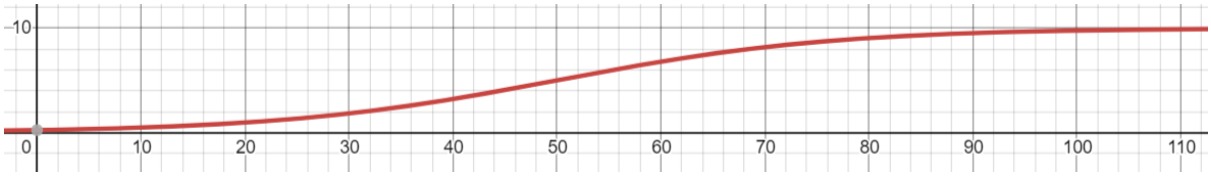
In this case, a creature that is adapted slightly better than randomly, with an adaptation value of 55, is almost certain to survive. The spread of values is not wide enough, which tells me the value of k is too large.

k = 0.5:



There is still not a wide enough spread of values.

k = 0.075:



When k is 0.075, creatures that are perfectly adapted will almost certainly live, while creatures that are reasonably well adapted (adaptation rating = 75) have an ~87% chance of survival. Without doing any more maths, this seems like a reasonable percentage chance, so for now I will use k = 0.075.

Later in the development process, when I add functionality that allows the user to change how punishing an environment is, the value that will change is the x_0 value. A more punishing environment will mean a higher x_0 so that a poorly adapted creature is even less likely to survive.

Maths is a little awkward in Swift, but by approximating the value of *e* and splitting the calculation into two parts I can implement the calculation in a function, like this:

Code

```
38      func calculateChanceOfDeath(creature: Creature) -> Double {
39          let adaptation = Double(calculateCreaturesAdaptation(creature: creature))
40          let denominator = 1 + pow(2.71828, -0.075 * (adaptation - 50))
41          let chanceOfDeath = 1/denominator
42          return chanceOfDeath
43      }
```

Test

```
123
124         // Arrange
125         let environment = Environment(environmentColour: 50, reproductiveChance: 0.5, selectionPressure: 0.1, populationSize: 20)
126
127         // Act
128         for creature in environment.creatures {
129             let chanceOfDeath = environment.calculateChanceOfDeath(creature: creature)
130             XCTAssertGreaterThan(chanceOfDeath, 0)
131             XCTAssertLessThan(chanceOfDeath, 1)
132         }
133     }

140
141         // Arrange
142         let environment = Environment(environmentColour: 50, reproductiveChance: 0.5, selectionPressure: 0.1, populationSize: 20)
143
144         // Act
145         for creature in environment.creatures {
146             let chanceOfDeath = environment.calculateChanceOfDeath(creature: creature)        = Thread 1: step over
147             print(chanceOfDeath)
148             // Breakpoint used to inspect specific values
149         }
150
151     }
```

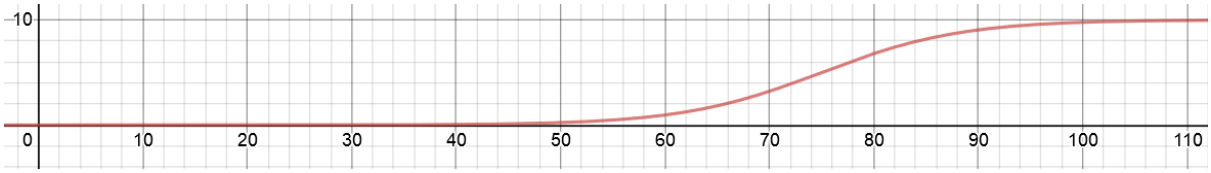
The first test checked whether the value returned by the calculateChanceOfDeath function is always between 0 and 1, and it passed first time. With the second test, I wanted to check two things: first, that the maths was right, and second, that the values returned were **sensible**. I therefore used a breakpoint (the blue arrow) to look at each creature's traitValue, the environment's colour, and each creature's chance of death to see whether the values calculated were sensible.

I realised that the values were *not* sensible, so the test failed. Given an environment with a traitValue of 50, the *worst* adapted creature has a traitValue of 0 or 100. Either way, the creature has an adaptation value of 50, which gives it a 50% chance of survival. This means that for the average environment, the worst odds of a creature living each generation is 50%. This is far too high.

I therefore adjusted the logistic function to return more reasonable numbers.

When

- $x_0 = 75$ and
- $k = 0.15$, this is the graph:



In this case, the worst adapted creature has ~2% chance of survival, which is far more reasonable. I adjusted my calculateChanceOfDeath function to use these new values.

Updated code

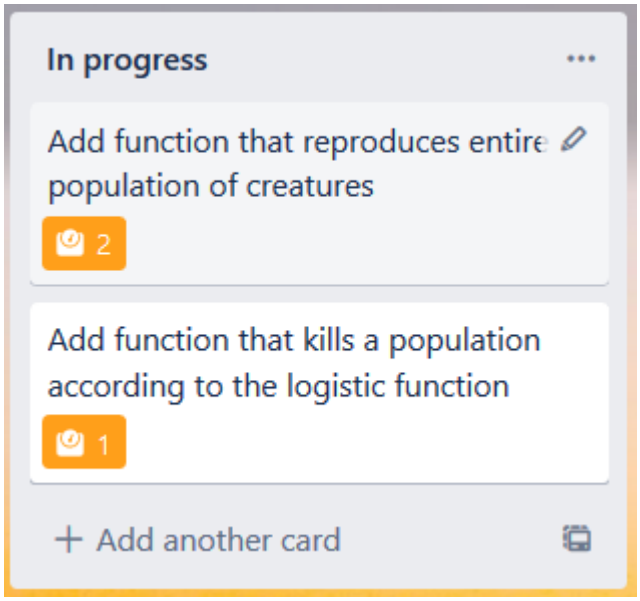
```
38      func calculateChanceOfDeath(creature: Creature) -> Double {
39          let adaptation = Double(calculateCreaturesAdaptation(creature: creature))
40          let denominator = 1 + pow(2.71828, -0.15 * (adaptation - 75))
41          let chanceOfSurvival = 1/denominator
42          return 1 - chanceOfSurvival
43      }
```

Now when the environment's traitValue is 50 and the creature's traitValue is 79 (below average), the creature's chance of living each generation is about 35% (below average). The mathematics to reach these results is correct, and the numbers produced are sensible. The relevant ticket can therefore move from "Testing" to "Complete".

S2 T2: Advancing one generation

The code written so far has made it possible to create a creature, create an environment, populate the environment with creatures, reproduce a creature, and to kill a creature according to how well it is adapted. All this now allows me to advance a generation in the simulation - this is the key step to completing success criterion 10. Advancing a generation can be split into two steps: first, each creature must be killed or not killed depending on how well adapted it is to the environment. Then each remaining creature must be allowed to reproduce.

I will move the two relevant tickets into "In progress":



Updating each generation in this way is what underlies the simulation. In this app, 'killing' a creature involves removing it from the array of creatures inside an environment.

Pseudocode

See pages 25-26.

Code

```
45      func advanceOneGeneration() {
46
47          for creature in self.creatures {
48              let fraction = Double.random(in: 0...1)
49              let chanceOfDeath = calculateChanceOfDeath(creature: creature)
50              if fraction < chanceOfDeath {
51                  self.creatures = self.creatures.filter { $0 != creature }        Binary operator '!=' cannot be applied to two 'Creature' operands
52              }
53          }
54
55      }
```

Before even implementing a test, however, I got an error. The error occurred because I did not make the Creature class comparable, meaning I cannot compare one instance of the class to another. Instead of making the Creature class comparable, which requires each instance of Creature to be instantiated with a unique identifier like a name or number (making the code more complex), I simply changed the way the creatures were removed from the array.

Updated code

```
for i in 0...(self.creatures.count - 1) {
    let fraction = Double.random(in: 0...1)
    let chanceOfDeath = calculateChanceOfDeath(creature: self.creatures[i])
    if fraction < chanceOfDeath {
        self.creatures.remove(at: i)
    }
}
```

The surviving creatures then had to reproduce:

```
for creature in self.creatures {
    let fraction = Double.random(in: 0...1)
    if fraction < self.reproductiveChance {
        self.creatures.append(creature.reproduce())
    }
}
```

Combining these two blocks of logic together gave me a function which advances one generation.

```
45      func advanceOneGeneration() {
46
47          for i in 0...(self.creatures.count - 1) {
48              let fraction = Double.random(in: 0...1)
49              let chanceOfDeath = calculateChanceOfDeath(creature: self.creatures[i])
50              if fraction < chanceOfDeath {
51                  self.creatures.remove(at: i)
52              }
53          }
54
55          for creature in self.creatures {
56              let fraction = Double.random(in: 0...1)
57              if fraction < self.reproductiveChance {
58                  self.creatures.append(creature.reproduce())
59              }
60          }
61
62      }
```

Test

Before even writing the test, I called this method on an environment to check if it worked, but I got this error message:

```
func advanceOneGeneration() {

    for i in 0...(self.creatures.count - 1) {
        let fraction = Double.random(in: 0...1)
        let chanceOfDeath = calculateChanceOfDeath(creature: self.creatures[i])   = Thread 1: Fatal error: Index out of range
        if fraction < chanceOfDeath {
            self.creatures.remove(at: i)
        }
    }

    for creature in self.creatures {
        let fraction = Double.random(in: 0...1)
        if fraction < self.reproductiveChance {
            self.creatures.append(creature.reproduce())
        }
    }

}
```

“Index out of range” tells me the for loop I wrote counted higher than the index of the last item in the array. This error occurred because removing creatures shortens the array, so the length of the array changes throughout the loop. To fix this, I added only creatures that survived to a new array and set that new array to self.creatures after the loop had finished.

Updated code

```
func advanceOneGeneration() {

    var tempArray: [Creature] = []
    for i in 0...(self.creatures.count - 1) {
        let fraction = Double.random(in: 0...1)
        let chanceOfDeath = calculateChanceOfDeath(creature: self.creatures[i])
        if fraction > chanceOfDeath {
            tempArray.append(self.creatures[i])
        }
    }
    self.creatures = tempArray

    for creature in self.creatures {
        let fraction = Double.random(in: 0...1)
        if fraction < self.reproductiveChance {
            self.creatures.append(creature.reproduce())
        }
    }

}
```

Test

```
func advanceOneGeneration() {

    var tempArray: [Creature] = []
    for i in 0...(self.creatures.count - 1) {                    = Thread 1: Fatal error: Can't form Range with upperBound < lowerBound
        let fraction = Double.random(in: 0...1)
        let chanceOfDeath = calculateChanceOfDeath(creature: self.creatures[i])
        if fraction > chanceOfDeath {
            tempArray.append(self.creatures[i])
        }
    }
    self.creatures = tempArray

    for creature in self.creatures {
        let fraction = Double.random(in: 0...1)
        if fraction < self.reproductiveChance {
            self.creatures.append(creature.reproduce())
        }
    }

}
```

While manually testing this function, I came across this error. Using the debugging tools built into XCode, I realised this happened because the self.creatures array was empty, i.e. every creature had died. The for loop was therefore trying to count from 0 to -1, which threw an error. I therefore added a guard statement before the kill and reproduce loops to ensure the self.creatures array *wasn't* empty. If it was, I simply did nothing – later, I will change this so the user gets a popup telling them the creatures have gone extinct.

Updated code

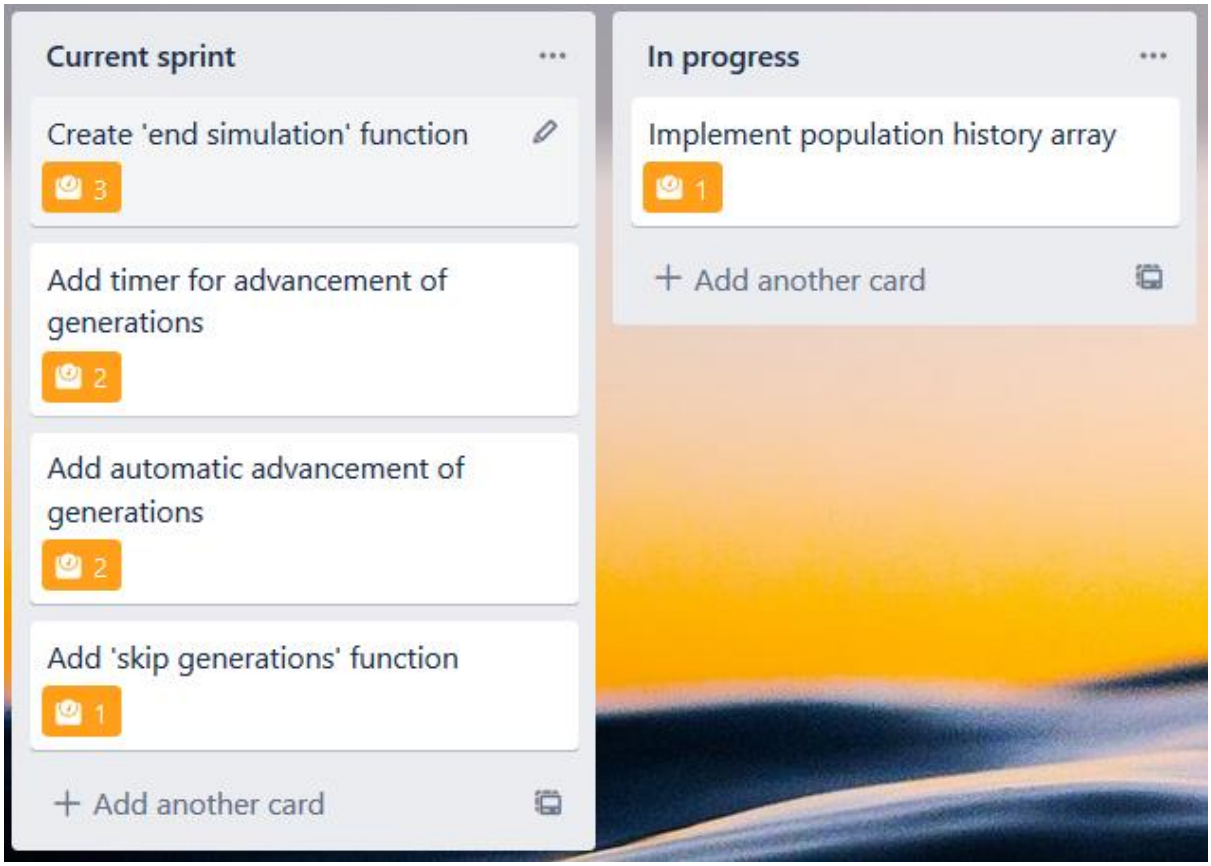
```
func advanceOneGeneration() {

    guard self.creatures.count > 0 else {
        // End simulation. Creatures have gone extinct!
        return
    }

    var tempArray: [Creature] = []
    for i in 0...(self.creatures.count - 1) {
        let fraction = Double.random(in: 0...1)
        let chanceOfDeath = calculateChanceOfDeath(creature: self.creatures[i])
        if fraction > chanceOfDeath {
            tempArray.append(self.creatures[i])
        }
    }
    self.creatures = tempArray

    guard self.creatures.count > 0 else {
        // End simulation. Creatures have gone extinct!
        return
    }

    for creature in self.creatures {
        let fraction = Double.random(in: 0...1)
        if fraction < self.reproductiveChance {
            self.creatures.append(creature.reproduce())
        }
    }

}
```

Test

Now I receive no errors when manually testing boundary cases such as extinct creatures. This function is very hard to automatically test because the nature of the advanceGeneration method is that there is probability involved, so there is no expected result I can test against. That being said, I no longer receive errors when I test the advanceGeneration method with any test data, so I will move the two tickets from “Testing” into “Complete”.

S2 T3: Population history array

Having written the code to advance one generation, I can begin to implement the population history array which is the key functionality for success criterion 13.



This involves adding a new attribute called populationHistory that is an array of arrays of creatures. Each item will be the array of creatures of a given generation, so that in the simulation summary each generation is easily accessible.

Pseudocode

See pages 25-26.

Code

```
let environmentColour: Int
let reproductiveChance: Double
let selectionPressure: Double
var creatures: [Creature] = []
var populationHistory: [[Creature]] = []

init(environmentColour: Int, reproductiveChance: Double, selectionPressure: Double, populationSize: Int) {
    self.environmentColour = environmentColour
    self.reproductiveChance = reproductiveChance
    self.selectionPressure = selectionPressure
    for _ in 1...populationSize {
        let creature = Creature(traitValue: nil)
        self.creatures.append(creature)
    }
    self.populationHistory.append(self.creatures)
}
```

The first entry in populationHistory is the initial population of creatures. After this, each time the advanceGeneration method is called, the new population will be appended to populationHistory.

Test

```
167
168     func testPopulationHistoryArrayIsUpdatedAfterEachGeneration() {
169
170         // Arrange
171         let environment = Environment(environmentColour: 50, reproductiveChance: 0.5, selectionPressure: 0.1, populationSize: 20)
172
173         // Act
174         for _ in 1...5 {
175             environment.advanceOneGeneration()
176         }
177         let expected = 6
178         let actual = environment.populationHistory.count
179
180         // Assert
181         XCTAssertEqual(actual, expected)
182
183     }
```

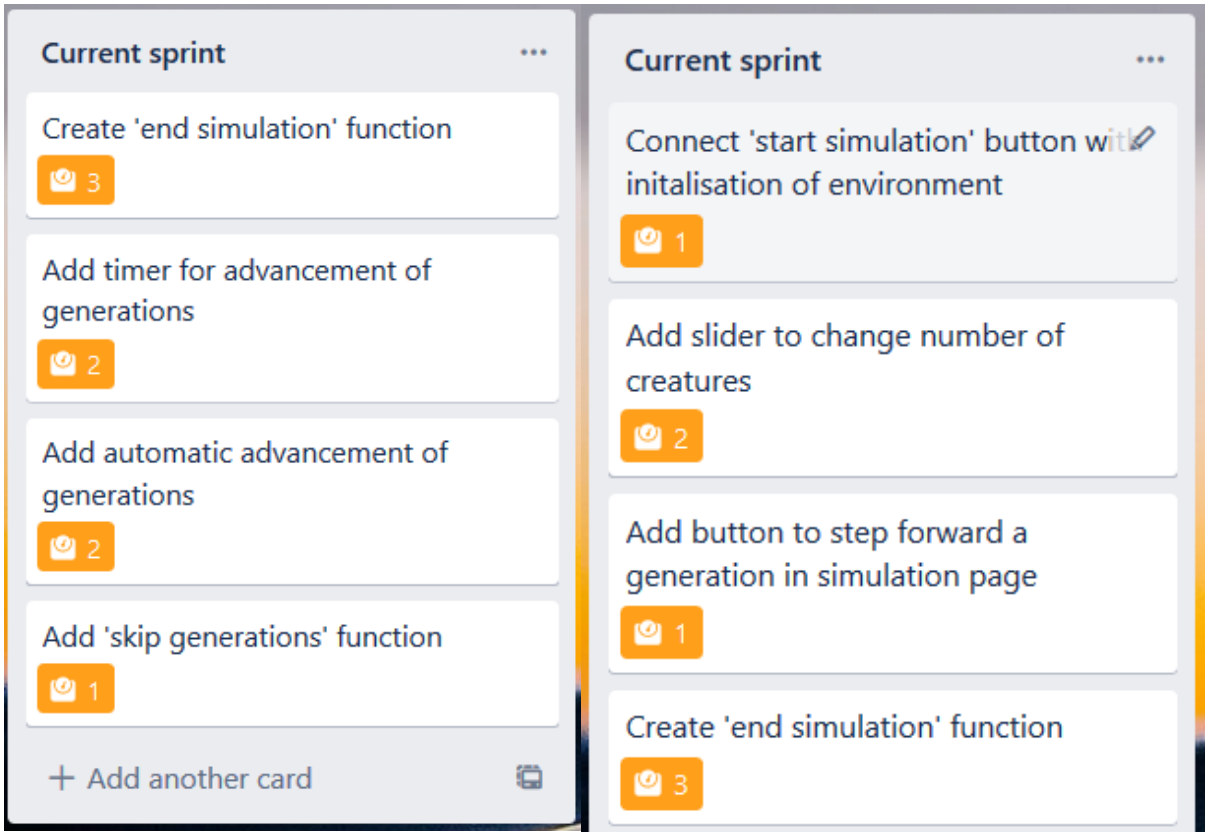
This test ensures an entry is made in populationHistory each generation, including the first one. This test passed first time, which tells me the implementation of populationHistory works as intended. To ensure more thoroughly that it works correctly, I added break points using XCode's inbuilt debugging functionality to inspect populationHistory after each generation, and my implementation does work. I will therefore move this ticket from "Testing" to "Completed".

S2 T4: Developing the UI

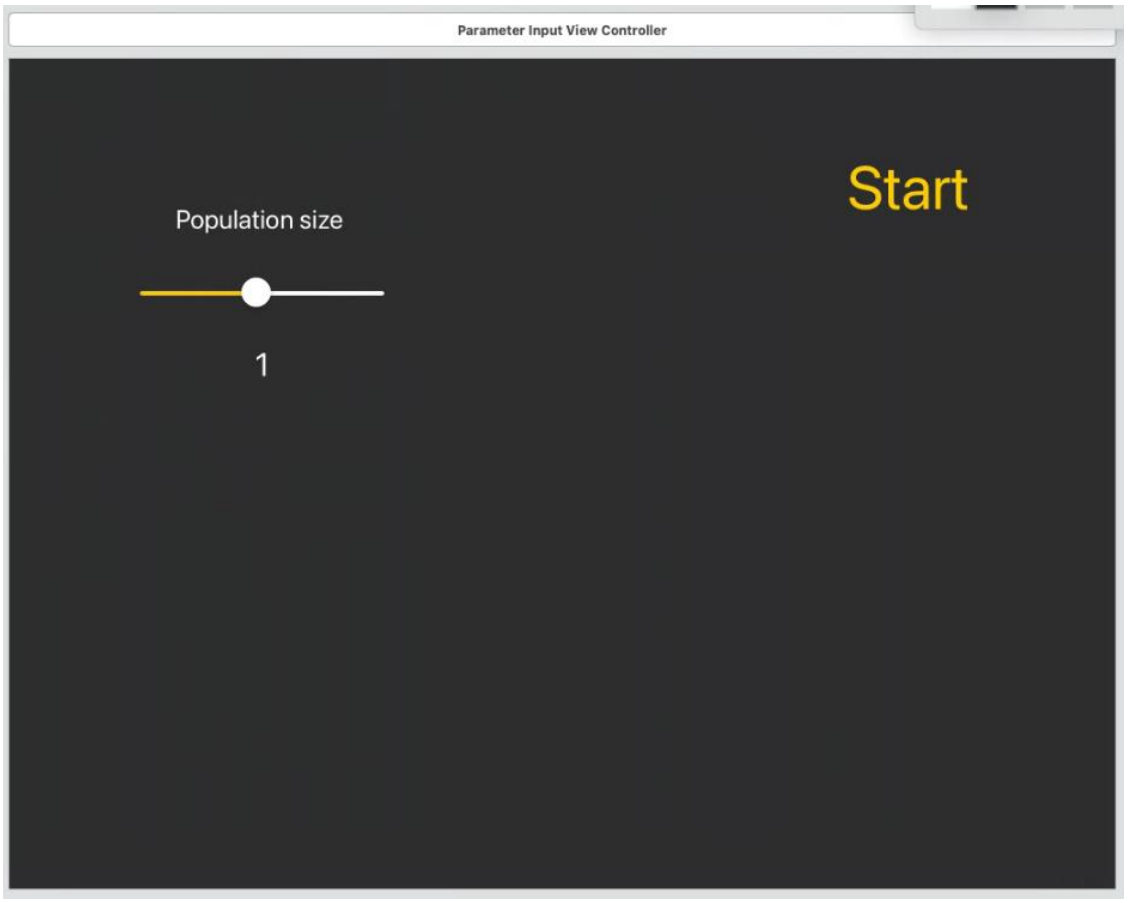
Looking at the next tickets in my backlog, I realised that it was hard to implement them without first fleshing out the UI from its extremely simple structure into something that has more functionality. I therefore decided to edit the tickets in my "Current sprint" list to prioritise the UI, which will contribute to success criteria 7 and 10.

Initial backlog

Updated backlog



Parameter input view UI



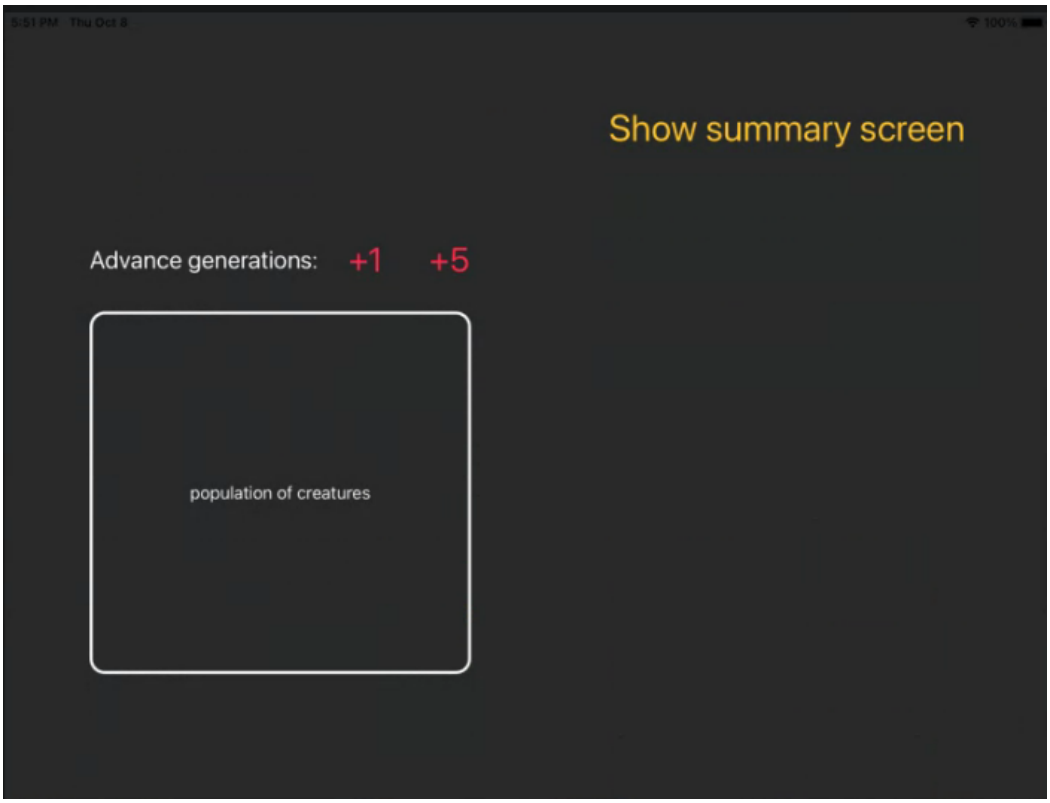
Parameter input view code

```
class ParameterInputViewController: UIViewController {

    var populationSize: Int = 10
    @IBOutlet weak var populationSlider: UISlider!
    @IBOutlet weak var populationSizeLabel: UILabel!

    override func viewDidLoad() {
        super.viewDidLoad()

        // Do any additional setup after loading the view.
    }

    @IBAction func showSimulationVC(_ sender: Any) {

        let environment = Environment(environmentColour: 50, reproductiveChance: 0.5, selectionPressure: 0.1, populationSize:
            self.populationSize)

        if let vc = storyboard?.instantiateViewController(identifier: "simulationVC") as? SimulationViewController {
            vc.environment = environment
            vc.modalPresentationStyle = .fullScreen
            present(vc, animated: true, completion: nil)
        }
    }

    @IBAction func populationSizeChanged(_ sender: UISlider) {
        let value = Int(sender.value)
        self.populationSize = value
        populationSizeLabel.text = String(value)
    }

}
```

I connected the slider and label to the parameter input scene so that the UI will have some actual functionality.

This gave me a simple parameter input page in which I can change the population size using a slider, and a ‘Start’ button that initialises an instance of the Environment class with the population size specified by the user.

Simulation view UI

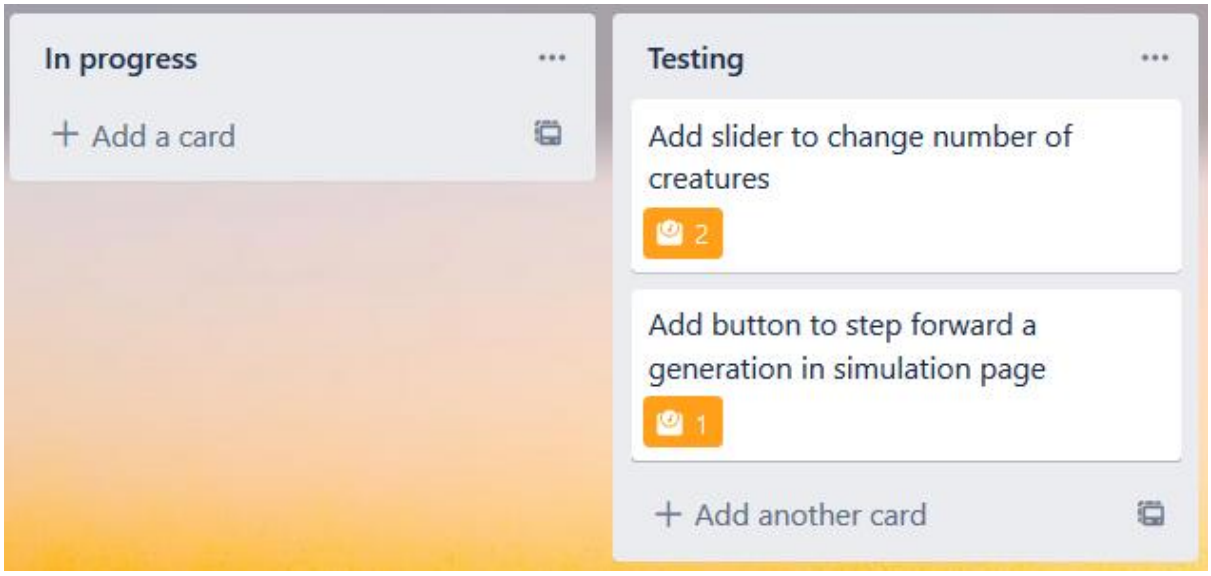


Simulation view code

```
import UIKit

class SimulationViewController: UIViewController {

    var environment: Environment?

    @IBOutlet weak var displayPopulation: UILabel!

    override func viewDidLoad() {
        super.viewDidLoad()

        displayPopulation.layer.borderWidth = 3.0
        displayPopulation.layer.cornerRadius = 15
        displayPopulation.layer.borderColor = UIColor.white.cgColor
    }

    func updatePopulationDisplay() {
        var printString: String = ""
        for creature in environment!.creatures {
            let intToString = String(creature.traitValue)
            printString.append(intToString)
        }
        displayPopulation.text = printString
    }


    @IBAction func showSummaryVC(_ sender: Any) {
        if let vc = storyboard?.instantiateViewController(identifier: "simulationSummaryVC") as? SimulationSummaryViewController {
            vc.environment = environment
            vc.modalPresentationStyle = .fullScreen
            present(vc, animated: true, completion: nil)
        }
    }

    @IBAction func advanceOneGeneration(_ sender: Any) {
        self.environment?.advanceOneGeneration()
        updatePopulationDisplay()
    }

    @IBAction func advanceFiveGenerations(_ sender: Any) {
        for _ in 1...5 {
            self.environment?.advanceOneGeneration()
        }
        updatePopulationDisplay()
    }

}
```

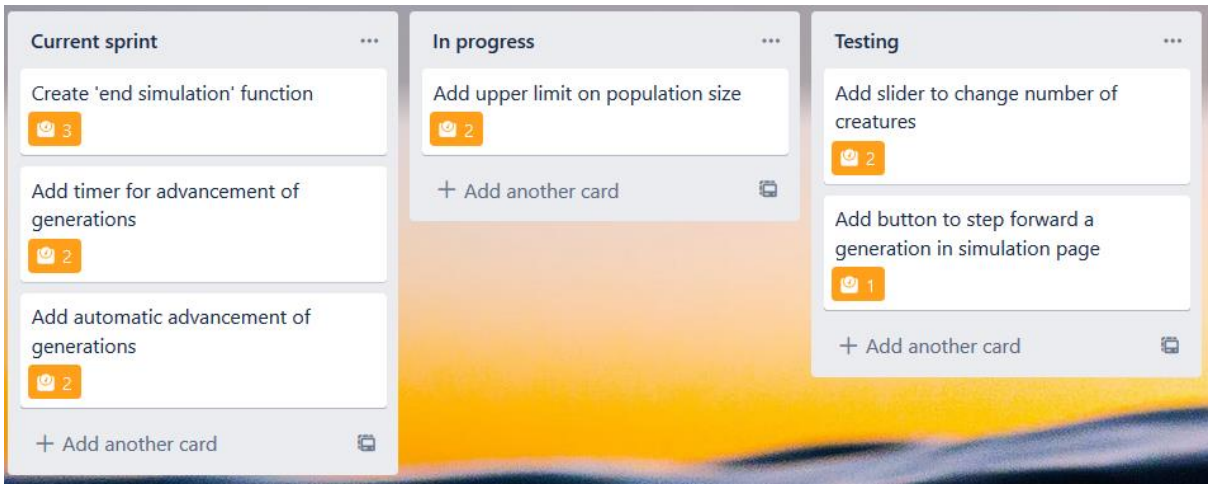
I wanted to add the ability to advance generations through the simulation, so I added the buttons and labels needed. I then connected these buttons and labels to the simulation view controller, adding functionality to the UI features.

Now, clicking the '+1' and '+5' buttons advance one and five generations respectively. The features I just added (parameter input using sliders and buttons to advance generations) have been tested manually, but do not have any automated UI tests, so I will move the relevant tickets from "In progress" to "Testing".



S2 T5: Adding upper limit to population size

While testing the app manually, I advanced 5 generations at a time to see what would happen if I pushed the simulation to its limits. What I quickly realised is that there is no upper limit on the size of each population, so the population could grow indefinitely – the app quickly froze while trying to do some relatively complex maths on each member of an insanely huge population. Not only is this a problem for users of the app, it is also not representative of actual evolution: when a population grows too large, competition for resources amongst the population itself kills off some of the creatures, providing a natural upper limit to the size of the population. I added a ticket to implement an upper limit on the population size to make the app more usable and accurate. This ticket contributes to success criterion 10.



To implement this upper limit, I had to decide how large a population I wanted. I decided on an upper limit of around 30 creatures, so that as the population approaches this size some variable makes it more likely for each creature to die each generation.

My initial solution was very simple: as the population size grows, the midpoint of the logistic kill curve increases linearly with the population size from its initial value of 75. Mathematically, this means that the midpoint $x_0 = 75 + c * (population\ size)$, where c is some constant. A large value of c would

keep the population size small, while a small value of c would let the population size grow very large as the simulation progresses.

Code

I first implemented this with $c = 1$:

```
func calculateChanceOfDeath(creature: Creature) -> Double {
    let adaptation = Double(calculateCreaturesAdaptation(creature: creature))
    let c = 1
    let dampingFactor = Double(c * self.creatures.count)
    let denominator = 1 + pow(2.71828, -0.15 * (adaptation - (75 + dampingFactor)))
    let chanceOfSurvival = 1/denominator
    return 1 - chanceOfSurvival
}
```

Test

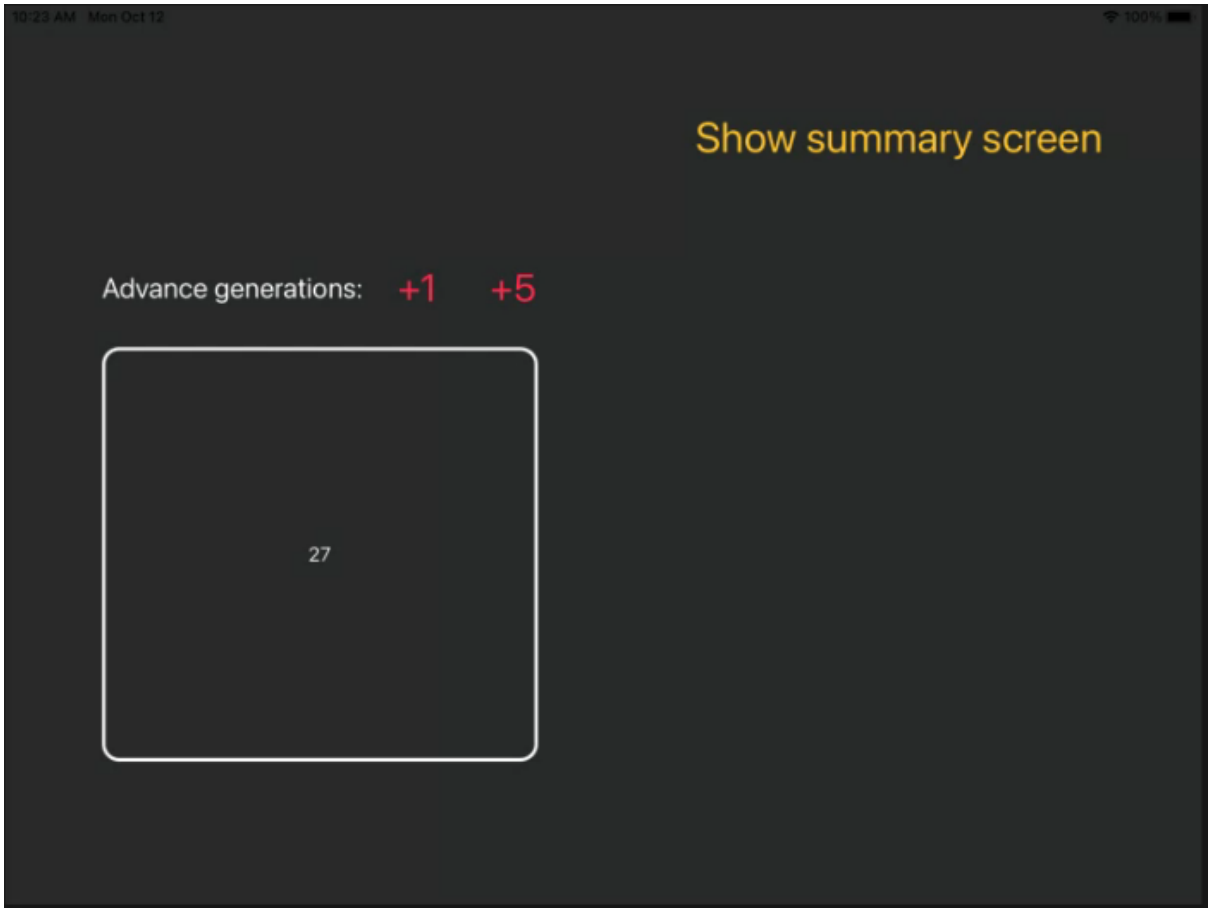
Now when I add 5 generations at a time, my population rarely grows above 20 creatures. I wanted my population to grow larger than this, so I then used $c \cong \frac{2}{3}$:

Updated code

```
func calculateChanceOfDeath(creature: Creature) -> Double {
    let adaptation = Double(calculateCreaturesAdaptation(creature: creature))
    let c: Double = 0.66
    let dampingFactor = c * Double(self.creatures.count)
    let denominator = 1 + pow(2.71828, -0.15 * (adaptation - (75 + dampingFactor)))
    let chanceOfSurvival = 1/denominator
    return 1 - chanceOfSurvival
}
```

Test

The population size now hovers around 25:



This works exactly as intended, and if needs be I can change the value of c very easily later. I therefore moved the relevant ticket into 'Complete' and moved on to the next ticket.

S2 T6: Ending the simulation

When the population size equals zero, the creatures have gone extinct and it becomes impossible for the simulation to continue – no creatures to reproduce means the population size will simply stay at zero. When this happens, I want to add a popup explaining briefly what had happened, relate this event to extinction, then allow the user to move into the simulation summary page. This will ensure the app is easy to understand, which is a key trait for an educational tool.

To implement this feature, I first needed to set up a system that caught when the population became zero. To do this, I made the 'advanceOneGeneration' function return the string "extinct" if the population size is zero using the guard statement I implemented earlier.

Code

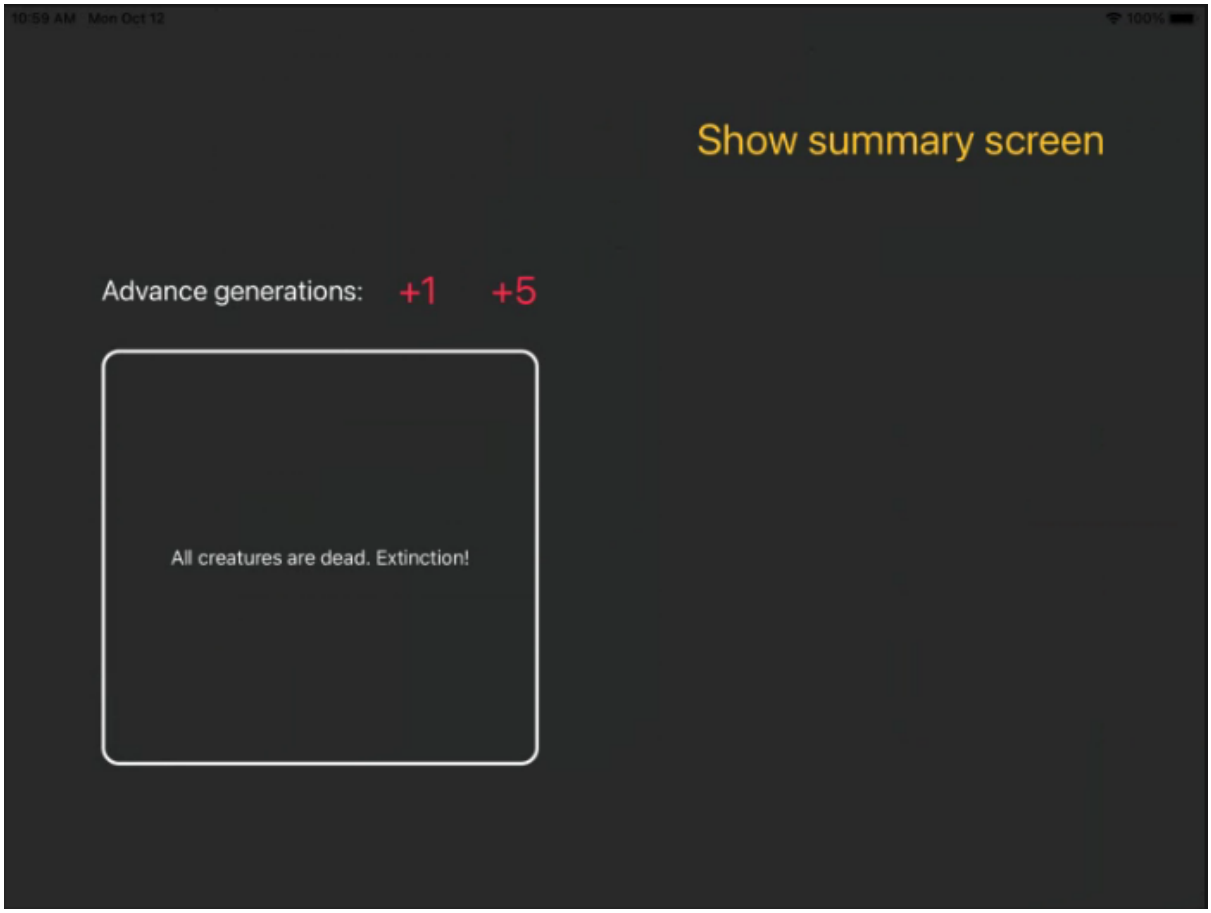
```
func advanceOneGeneration() -> String? {

    guard self.creatures.count > 0 else {
        return "extinct"
    }
```

In my simulation view controller, I added some logic that checked whether the 'advanceOneGeneration' function had returned "extinct", and if it had, to do something inside an if statement:

```
@IBAction func advanceOneGenerationButton(_ sender: Any) {
    if environment?.advanceOneGeneration() == "extinct" {
        // Do something (like create a popup window)
    } else {
        updatePopulationDisplay()
    }
}
```

Test

To test this function, I replaced the comment with a simple display command. When the population size reaches zero, this happens:



Now, instead of printing a message to this display, I instead want to call a popup. First, I wrote the simple function that creates a popup.

Code

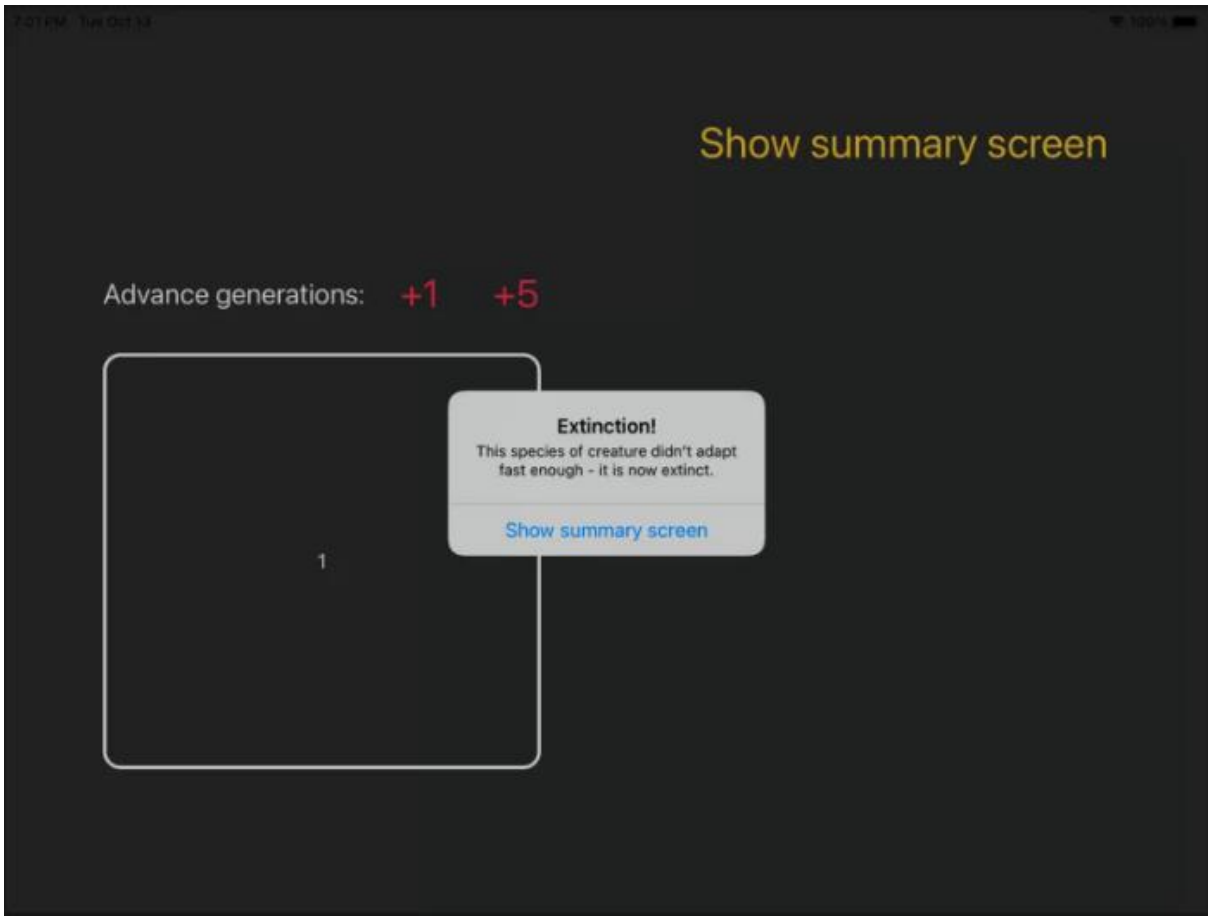
```
func extinctionAlert() {
    let alert = UIAlertController(title: "Extinction!", message: "This species of creature didn't adapt fast enough - it is now
        extinct", preferredStyle: .alert)
    alert.addAction(UIAlertAction(title: "Show summary screen", style: .default, handler: { action in
        self.showSummaryVC()
    }))
    self.present(alert, animated: true)
}
```

This function needs to be called if the population size hits zero.

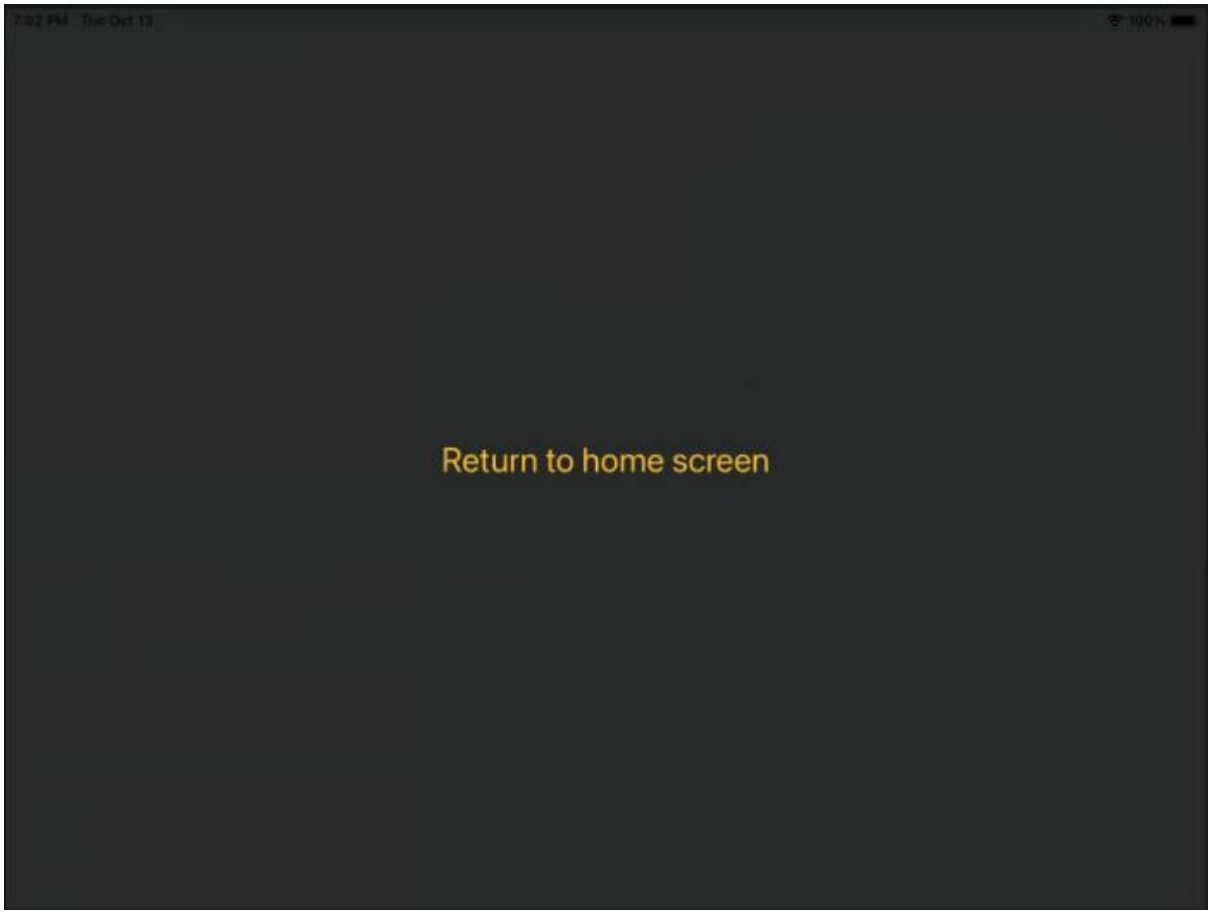
```
@IBAction func advanceOneGenerationButton(_ sender: Any) {
    if environment?.advanceOneGeneration() == "extinct" {
        extinctionAlert()
    } else {
        updatePopulationDisplay()
    }
}
```

Test

I ran the app, and now when the population size hits zero, this is what the user sees:



And pressing the button brings me to the summary screen:



The ticket to end the simulation can therefore be moved into “Complete”.

End of sprint review

Stakeholder feedback

Questions:

- What did you like about the app?
- What would you like to see changed?
- What should I focus on moving forward?

G. L.: I really liked the user interface as it was very clean. I liked the minimalistic style and colour scheme, make sure you stick with these colours. It makes it look very clean and scientific and modern; a lot of science simulators look quite ugly even If they have good science behind it. While the UI is very nice, It would be nice to see some animations in the future, maybe like some circles randomly moving around or something (imagine like a top down view of the creatures that looks a bit like the particle model for like liquids that increases and can decrease in size. Even if complex animations are outside the scope of this project which I can understand, more explanation would be nice. I’m thinking about those icons with (i) that you can press which you can click on to get more information. Maybe have some explanation within the app as to what “advance generations” and other functions actually mean and what exactly is going on. How does the starting population size have an impact on these factors? So, what I mean to say is that if you can’t display something through animations maybe add some

more text explanation. Maybe think about how the app can be more educational and how it can better teach the concept of evolution to users.

My biology teacher stakeholder is on maternity leave, so I am not able to ask for her feedback.

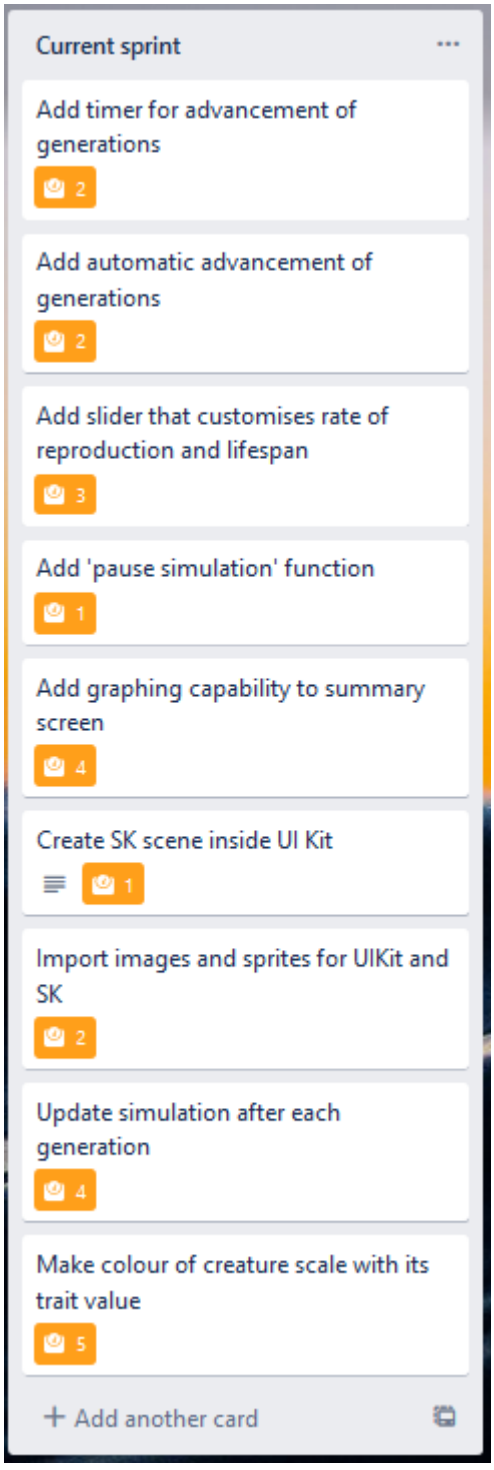
Conclusion

From the stakeholder feedback, it's clear that I haven't made the app as easy to understand as it needs to be for an educational tool. I therefore need to devote some time to adding description throughout the app of – for instance – what each slider does, what the buttons are for, and what is actually *happening* as the simulation progresses.

I am satisfied with the progress I made in this sprint, so I will work on roughly the same number of tickets in the next sprint while keeping the stakeholder feedback in mind.

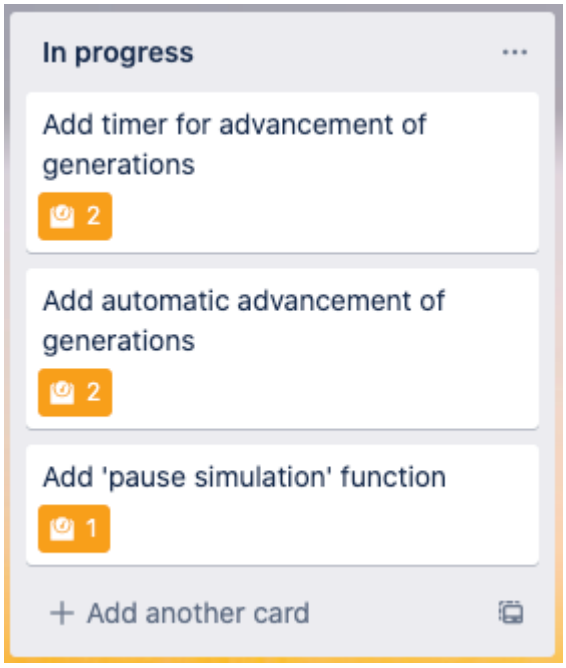
Sprint 3

Below is the sprint backlog for the third sprint.



S3 T1: Add automatic advance through generations

I wanted to allow the user to automatically advance through generations instead of manually pressing the “+1” or “+5” button, just so the app is easier to use. To start with, I moved the relevant tickets into “In progress”. This contributes to success criterion 10.



The first thing I needed was a timer that 'ticked' as fast as I wanted new generations to be created.

Code

```
let timer = Timer.scheduledTimer(timeInterval: 1.0, target: self, selector: #selector(autoAdvanceGeneration), userInfo: nil,
    repeats: true)
```

This timer ticks once per second, and each time it ticks it should advance one generation. To implement this and to simultaneously clean up my code, I created a separate function that advances one generation. Pressing the "+1" or "+5" button now calls this separate function once or five times respectively.

Refactored code

```
func advanceOneGeneration() {
    if environment?.advanceOneGeneration() == "extinct" {
        extinctionAlert()
    } else {
        updatePopulationDisplay()
    }
}

@IBAction func advanceOneGenerationButton(_ sender: Any) {
    advanceOneGeneration()
}

@IBAction func advanceFiveGenerations(_ sender: Any) {
    for _ in 1...5 {
        advanceOneGeneration()
    }
}

let timer = Timer.scheduledTimer(timeInterval: 1.0, target: self, selector: #selector(autoAdvanceGeneration), userInfo: nil,
    repeats: true)

@objc func autoAdvanceGeneration() {
    advanceOneGeneration()
}
```

Test

```
libsystem_kernel.dylib`__pthread_kill:
    0x7fff5dca6330 <+0>:  movl   $0x2000148, %eax          ; imm = 0x2000148
    0x7fff5dca6335 <+5>:  movq   %rcx, %r10
    0x7fff5dca6338 <+8>:  syscall
->  0x7fff5dca633a <+10>: jae    0x7fff5dca6344            ; <+20>
    0x7fff5dca633c <+12>: movq   %rax, %rdi
    0x7fff5dca633f <+15>: jmp    0x7fff5dca0629
    0x7fff5dca6344 <+20>: retq
    0x7fff5dca6345 <+21>: nop
    0x7fff5dca6346 <+22>: nop
    0x7fff5dca6347 <+23>: nop
```

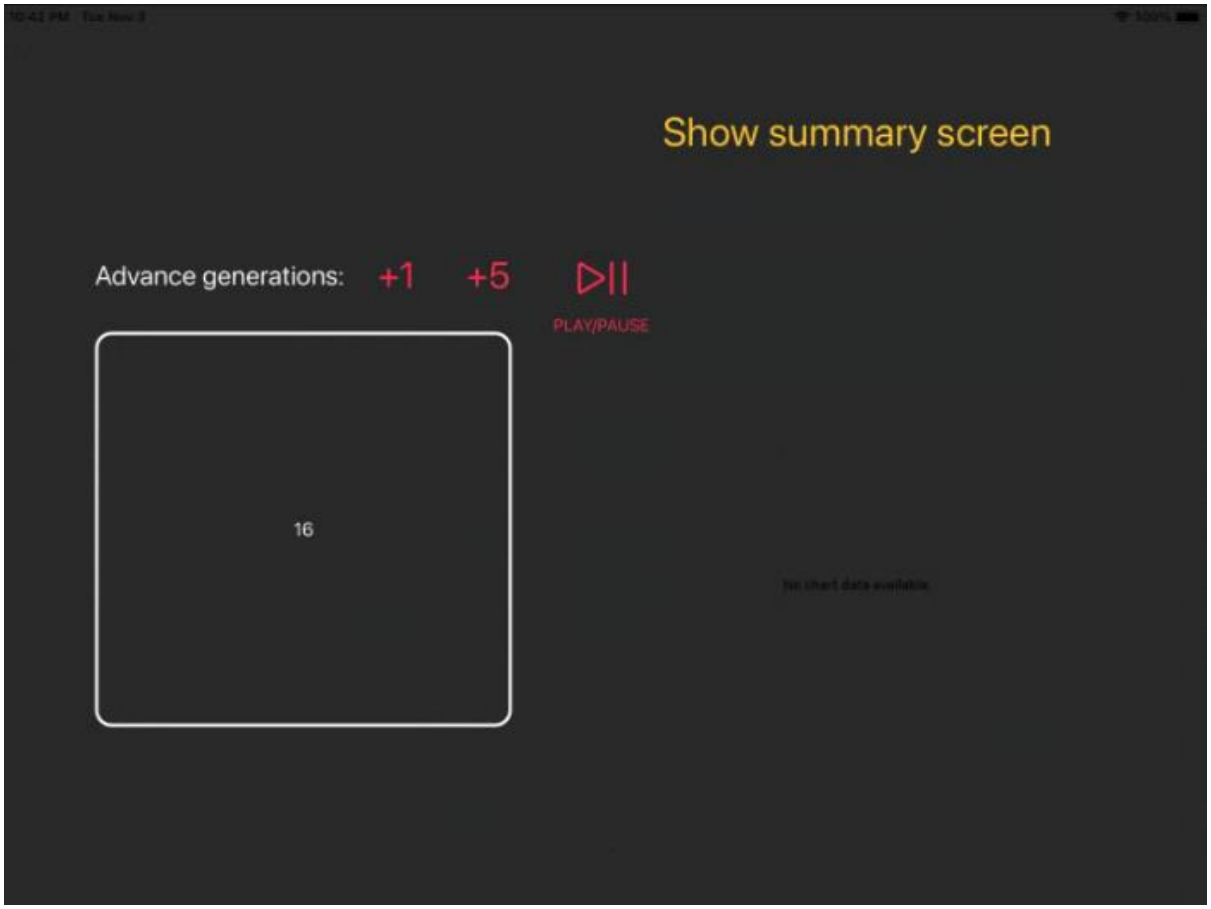
Thread 1: "-[__SwiftValue autoAdvanceGeneration]: unrecognized selector sent to instance 0x600000c65710"

When I ran the code, it failed dramatically. This failure happened because the timer isn't attached to anything – the timer isn't yet called by anything. I therefore attached the timer to a button and slightly refactored the code again.

Updated code

```
func advanceOneGeneration() {
    if environment?.advanceOneGeneration() == "extinct" {
        extinctionAlert()
    } else {
        updatePopulationDisplay()
    }
}

@IBAction func advanceOneGenerationButton(_ sender: Any) {
    advanceOneGeneration()
}

@IBAction func advanceFiveGenerations(_ sender: Any) {
    for _ in 1...5 {
        advanceOneGeneration()
    }
}

@IBAction func startStopAutoAdvance(_ sender: Any) {
    if timer == nil {
        timer = Timer.scheduledTimer(withTimeInterval: 1.0, repeats: true) { timer in
            self.advanceOneGeneration()
        }
    } else {
        timer!.invalidate()
        timer = nil
    }

}
```

Test



To test this timer, I simply ran the app and pressed the start/stop button several times with gaps of at least 5 seconds in between presses. I observed that the size of the population changed each second when the timer was on (when I first pressed the button), and stopped changing when the timer was off (when I pressed it again). This told me that the automatic advancement of generations worked as expected.

S3 T2: Add bar chart of population to simulation page

From the stakeholder feedback in the second sprint's review, I realised I should prioritise visuals for what was happening inside the population. I previously had only a number showing the population size (which was useful for testing). I therefore began work on a histogram that intuitively summarised the distribution of creatures in each generation, as per **success criteria 9 and 12**.

It would be effectively impossible for me to single-handedly create a complex graph in Swift in the time available, so I first had to install a library that handles graphs. I decided to use a library called "Charts" by Daniel Cohen Gindi as it was by far the most widely recommended and seemed to include all the functionality I needed.

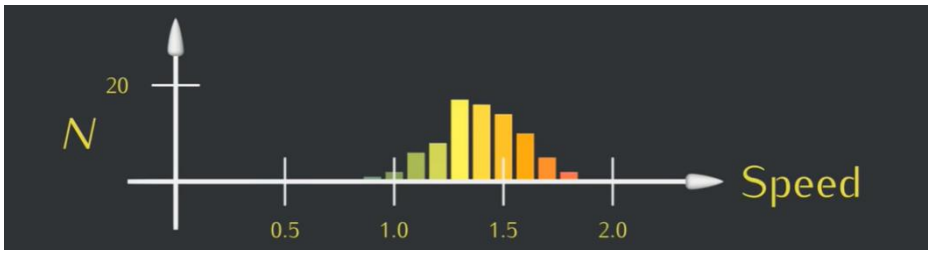
Installing this library was very easy. I first added CocoaPods (a library dependency manager for Swift and C) to my project by running 'pod init' in the macOS command line. I then ran 'pod install', which referenced the below file in which I had added the name of the library.

```
# Uncomment the next line to define a global platform for your project
# platform :ios, '9.0'

target 'Evolution simulator' do
  # Comment the next line if you don't want to use dynamic frameworks
  use_frameworks!

pod 'Charts'

  # Pods for Evolution simulator

  target 'Evolution simulatorTests' do
    inherit! :search_paths
    # Pods for testing
  end

  target 'Evolution simulatorUITests' do
    # Pods for testing
  end

end
```

This allowed me to use the library in my project.

I then added a BarChartView to my SimulationViewController.

This is the view in which I will load my graph. I wanted this graph to be a bar chart similar to this one from page 16 of my Design section.



Each section is a range of traitValues (e.g. 1-10, 11-20), where the height of each bar is the number of creatures with traitValues in that range in the current generation.

Code

```
func setChart() {

    yValues = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
    var dataEntries: [BarChartDataEntry] = []

    for creature in environment!.creatures {
        if 1...10 ~= creature.traitValue {
            yValues[0] += 1
        } else if 11...20 ~= creature.traitValue {
            yValues[1] += 1
        } else if 21...30 ~= creature.traitValue {
            yValues[2] += 1
        } else if 31...40 ~= creature.traitValue {
            yValues[3] += 1
        } else if 41...50 ~= creature.traitValue {
            yValues[4] += 1
        } else if 51...60 ~= creature.traitValue {
            yValues[5] += 1
        } else if 61...70 ~= creature.traitValue {
            yValues[6] += 1
        } else if 71...80 ~= creature.traitValue {
            yValues[7] += 1
        } else if 81...90 ~= creature.traitValue {
            yValues[8] += 1
        } else if 91...100 ~= creature.traitValue {
            yValues[9] += 1
        }
    }

    for i in 0 ..< xValues.count {
        let dataEntry = BarChartDataEntry(x: Double(i), y: Double(yValues[i]))
        dataEntries.append(dataEntry)
    }

    let chartDataSet = BarChartDataSet(dataEntries)
    let chartData = BarChartData(dataSet: chartDataSet)
    simulationBarChartView.data = chartData

}
```

Test

This chart does not respond to user input, as it only relies on the updates to the simulation. I can therefore only test that it displays the correct generation's information.

Initial population

I initiated this simulation with a generation of 10 creatures. The view on the left displays the number of creatures in the current generation, so we can see the simulation was initiated correctly. In the bar chart to the right, the distribution is random (as we would expect) and the sum of the creatures (1 + 3 + 2 + 2 + 1 + 1) is 10. The bar chart was therefore initiated correctly.

Next generation

Stepping forward another generation, there are now 2 creatures, which is reflected in the graph.

Lots of generations

After stepping through many more generations, the population approaches a point where there is a small distribution of creatures around 50 (the traitValue of the environment).

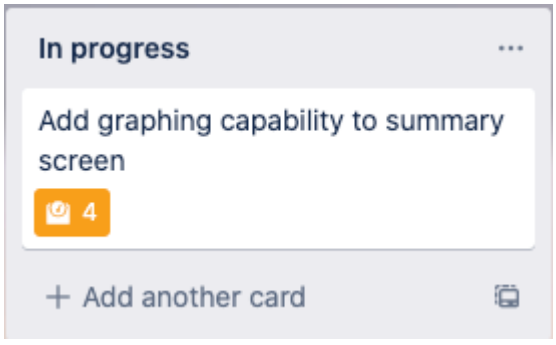
Different environment

Just to check, I initiated another environment with a traitValue of 1 to see if the population and graph responded as expected. This population became very focused around a very low traitValue.

This graph works!

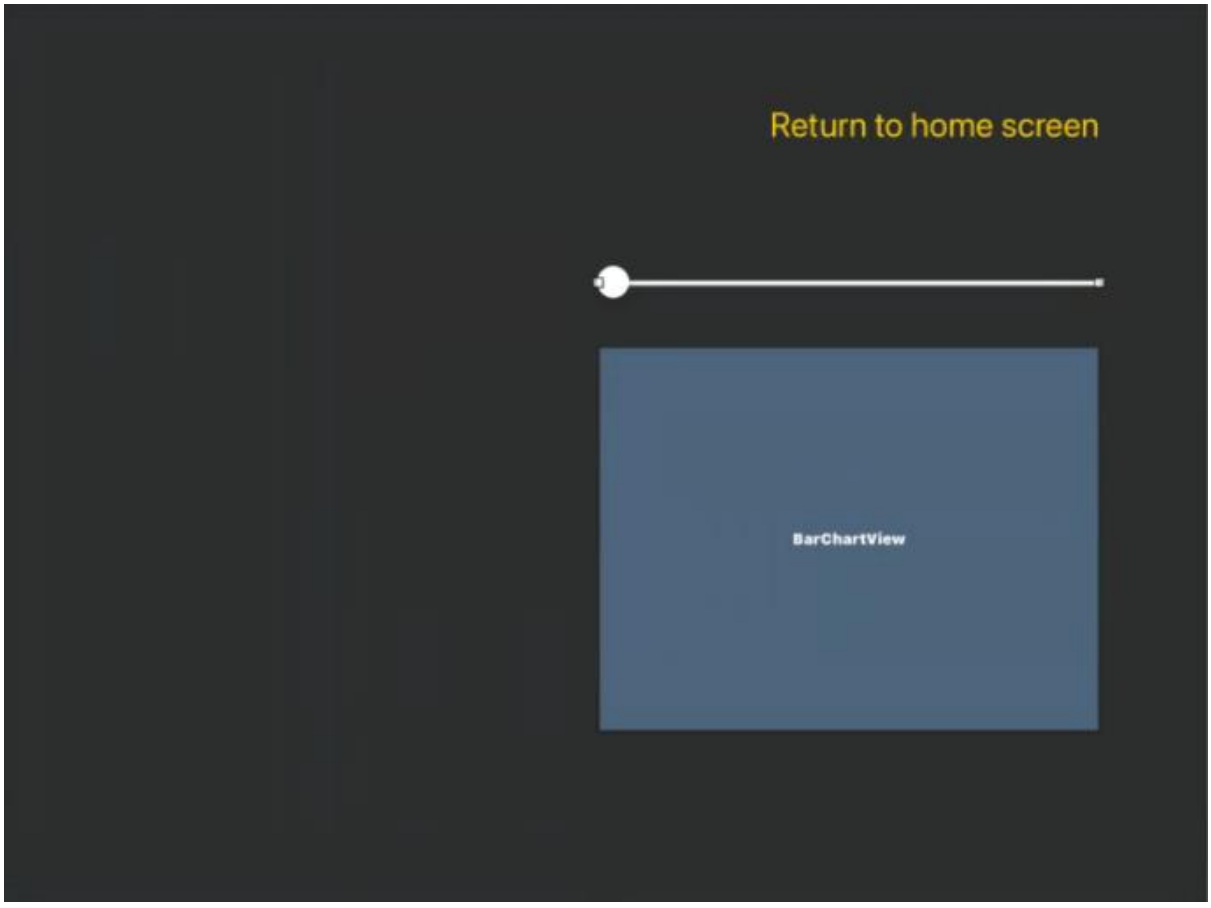
S3 T3-4: Add interactive graphs to summary screen

So far, I have done no work on the summary screen except to add a button that returns the user to the home screen. I wanted to make this screen useful and more educational as suggested in the second sprint's stakeholder feedback, and to meet success criterion 13. I therefore started work on an interactive graph in the summary screen, similar to the graphs in the "Improved Evolution Simulator" found in the analysis section on page 7.



I wanted the first graph to be a bar chart similar to the one in the simulation page, but where the user can select each generation's population using a slider. The first thing I needed to add was a BarChartView inside the SimulationSummaryViewController.

I then added a slider that would select the generation to be displayed.



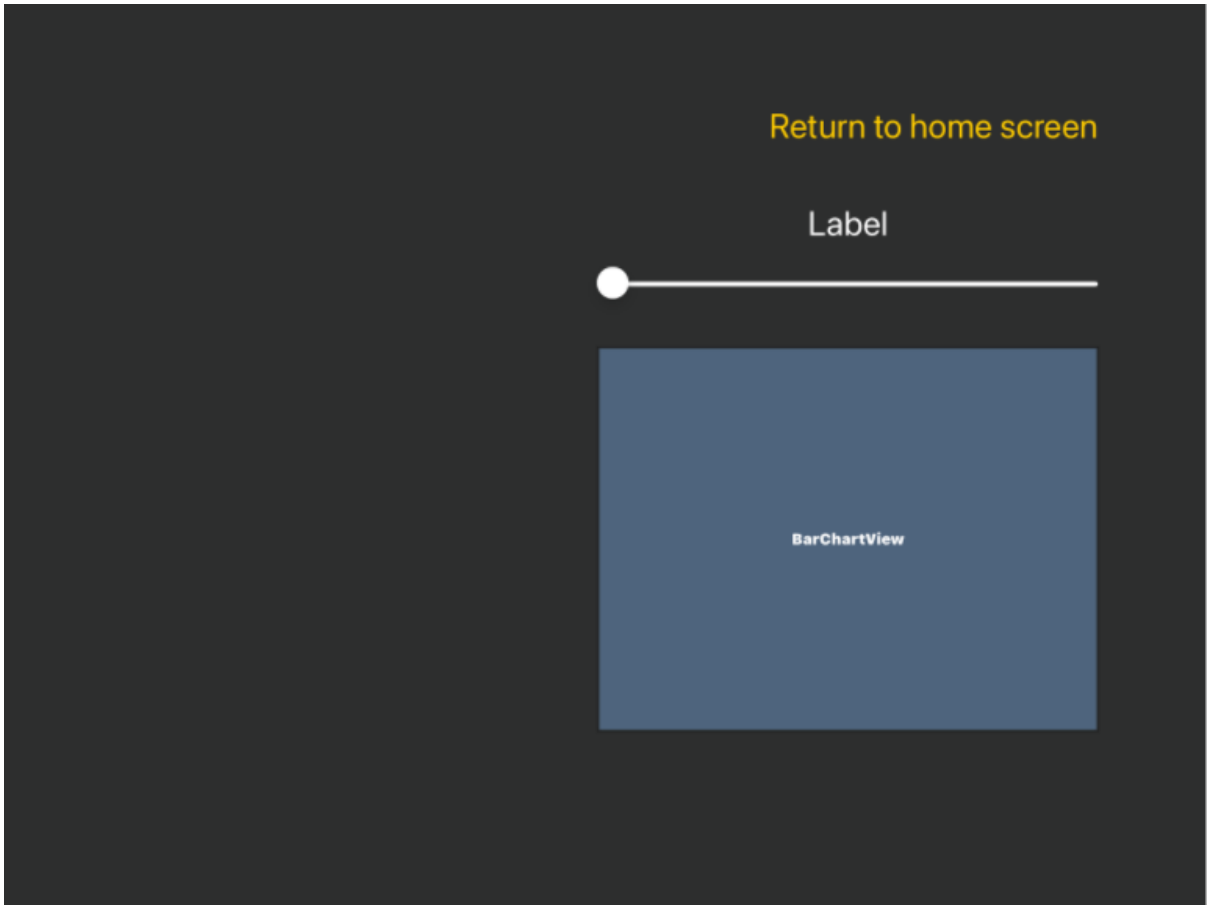
I then used similar logic to the previous bar chart, but this time the generation being displayed is not the current generation in the simulation, but the generation selected by the user. For this to work, the bounds of the slider must start at 1 and end at the number of generations progressed through by the user. This was easy to implement.

Code

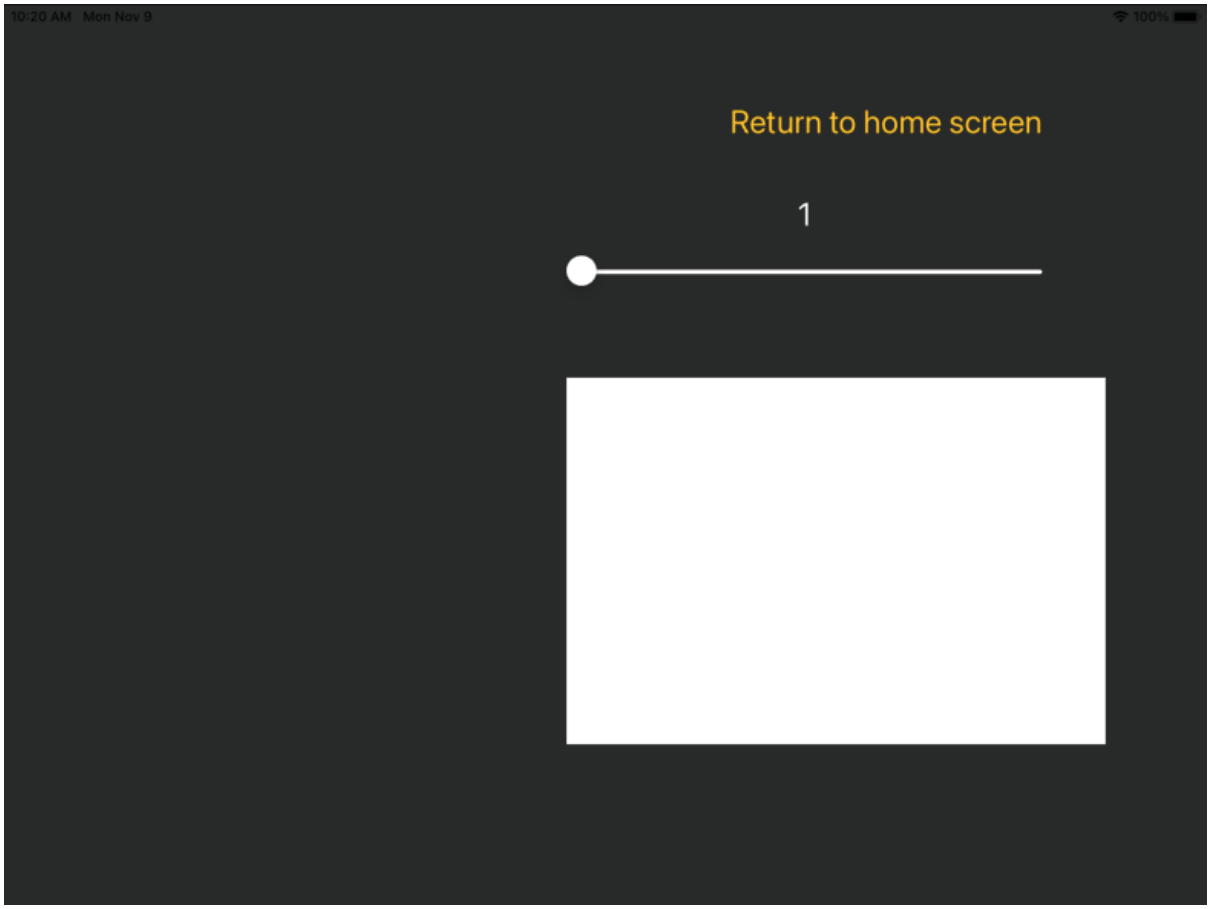
```
override func viewDidLoad() {
    super.viewDidLoad()

    generationSelector.minimumValue = 1
    generationSelector.maximumValue = Float(environment!.populationHistory.count)
    setChart(generation: 1)
}
```

Test

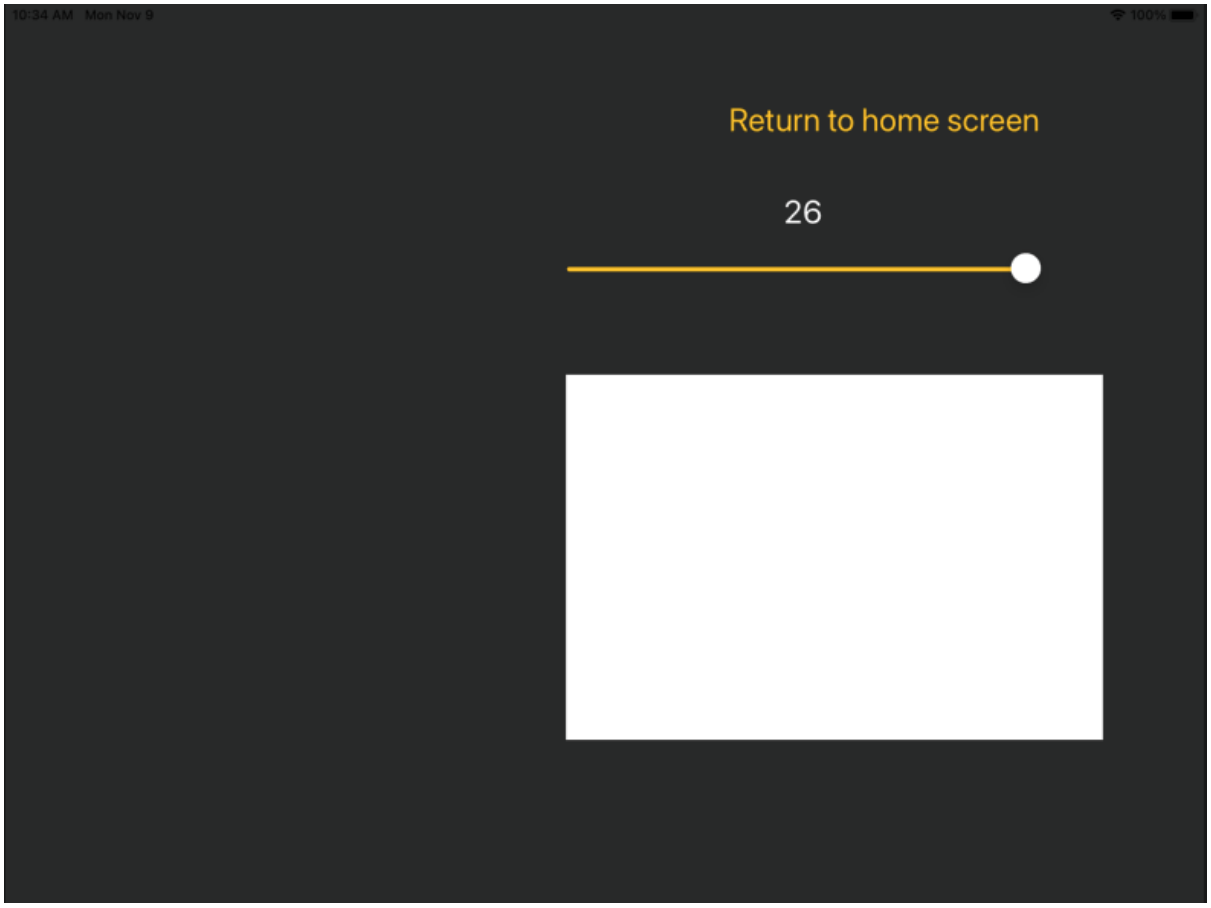
To test the slider bounds, I added a label that displays the selected generation number.



Running the app, this is what the user sees.



Selecting the maximum generation also works correctly.



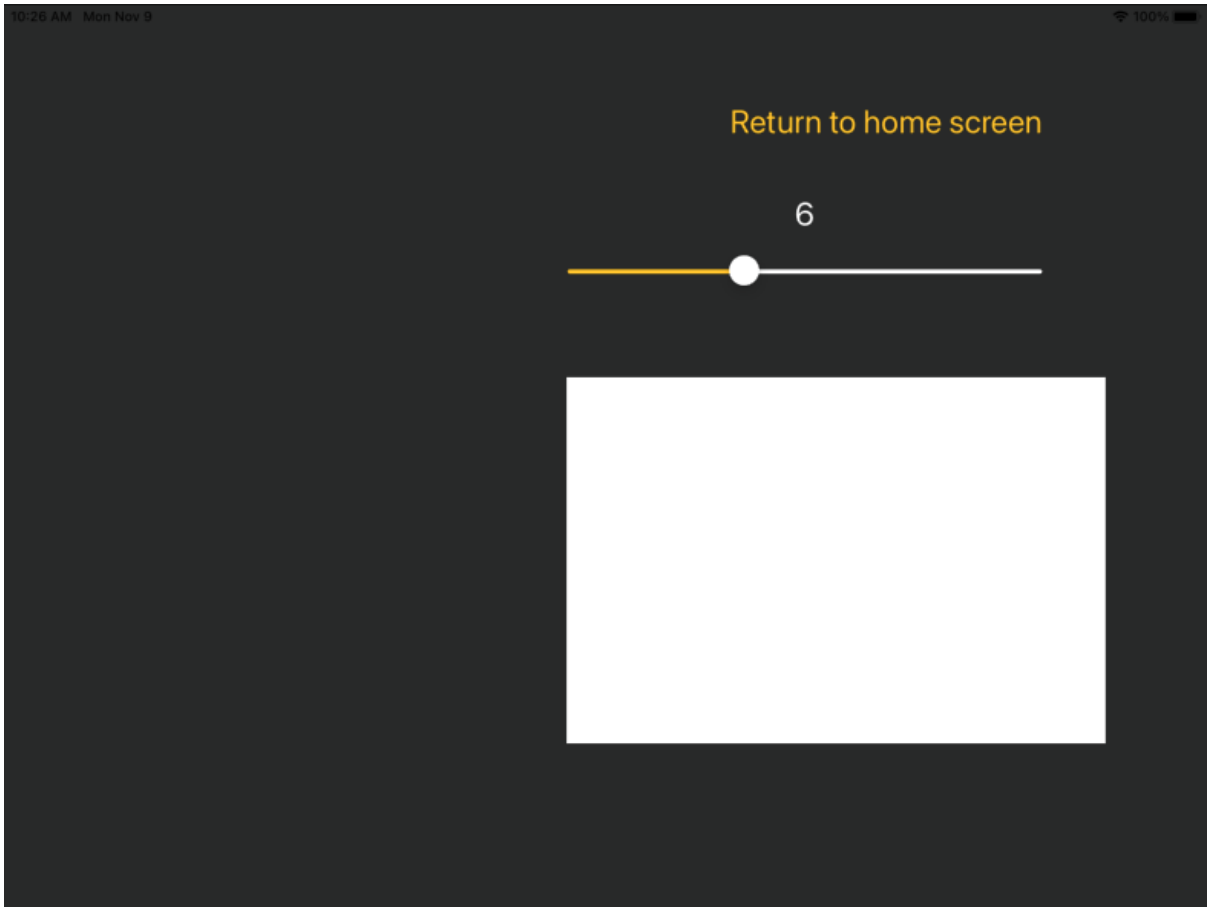
For now, the bar chart is plain white because there is no data available for it to display. To access the data the bar chart needs and use it in a format the bar chart recognises, I used similar code to the previous bar chart.

Code

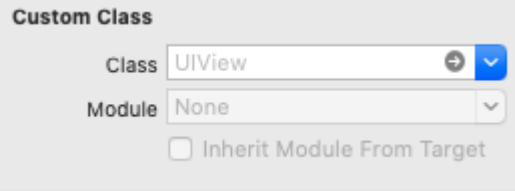
```
func setChart(generation: Int) {

    displayGenerationNumber.text = String(generation)

    var yValues = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
    var dataEntries: [BarChartDataEntry] = []

    for creature in environment!.populationHistory[generation-1] {
        if 1...10 ~= creature.traitValue {
            yValues[0] += 1
        } else if 11...20 ~= creature.traitValue {
            yValues[1] += 1
        } else if 21...30 ~= creature.traitValue {
            yValues[2] += 1
        } else if 31...40 ~= creature.traitValue {
            yValues[3] += 1
        } else if 41...50 ~= creature.traitValue {
            yValues[4] += 1
        } else if 51...60 ~= creature.traitValue {
            yValues[5] += 1
        } else if 61...70 ~= creature.traitValue {
            yValues[6] += 1
        } else if 71...80 ~= creature.traitValue {
            yValues[7] += 1
        } else if 81...90 ~= creature.traitValue {
            yValues[8] += 1
        } else if 91...100 ~= creature.traitValue {
            yValues[9] += 1
        }
    }

    for i in 0 ..< xValues.count {
        let dataEntry = BarChartDataEntry(x: Double(i), y: Double(yValues[i]))
        dataEntries.append(dataEntry)
    }

    let chartDataSet = BarChartDataSet(dataEntries)
    let chartData = BarChartData(dataSet: chartDataSet)
    simulationSummaryBarChartView.data = chartData

}
```

The “generation” parameter is new in this block of code, and allows the user to scroll through each generation and see the distribution of creatures in each one.

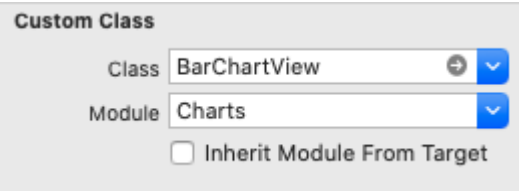
Test

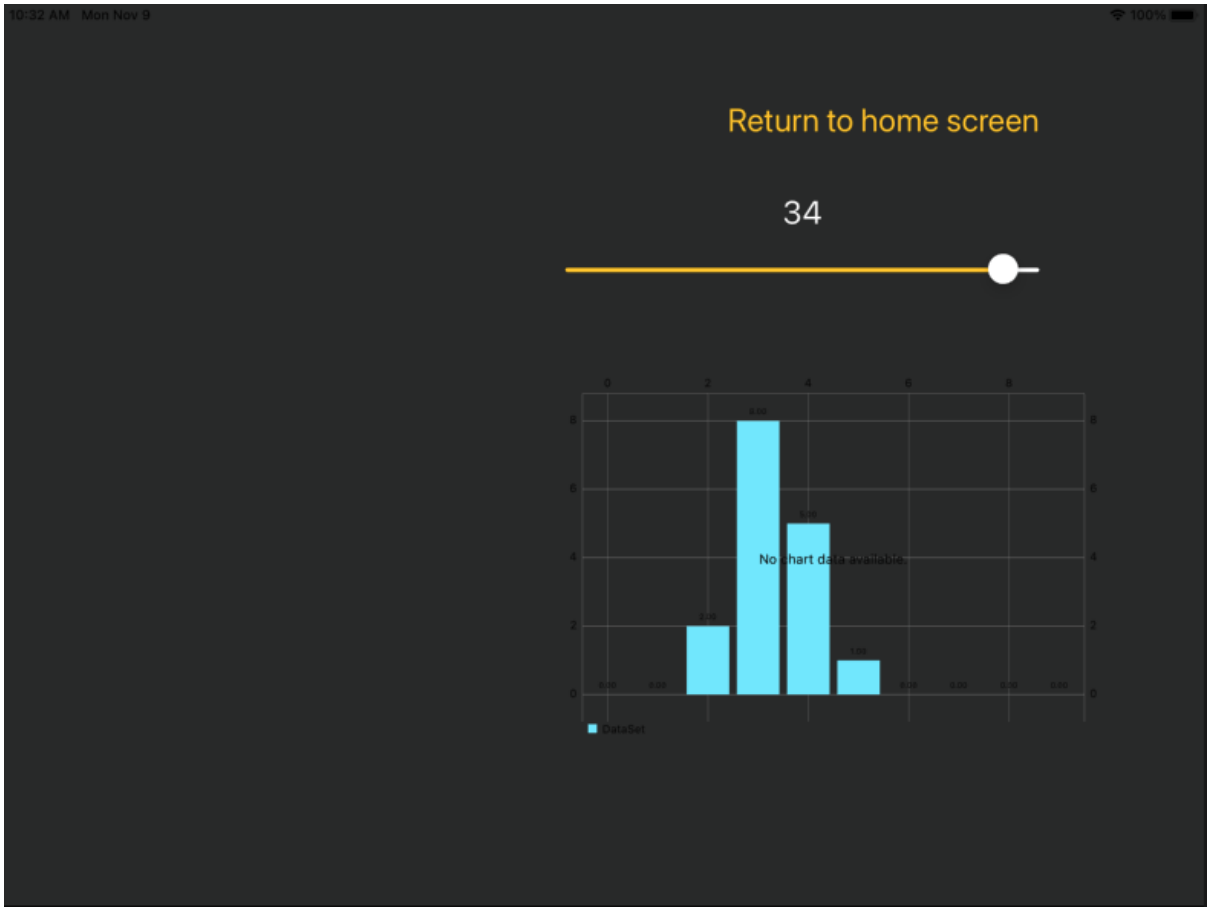


This test failed. The bar chart is still plain white, even though it should have access to the populationHistory data. I carefully looked over the code to see where I had gone wrong, but couldn't find anything. Eventually, I realised the embedded BarChartView was not actually a BarChartView, but was instead a UIView that can't display graphs at all.



Once I changed this to a BarChartView, it worked immediately.





This test succeeded, as the graph now displays the selected generation's population distribution. This was very frustrating, but I'm also very glad it now works!

S3 T5: Allow user to customise reproductive chance

The user should be able to customise the chance a creature reproduces each generation so that they can see the impact this might have on a population in the real world, as per success criterion 6. To allow the user to do this, I first needed to add a slider and label linked to the ParameterInputViewController.

I linked the slider and label so that changing the slider changes the displayed value in the label.

Code

```
@IBAction func reproductiveChanceChanged(_ sender: UISlider) {
    let value = sender.value
    reproductiveChanceLabel.text = String(value)
}
```

Test

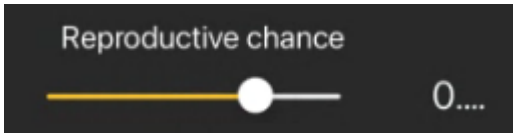
To test this connection, I ran the app in XCode's device simulator and tried changing the value of the slider.

This test failed. I thought the number wasn't displaying correctly because there were too many decimal places for it to display, so I tried rounding it to 2 decimal places.

Updated code

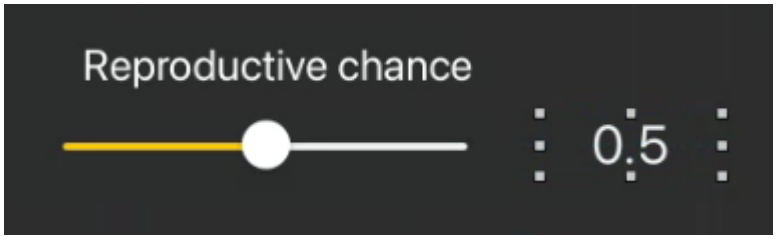
```
@IBAction func reproductiveChanceChanged(_ sender: UISlider) {
    let value = Double(sender.value)
    let roundedValue = round(value * 100)/100
    reproductiveChanceLabel.text = String(roundedValue)
}
```

Test



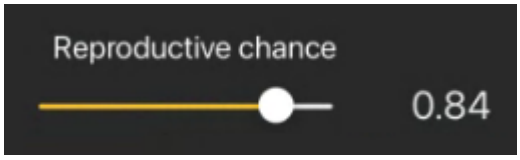
This test also failed, as the number still wasn't displaying correctly. I then tried increasing the size of the label to see if this was the problem.

Updated UI



The label is now much wider than it was.

Test



This test succeeded! The number now displays as I wanted it to, to 2 decimal places.

Now, to actually complete the ticket I need to link this to my simulation. I need to create a new variable within the ParameterInputViewController that holds the selected value of reproductive chance, and then pass this variable into the environment initialiser.

Code

```
var reproductiveChance: Double = 0.5
@IBOutlet weak var reproductiveChanceSlider: UISlider!
@IBOutlet weak var reproductiveChanceLabel: UILabel!
```

```
@IBAction func reproductiveChanceChanged(_ sender: UISlider) {
    let value = Double(sender.value)
    let roundedValue = round(value * 100)/100
    self.reproductiveChance = value
    reproductiveChanceLabel.text = String(roundedValue)
}
```

```
let environment = Environment(environmentColour: 40, reproductiveChance: self.reproductiveChance, selectionPressure: 0.1,
    populationSize: self.populationSize)
```

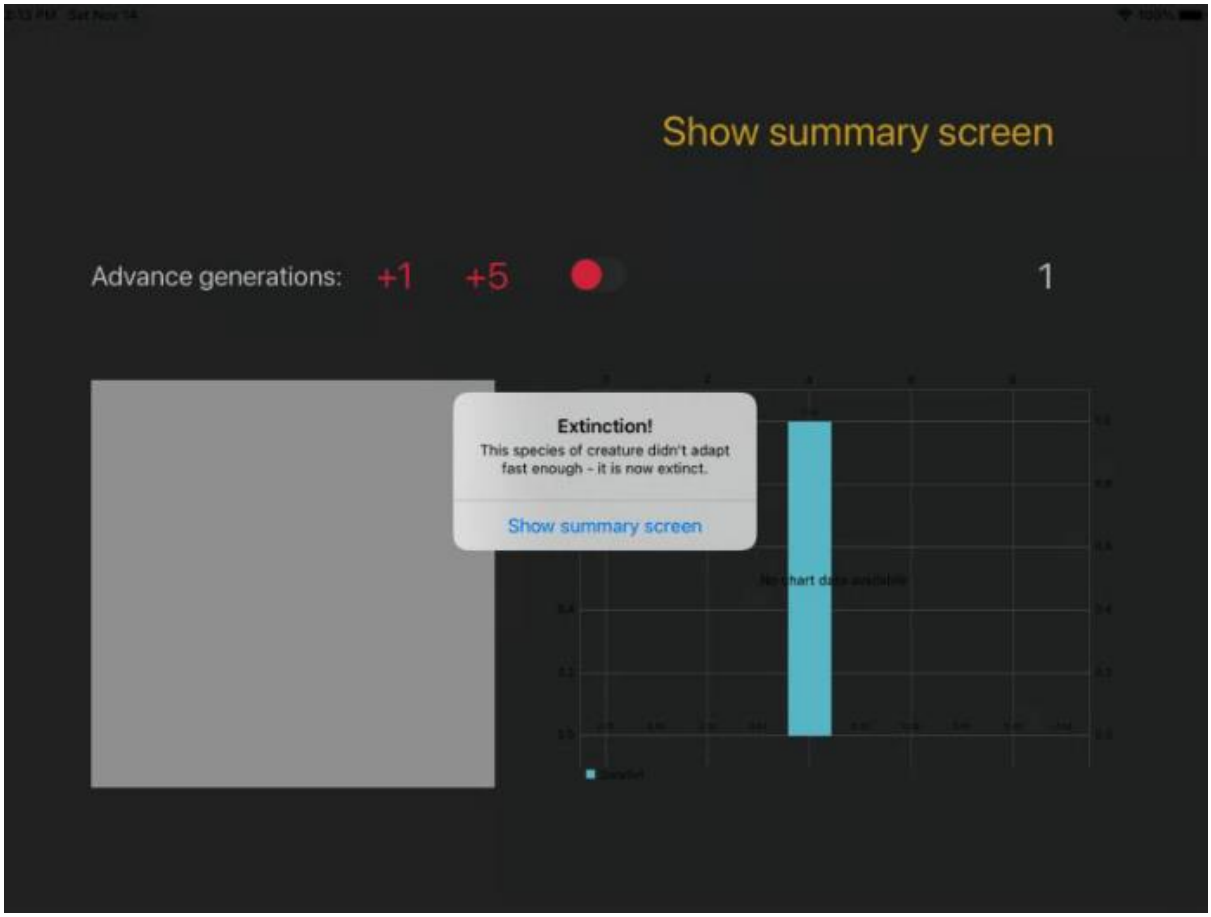
The top block of code is the new variable, the second block updates the variable whenever the slider changes, and the third block passes the variable into the environment initialiser.

Test

To test this new code, I ran the simulation twice: once with a reproductiveChance of 0, and once with a reproductiveChance of 1. When the value is 0, I never expect the creatures to reproduce and the population should rapidly fall to 0. When the reproductiveChance is 1, the creatures should reproduce

every generation and the size of the population should grow until it is capped by the damping factor I introduced in S2 T5 on page 53.

Value of 0:



After only 3 generations, the population size had dropped from 20 to 0.

Value of 1:

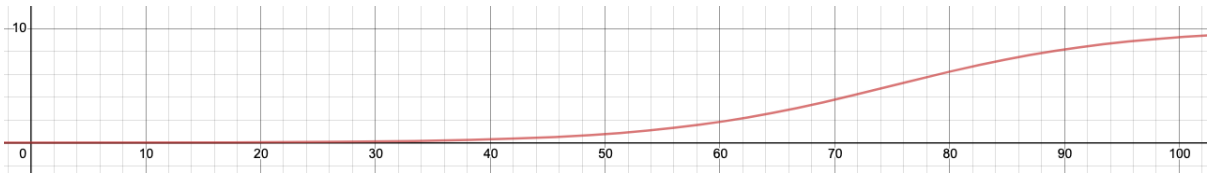
After advancing 15 generations, the population had levelled off at a population size of about 30, up from 20. This is what we expected to see, so the test succeeded.

S3 T6: Add slider and label to change selection pressure

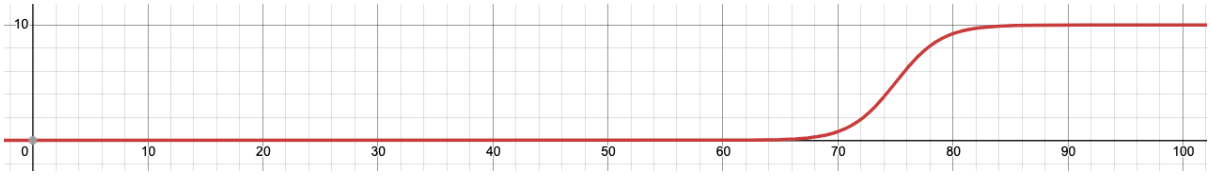
To allow the user to further customise the simulation and see how certain changes would affect a real population, I wanted to allow the user to customise how punishing an environment is. That is to say, how likely a creature is to die if it is poorly adapted to the environment. A less punishing environment should lead to a wider distribution of creature traits, and a more punishing environment should lead to a very narrow distribution. This is part of success criterion 6.

To make an environment more punishing, the kill function should have a steeper central gradient.

Less punishing:



More punishing:



The difference between these graphs is a single variable.

$$f(x) = \frac{L}{1 + e^{-k(x-x_0)}},$$

When k = 0.1, you get the top graph. When k = 0.5, you get the lower graph. When the user customises how punishing the environment is, they will be changing this value.

To implement this, I first have to include a slider that has sensible upper and lower bounds. For now, I will use a lower bound of k = 0.025 and an upper bound of k = 0.3, which I will change later if they do not produce interesting (i.e. long-lasting and varied) simulations.

First, I will add a slider and label to the parameter input screen.

This slider has a default value of 0.15 (the value I have been using so far), a minimum value of 0.025, and a maximum value of 0.3. This ticket is therefore complete.

S3 T7: Link slider and label to simulation

To make the slider I just added useful, I must connect it to the parameter input screen and from there to the initialisation of the environment.

Code

```
var selectionPressure: Double = 0.15
@IBOutlet weak var selectionPressureSlider: UISlider!
@IBOutlet weak var selectionPressureLabel: UILabel!

@IBAction func selectionPressureChanged(_ sender: UISlider) {
    let value = Double(sender.value)
    let roundedValue = round(value * 1000)/1000
    self.selectionPressure = value
    selectionPressureLabel.text = String(roundedValue)
}
```

```
@IBAction func showSimulationVC(_ sender: Any) {

    let environment = Environment(environmentColour: 40, reproductiveChance: self.reproductiveChance, selectionPressure:
        self.selectionPressure, populationSize: self.populationSize)

    if let vc = storyboard?.instantiateViewController(identifier: "simulationVC") as? SimulationViewController {
        vc.environment = environment
        vc.modalPresentationStyle = .fullScreen
        present(vc, animated: true, completion: nil)
    }
}
```

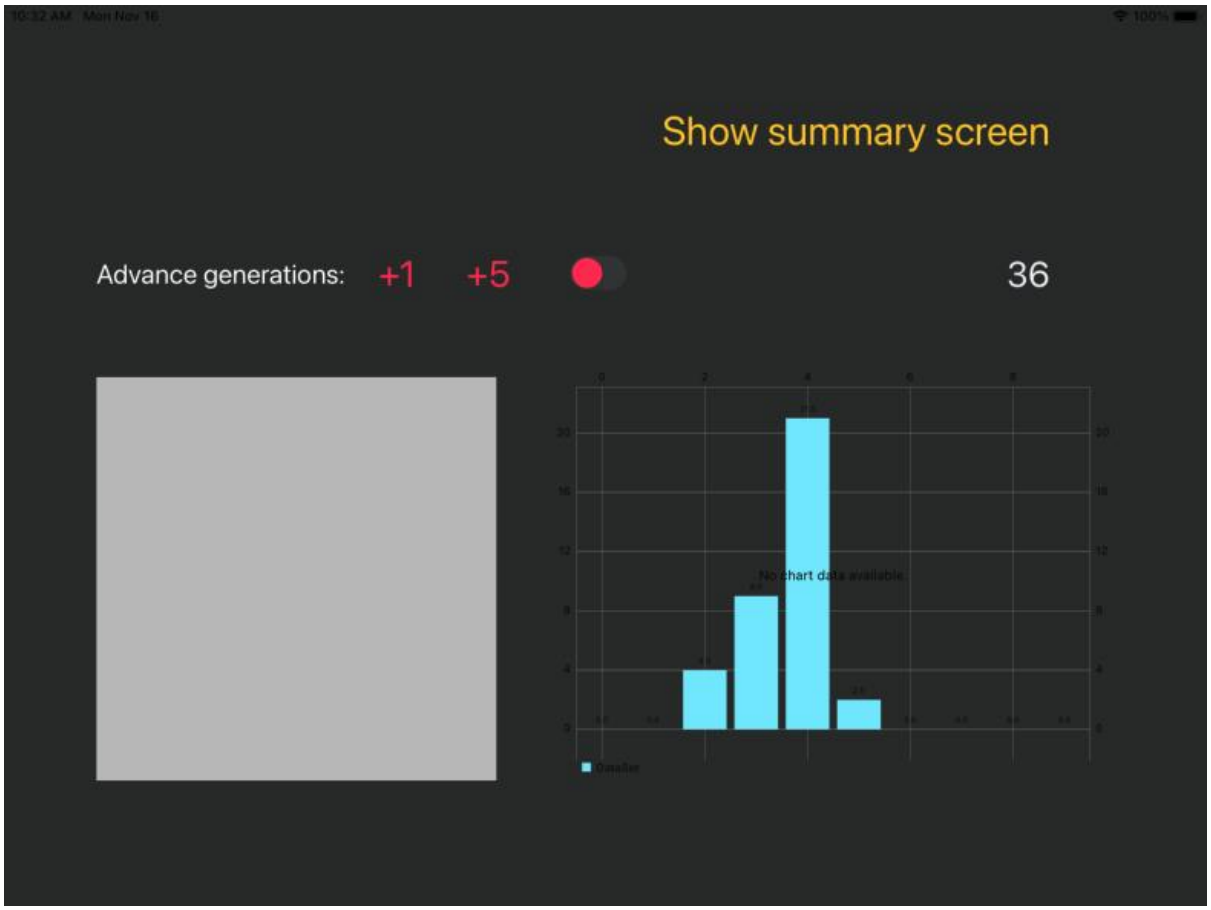
Now, changing the slider should change the variable selectionPressure to the selected value and the text in the label to the value rounded to 3 decimal places.

Test

Using a high value of selectionPressure should give me a narrow distribution of creature traits displayed in my graph. Using a low value should give me a wide distribution.

Low value (expecting wide distribution):


Start
Population size
20
Reproductive chance
1.0
Selection pressure
0.025


Show summary screen
Advance generations:
+1
+5
36
No chart data available

High value (expecting narrow distribution):

There was no discernible difference between the two distributions, so this test failed. I have just realised (literally while writing this) that although I changed the selectionPressure of the environment, I don't actually *use* the variable selectionPressure anywhere in my code. I will therefore update my kill function to use the value of selectionPressure I passed in rather than the 0.15 I hard-coded earlier.

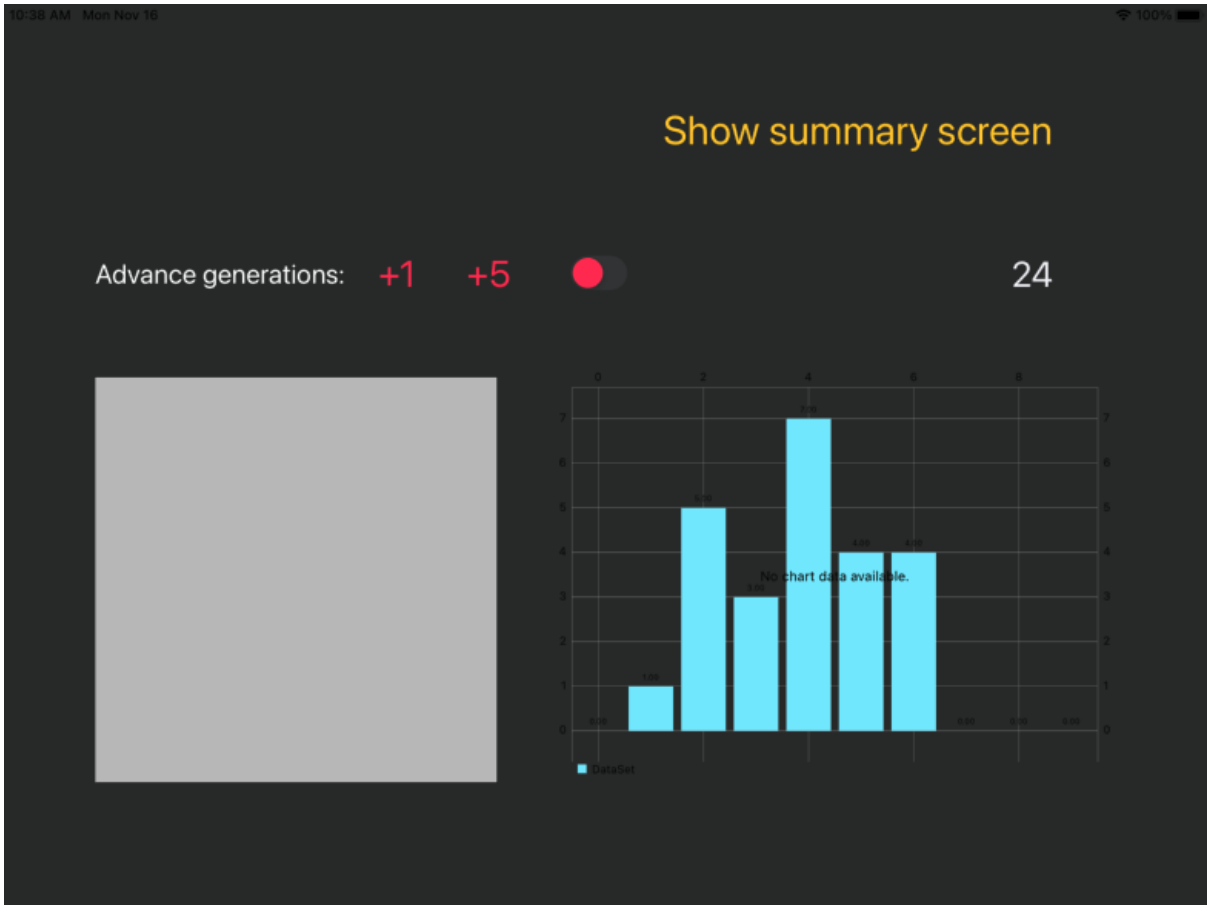
Updated code

```
func calculateChanceOfDeath(creature: Creature) -> Double {
    let adaptation = Double(calculateCreaturesAdaptation(creature: creature))
    let dampingFactor = 0.66 * Double(self.creatures.count)
    let denominator = 1 + pow(2.71828, -self.selectionPressure * (adaptation - (75 + dampingFactor)))
    let chanceOfSurvival = 1/denominator
    return 1 - chanceOfSurvival
}
```

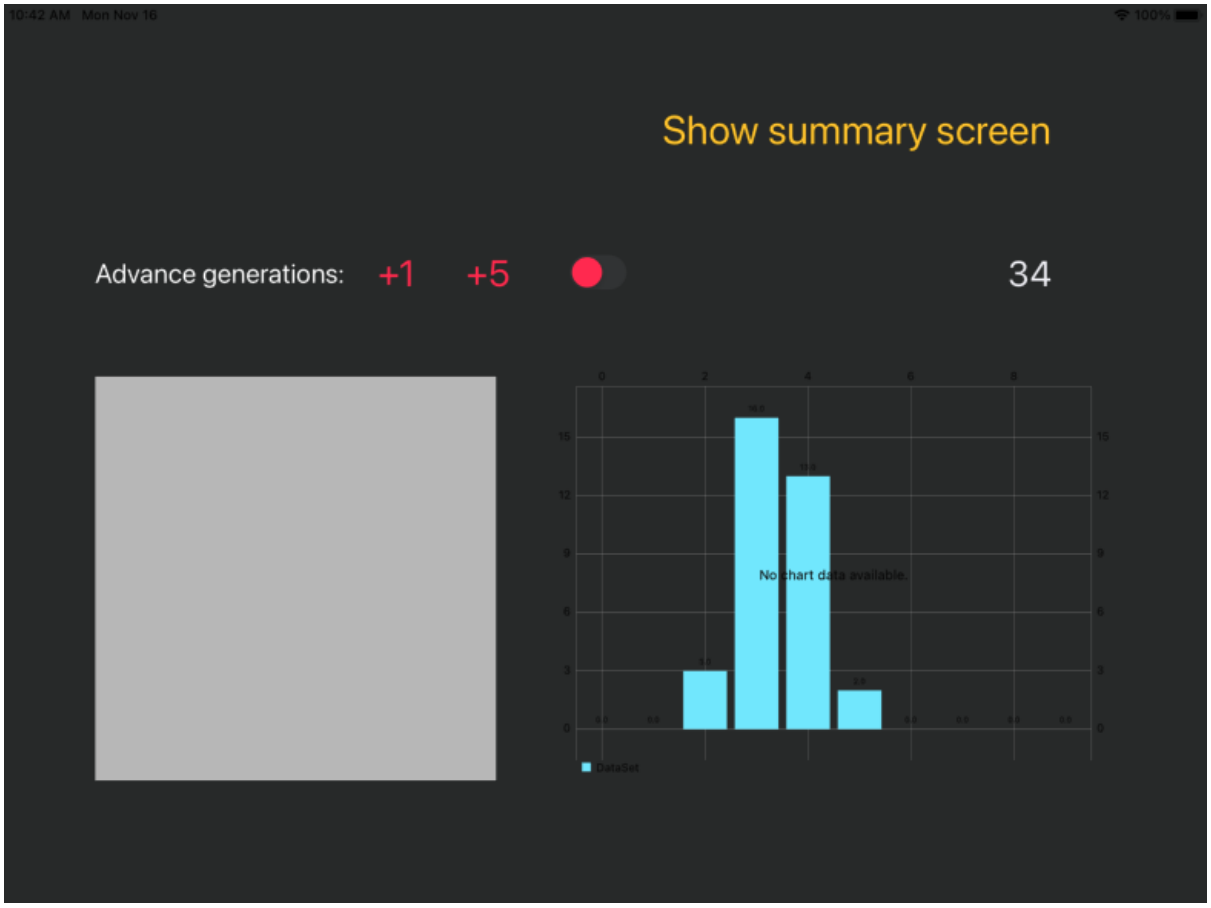
I am now using –self.selectionPressure rather than –0.15.

Test

Low value (wide distribution):



High value (narrow distribution):



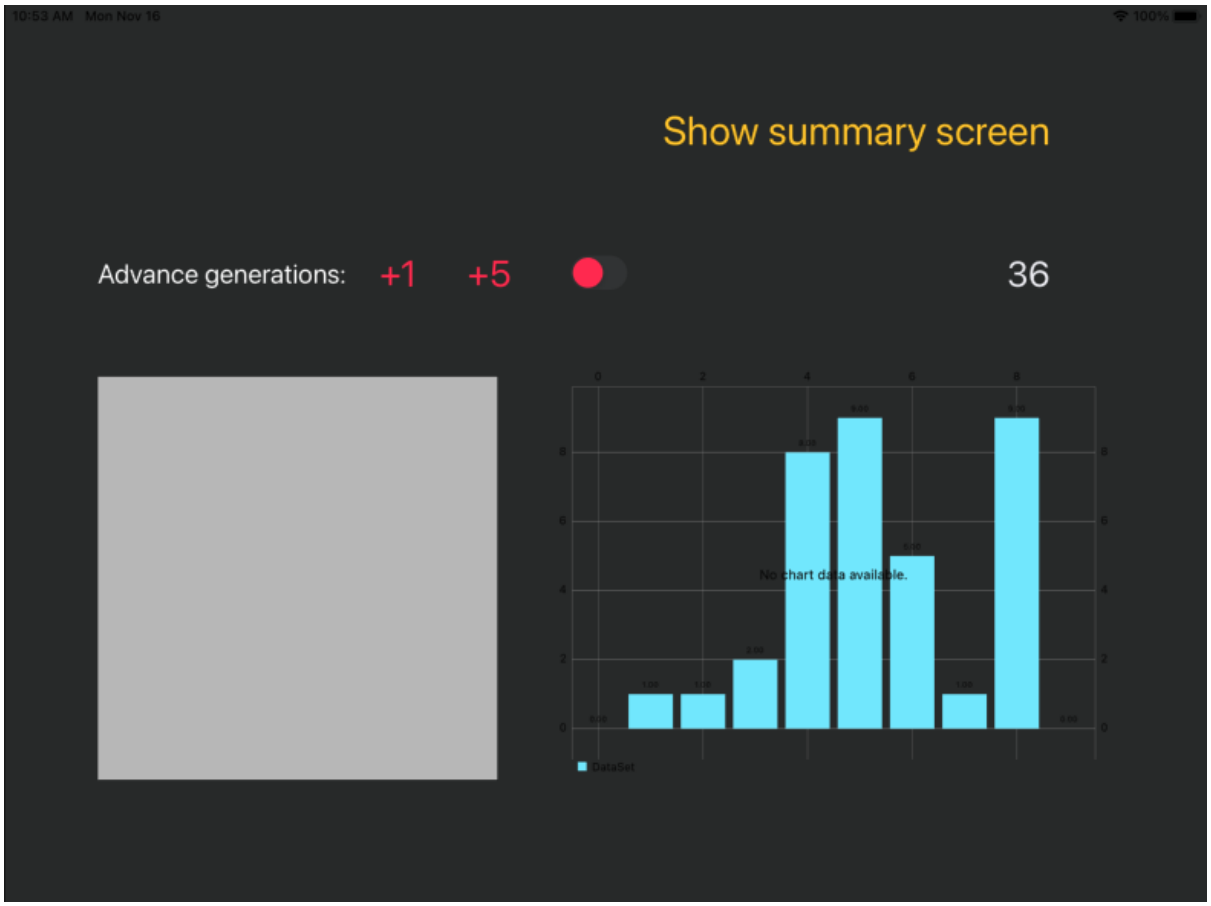
There is a much more clear difference between these distributions, so the test passed. That being said, there is not as large a difference between the two as I would like. I will therefore change the maximum value of the slider to 0.5, and the minimum value of the slider to 0.01. I will also display only the value of selectionPressure to 2 decimal places as 3 is not useful.

Changes


```
@IBAction func selectionPressureChanged(_ sender: UISlider) {
    let value = Double(sender.value)
    let roundedValue = round(value * 100)/100
    self.selectionPressure = value
    selectionPressureLabel.text = String(roundedValue)
}
```

Test

k = 0.01

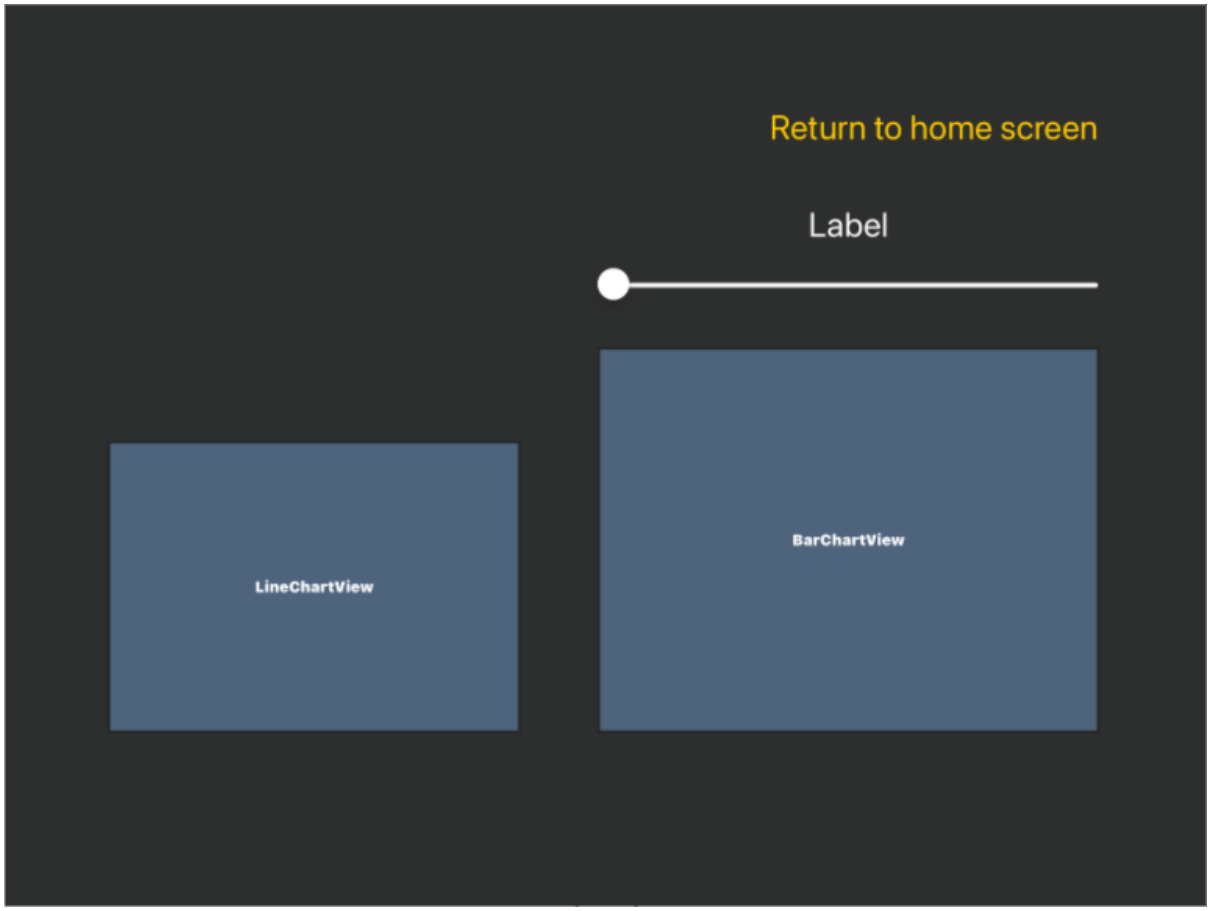


This distribution is *much* wider than before. This is what I was hoping for! This ticket is complete.

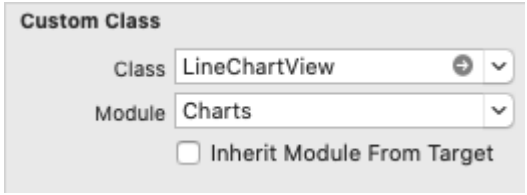
S3 T8: Add line chart of population size over time

To give the user a more intuitive understanding of the population's change over time, I wanted to add a line chart to the simulation summary screen that summarises the changing size of the population over time as per success criterion 13.

The first thing I needed to add was an embedded LineChartView to the SimulationSummaryScreen ViewController.



I made sure its class was of type LineChartView – this is what I didn't do last time and what wasted so much time.



Code

First, I had to link this container view to the SimulationSummaryViewController.

```
// For line chart
@IBOutlet weak var simulationSummaryLineChartView: LineChartView!
```

Next, I had to set the x-values of this line chart to the generation numbers and the y-values to the size of that generation's population.

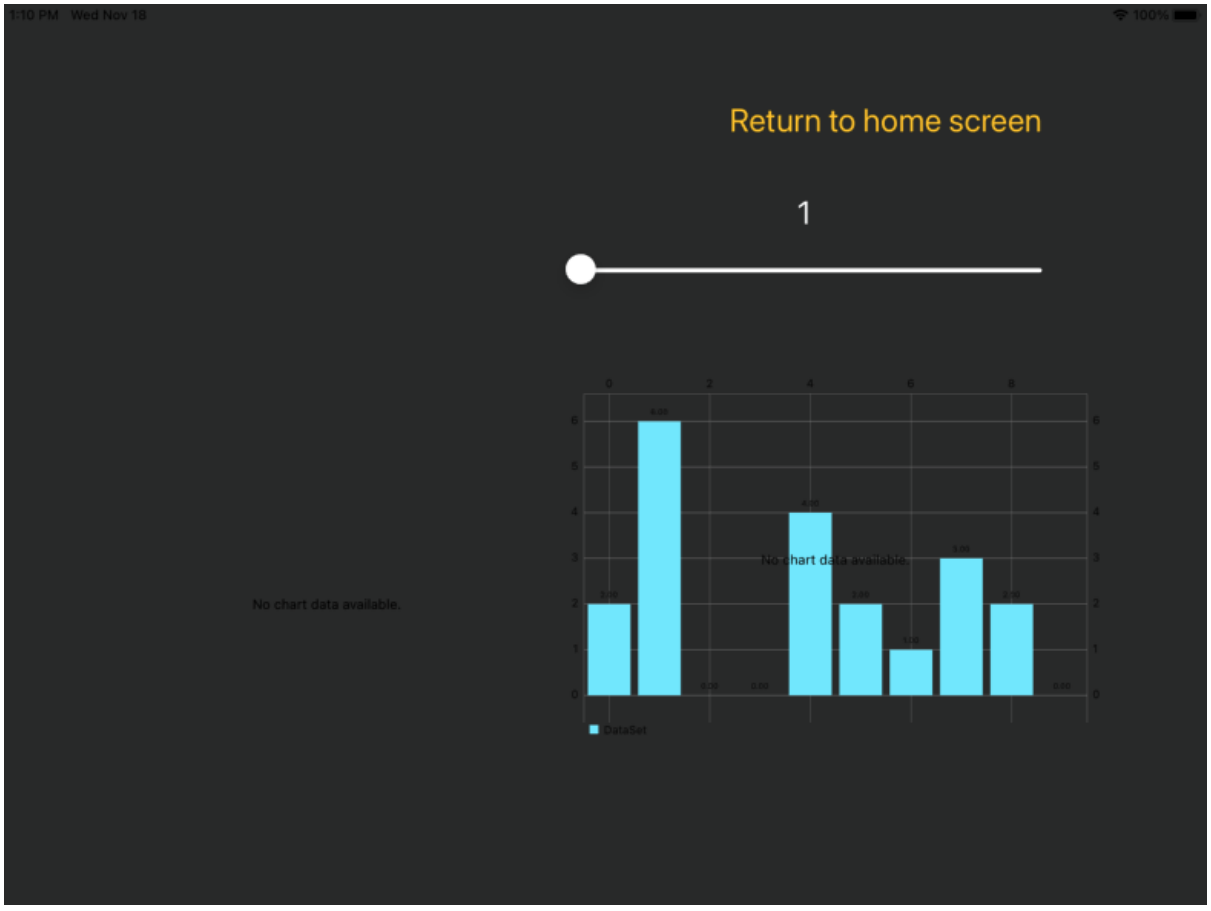
```
var xValues: [Int] = []
for i in 1 ... environment!.populationHistory.count {
    xValues.append(i)
}

var yValues: [Int] = []
for i in 0 ..< environment!.populationHistory.count {
    yValues.append(environment!.populationHistory[i].count)
}
```

Finally, I had to use these x- and y-values to populate the line chart.

```
var dataEntries: [ChartDataEntry] = []
for i in 0 ..< xValues.count {
    let dataEntry = ChartDataEntry(x: Double(xValues[i]), y: Double(yValues[i]))
    dataEntries.append(dataEntry)
}

let lineChartDataSet = LineChartDataSet(entries: dataEntries, label: nil)
let lineChartData = LineChartData(dataSet: lineChartDataSet)
simulationSummaryLineChartView.data = lineChartData
```

Test

To test this, I initially just ran the app to see if it displayed correctly.

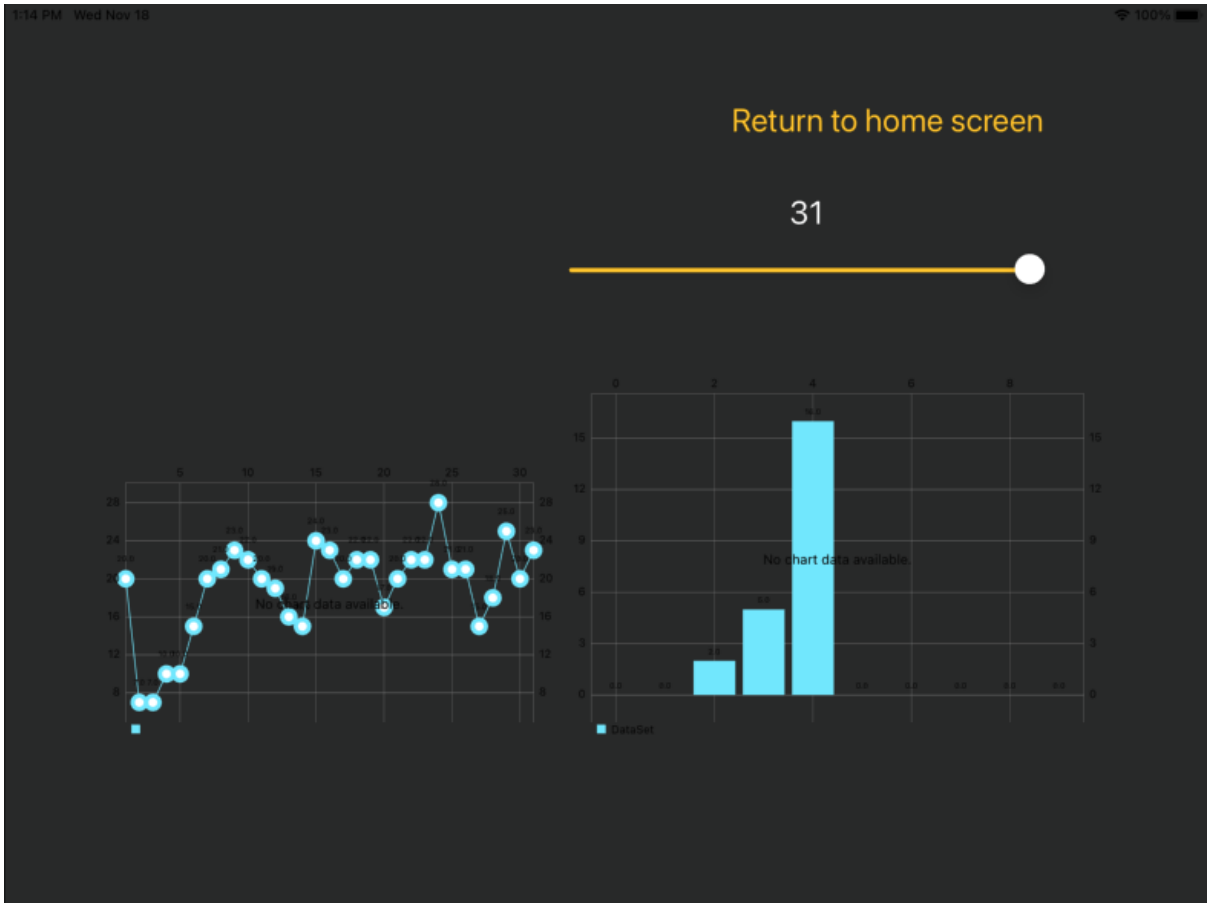


Where I should have had the line chart, I got a blank screen, so this test failed. I realised that I hadn't actually called the setLineChart function so the line chart never had any data added to it. I therefore called this function to the SimulationSummary's viewDidLoad function. This function is called as soon as the view loads, which is what I want.

Updated code

```
override func viewDidLoad() {
    super.viewDidLoad()

    generationSelector.minimumValue = 1
    generationSelector.maximumValue = Float(environment!.populationHistory.count)
    setBarChart(generation: 1)
    setLineChart()
}
```

Test



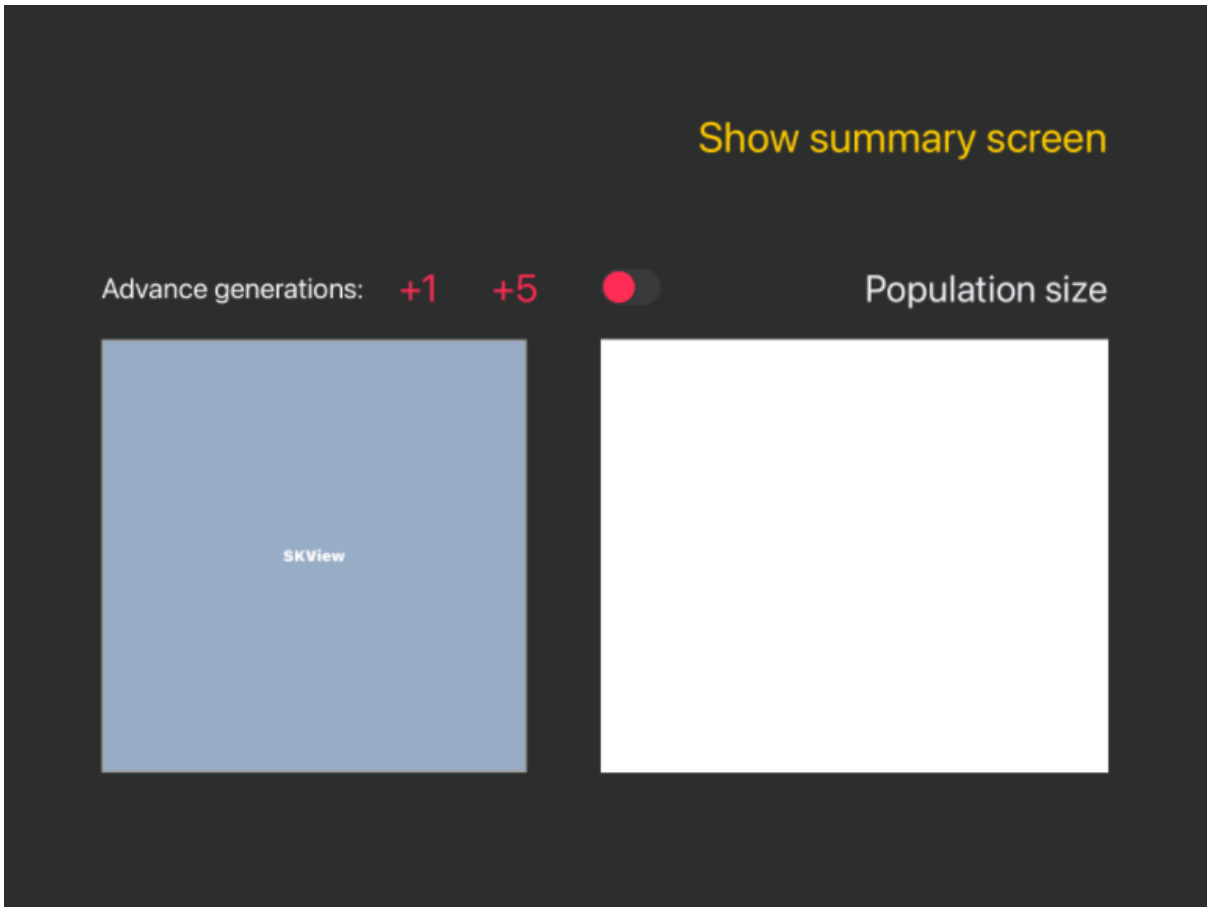
The line chart now loads! There are as many entries as there are generations and the population size fluctuates around 22, which is what I saw in the simulation. This test therefore passed, and the ticket is complete. However, I would like to change how this line chart looks later on if I have time, as right now I don't like the large dots at each data point. I'd also like to add a vertical line that shows which generation the user is looking at in the line chart.

S3 T9: Add SpriteKit view for animations

I wanted animations in my app to create a simple visual description of the population, where each creature is (for instance) a coloured circle, as per success criterion 7. This makes the app more interesting, and more intuitive, which is very important for an app that is supposed to be easy to use.

The first thing I needed was a SpriteKit view embedded in my simulation screen, and to connect that SpriteKit view to the SimulationViewController.

Code




```
// For animations
@IBOutlet weak var skView: SKView!
```

Test

To test that this SKView is correctly connected to the SimulationViewController, I tried to simply change the colour of the view to red, and then run the app.

```
override func viewDidLoad() {
    super.viewDidLoad()

    let scene = SKScene(size: skView.bounds.size)
    scene.backgroundColor = .red
    self.skView.presentScene(scene)
    updateViews()
}
```

The SKView is not red, so this test failed.

Solving this issue took me a very long time. My code was extremely simple and the code seemed to almost exactly match Apple's documentation. I tried moving the scene's anchor point, and looked through a huge number of forums and threads to see if anyone had had this issue. Eventually, I found Apple's documentation on container views, the type of view I was using to load the SKView in the first place.

The problem is that I had misunderstood what a container view does. Instead of using a container view within a parent view to host a different kind of view, it hosts multiple child views within itself. This isn't what I actually wanted, so I removed the container view completely and tried again.

Test

This test succeeded! It was a great feeling to finally get this working. Now I could get to work on the actual animations.

End of sprint review

I am happy with the progress I've made this sprint. Although I failed to complete the last three tickets in the sprint backlog for this sprint, the earlier tickets took much more time than anticipated so my progress through the project has still been as fast as I had hoped.

Stakeholder feedback

Questions:

- What did you like about the app?
- What would you like to see changed?
- What should I focus on moving forward?

G.L.: I really like the direction that the app is progressing in. There is much more data and information being conveyed through the app. There is much more content to gain from the app. I think the biggest thing is the contextualise the data and graphs being presented. I would love to have more labels or pop ups explaining exactly what slider and button is doing. This is intended to be a teaching tool so using the data to create only visuals without any explanation is not useful.

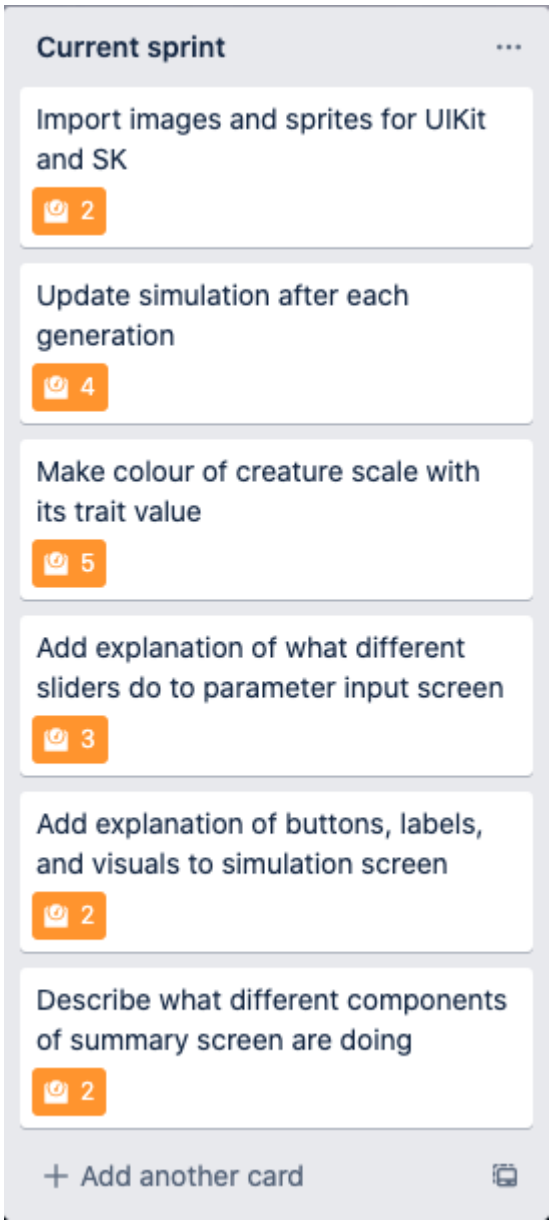
My biology teacher stakeholder is still unavailable due to her being on maternity leave.

Conclusion

It is clear that for this app to be educational, I need to focus more on explaining the different components of the app than just working on functionality. I have therefore added three tickets to my sprint backlog for sprint 4 that are dedicated to adding explanation throughout the app of what each component is doing and explaining how my app overall is simulating evolution.

Sprint 4

Here is the sprint backlog for this sprint:

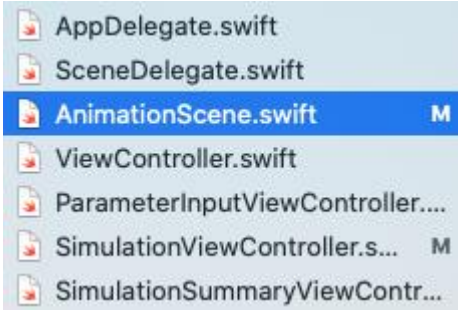


S4 T1: Add simple sprite to SKView

To add animations to my app as specified in success criterion 7, I need to add sprites to the SKScene being loaded into my SKView. First, I wanted to refactor my code so that the SKScene is a separate class.

Refactoring code

I first created a new swift file called AnimationScene that will store the scene.

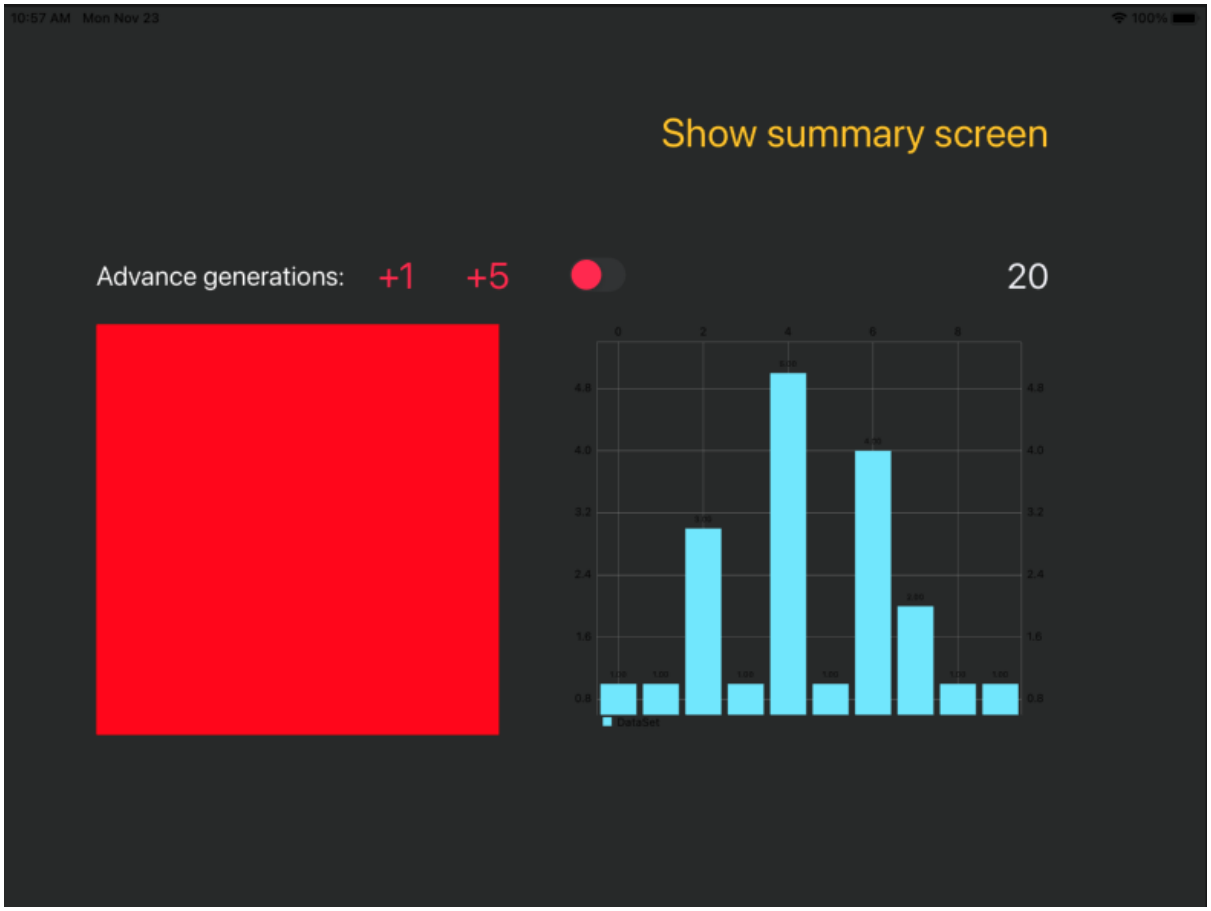


I then set the class to type SKScene, and set the colour of the scene to red to test whether the refactored code works.

```
import UIKit
import SpriteKit

class AnimationScene: SKScene {

    required init?(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)
    }

    override init(size: CGSize) {
        super.init(size: size)

        backgroundColor = .red
    }

}
```

Test

To test this refactored code, I simply need to run the app and check whether the colour of the scene is red.



This test succeeded. I can therefore begin adding nodes (the sprites that will eventually be the creatures) to the AnimationScene.

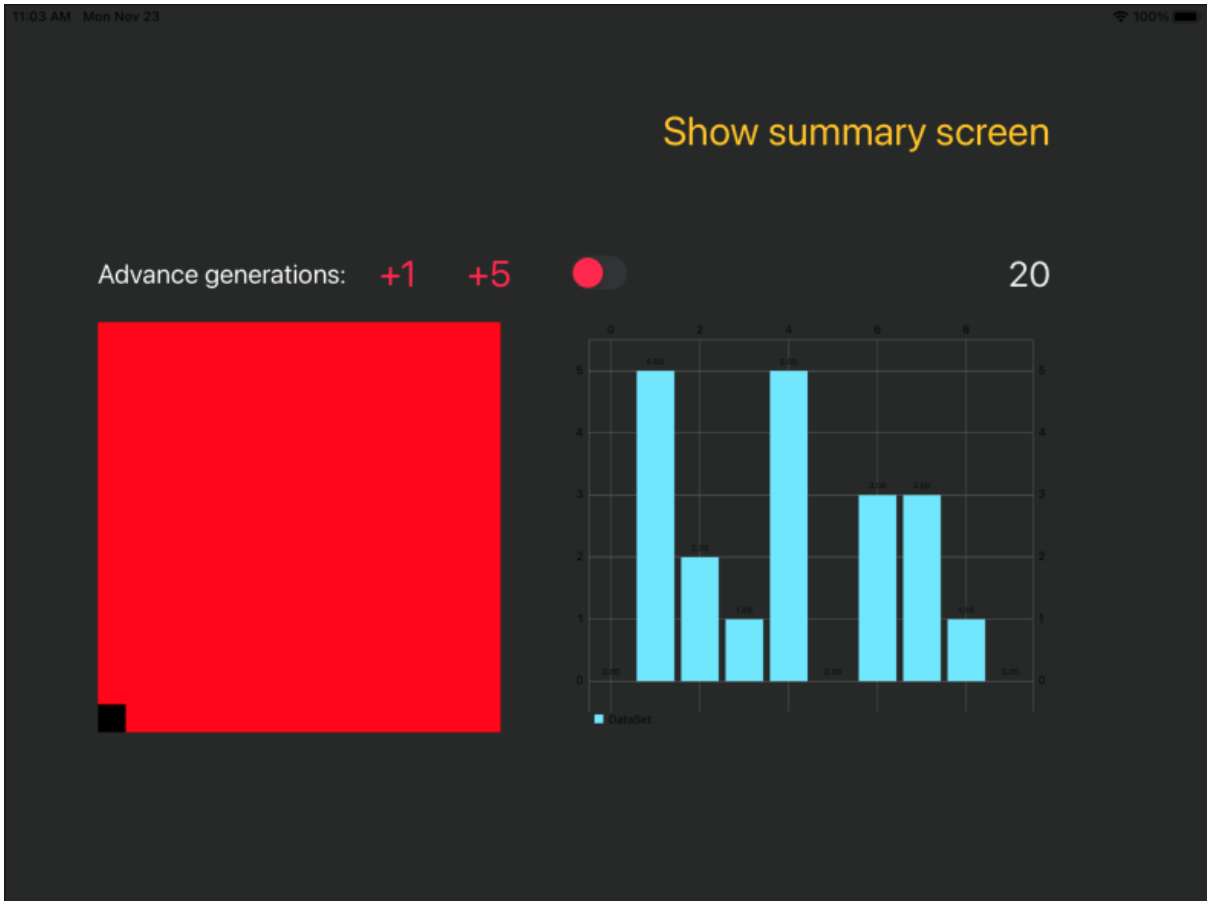
Code

The simplest node I can add is a black square. I will therefore start by trying to add a black square.

```
override init(size: CGSize) {
    super.init(size: size)

    backgroundColor = .red
    let sprite = SKSpriteNode(color: .black, size: CGSize(width: 50, height: 50))
    addChild(sprite)
}
```

Test

Testing whether or not I added a sprite is as simple as running the app and seeing if the sprite is there.



The small black square is there, so this test succeeded.

S4 T2: Randomise position of sprite in SKView

I can't have every single sprite loading into the bottom left corner, stacked on top of one another, so I want to randomise each sprite's positition in each generation. To do this, I first need to be able to randomise the position of each sprite in the SKView. Related to success criterion 7.

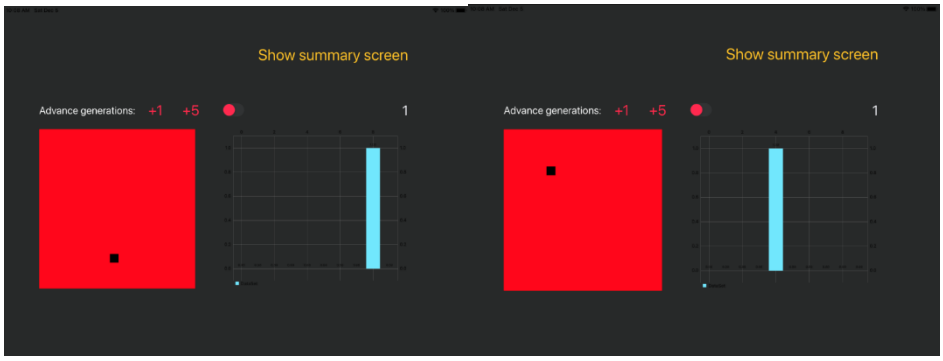
Code

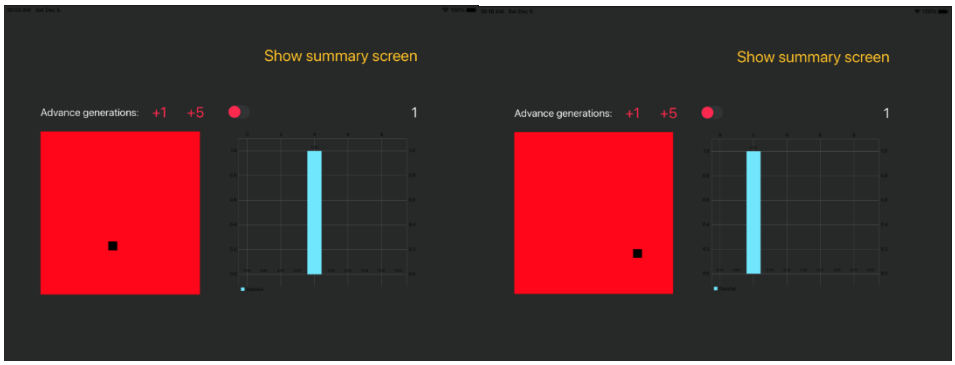
```
func randomSpritePosition() -> CGPoint {
    let xPos = CGFloat(arc4random_uniform(UInt32(self.size.width - 20))) + 10
    let yPos = CGFloat(arc4random_uniform(UInt32(self.size.height - 20))) + 10

    return CGPoint(x: xPos, y: yPos)
}
```

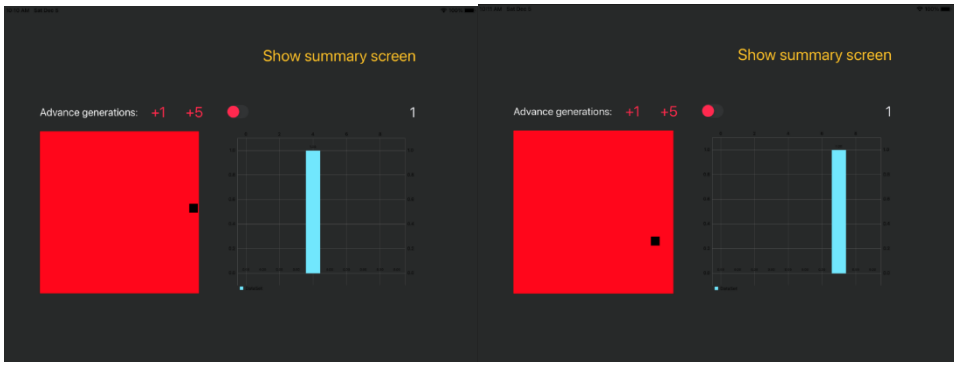
This function returns a random x-y coordinate inside the SKView. The –20 and +10 ensure that the entire sprite is within the view – the centre of the sprite is its coordinate, so placing a sprite right on the edge of the view would mean half the sprite is off screen.

Test

To test this random placement, I ran the app 5 times and checked to see if the sprite was in a different position, but within the view.







Each of these positions is different, but within the view, so this test succeeded.

S4 T3: Display as many sprites as there are creatures

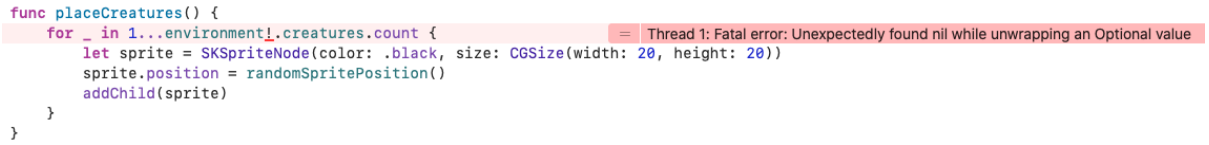
To have this simulation represent the population of creatures, I need the simulation to contain as many sprites as there are creatures in each generation. Linked to success criterion 7.

Code

```
func placeCreatures() {
    for _ in 1...environment!.creatures.count {
        let sprite = SKSpriteNode(color: .black, size: CGSize(width: 20, height: 20))
        sprite.position = randomSpritePosition()
        addChild(sprite)
    }
}
```

Test

I expect this function to place a number of sprites randomly within the SKView. The number of sprites should equal the number of creatures in each generation of the environment. This test will therefore simply run through several generations of creatures and check whether the number of sprites is right, and if they are placed randomly.



Before the view even loaded, the app crashed and I got this error – this test failed. What this tells me is the 'environment' variable has not been set to something other than nil before I try to access it, so trying to count the creatures stored within it creates an error.

It took me two whole days, but I eventually realised my (silly) mistake. I had called placeCreatures() inside the initialiser of AnimationScene, before setting the environment variable of AnimationScene to not nil.

Updated code

First, I removed placeCreatures() from the initialiser:

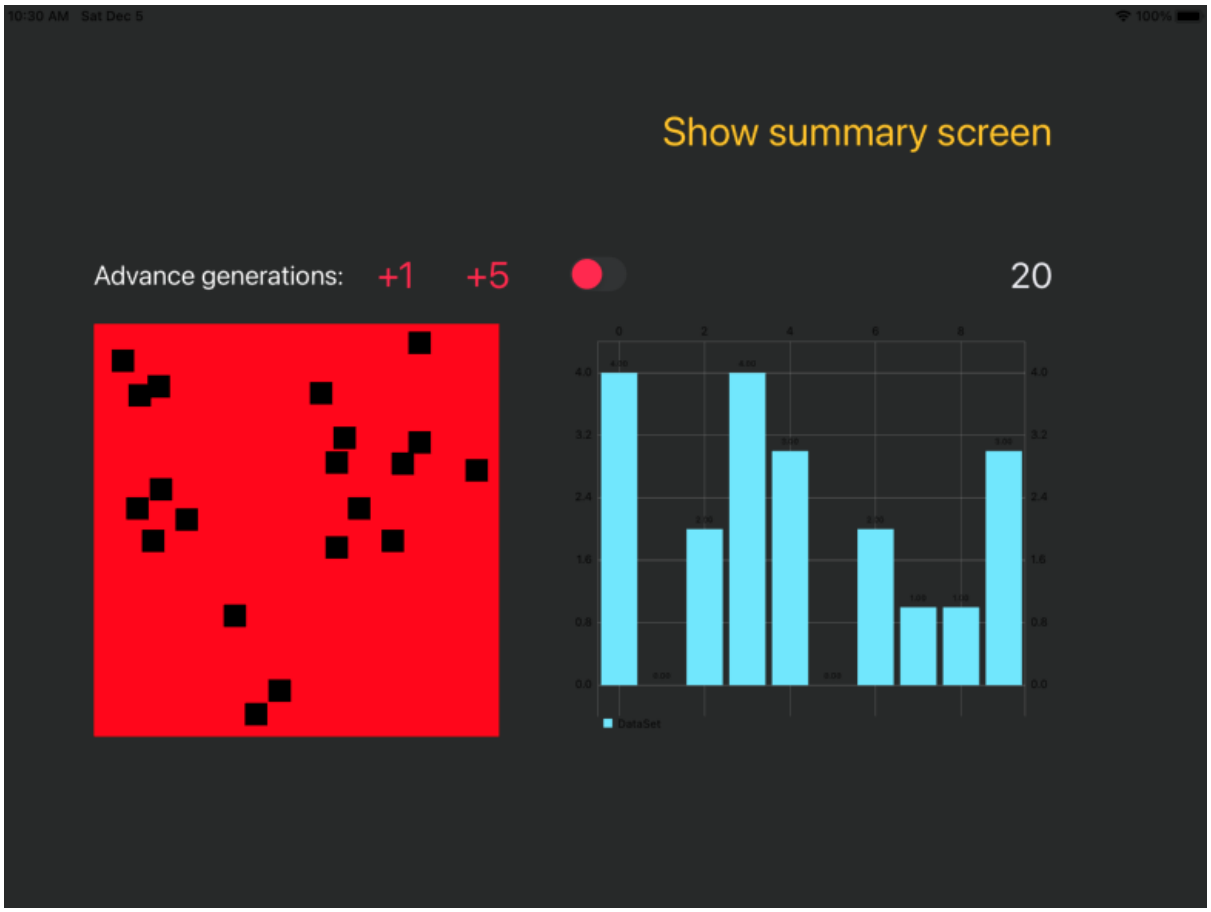
```
override init(size: CGSize) {
    super.init(size: size)

    backgroundColor = .red
}
```

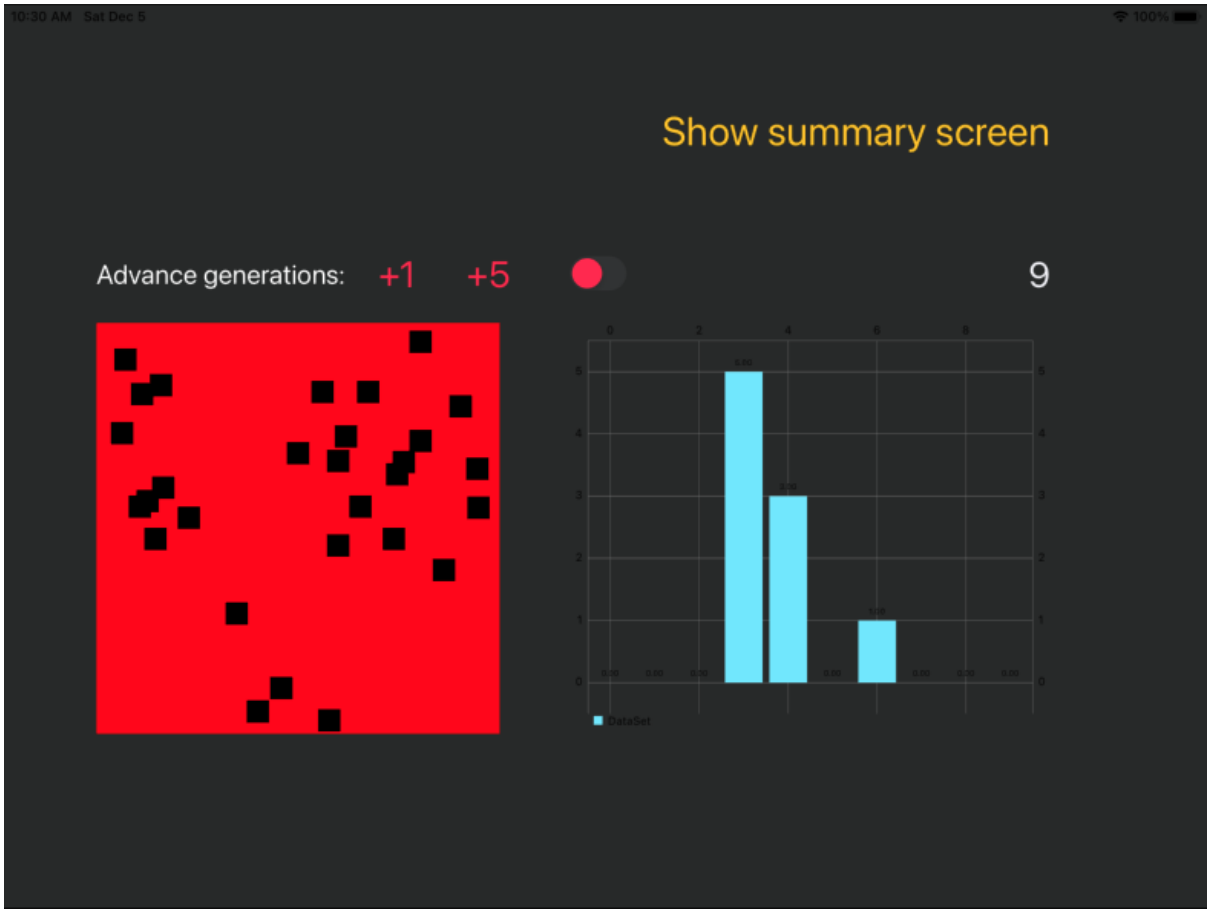
Then, I only called placeCreatures() *after* I had set the scene's environment variable.

```
override func viewDidAppear(_ animated: Bool) {
    scene = AnimationScene(size: skView.bounds.size)
    scene!.environment = environment
    scene!.placeCreatures()
    self.skView.presentScene(scene)
}
```

Test

The test is the same as before. I will advance several generations and check if there are the correct number of sprites for each generation, and that the sprites are placed randomly.





Once again, this test failed. There are more sprites in the second view than the first, even though the number of creatures has gone down. I realised I hadn't deleted the sprites of the last generation before loading in the sprites of the new generation.

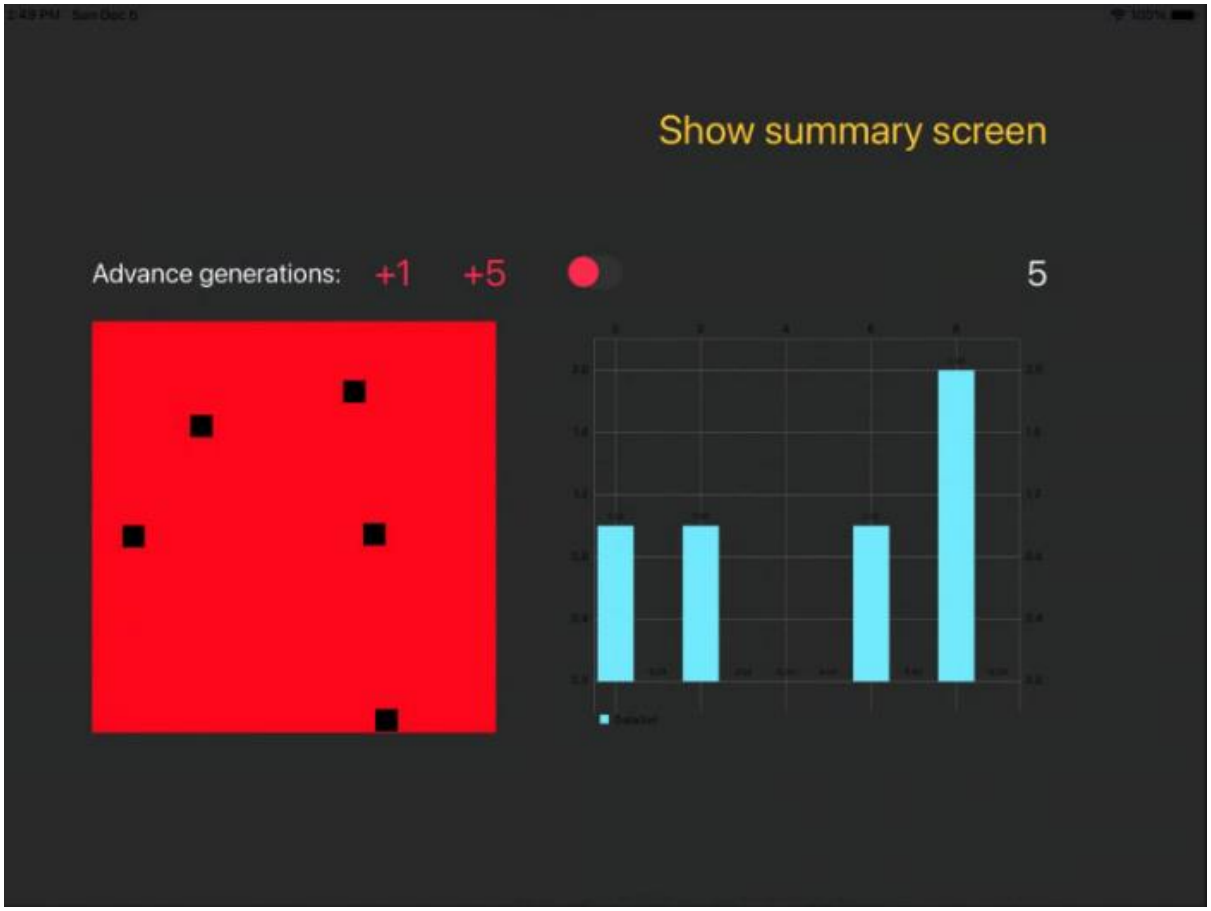
Updated code

```
func placeCreatures() {
    for child in children {
        child.removeFromParent()
    }
    for _ in 1...environment!.creatures.count {
        let sprite = SKSpriteNode(color: .black, size: CGSize(width: 20, height: 20))
        sprite.position = randomSpritePosition()
        addChild(sprite)
    }
}
```

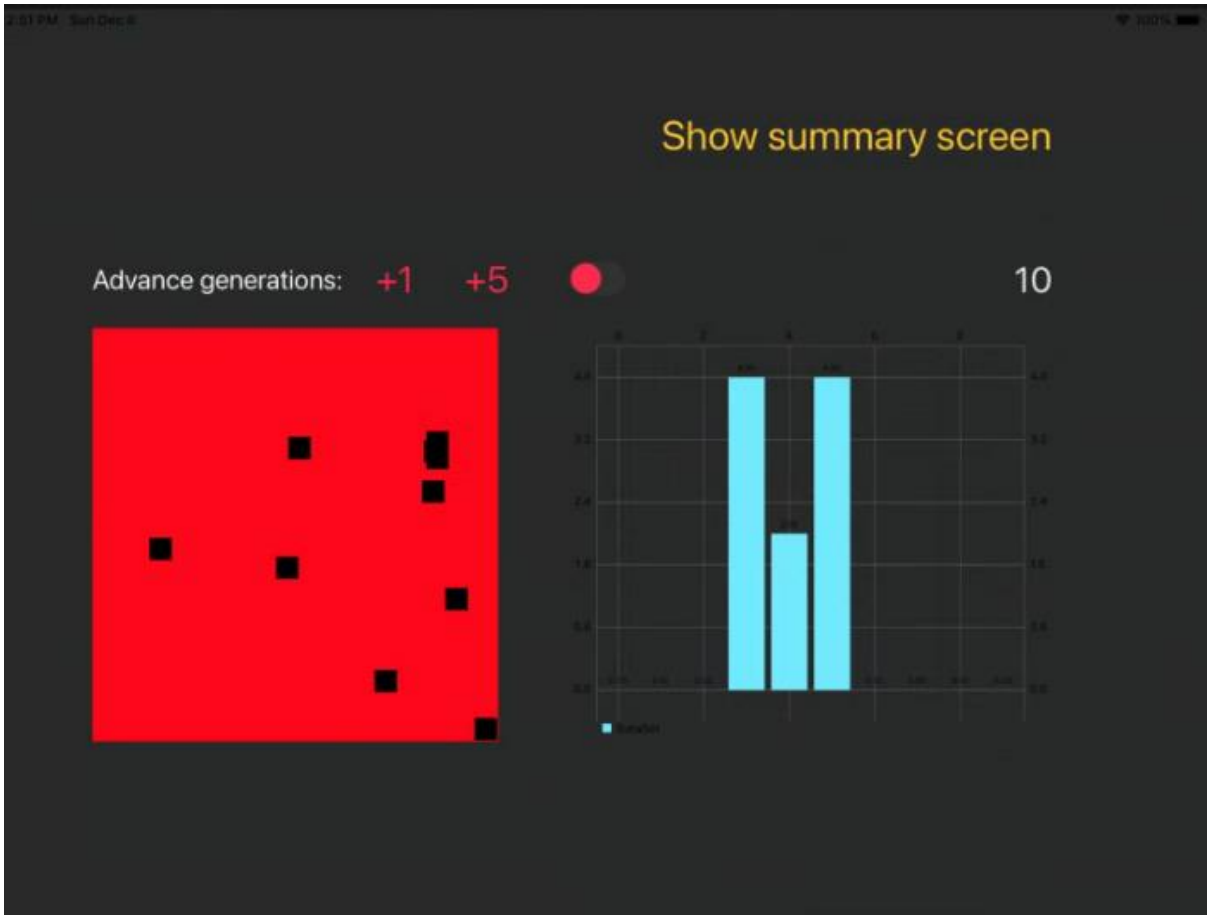
Now, the previous sprites are deleted before loading in the new ones.

Test

The test will be the same as before.


Show summary screen
Advance generations: +1 +5
5


Show summary screen
Advance generations: +1 +5
8



There are now the correct number of the sprites and they're positioned randomly in the view, so this test passed. However, it is quite hard to see some of the sprites because they're positioned on top of one another – I will fix this later if I have time.

S4 T4: Colour each sprite to represent its traitValue

To make the evolution of camouflage intuitive to the user, I will colour each sprite in the simulation to represent its camouflage as per success criterion 7.

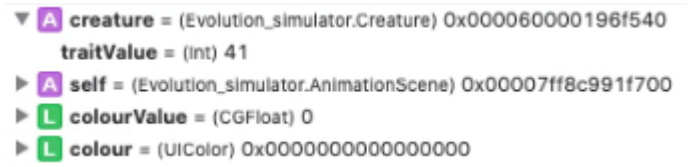
Code

```
func colourSprite(creature: Creature) -> UIColor {
    let colourValue = CGFloat(creature.traitValue / 100)
    let colour = UIColor(red: colourValue, green: colourValue, blue: colourValue, alpha: 1)
    return colour
}

for creature in environment!.creatures {
    let sprite = SKSpriteNode(color: colourSprite(creature: creature), size: CGSize(width: 20, height: 20))
    sprite.position = randomSpritePosition()
    addChild(sprite)
}
```

Test

When I run the app, I expect to see that the colour of each sprite is a different shade of grey. A creature with a traitValue of 1 should be almost black, whilst a creature with a traitValue of 100 should be pure white.

This test failed. Instead of the creatures being different shades of grey, they are all pure black. It is possible that every one of the 20 random creatures has a traitValue of 1, but this is astoundingly unlikely. To check, I added a breakpoint in my code to see what colourValue was being set to as the app ran.



Here you can see the creature has a traitValue of 41, but after 41 is divided by 100 and set to type CGFloat, it somehow becomes 0. I thought this was an issue with the conversion to data type CGFloat, but it turned out the issue was with how Swift handles integer division. When two integers divide to give a non-integer (e.g. 7/2 = 3.5) the non-integer part is simply discarded, so in Swift, 7/2 = 3. This means that dividing a creature's traitValue by 100 – which always gave a number between 0 and 1 – always gets rounded down to 0.

To fix this, I simply converted traitValue to a Double before dividing by 100 so that the decimal part of the result is kept.

Updated code

```
func colourSprite(creature: Creature) -> UIColor {
    let colourValue = CGFloat(Double(creature.traitValue) / 100)
    let colour = UIColor(red: colourValue, green: colourValue, blue: colourValue, alpha: 1)
    return colour
}
```

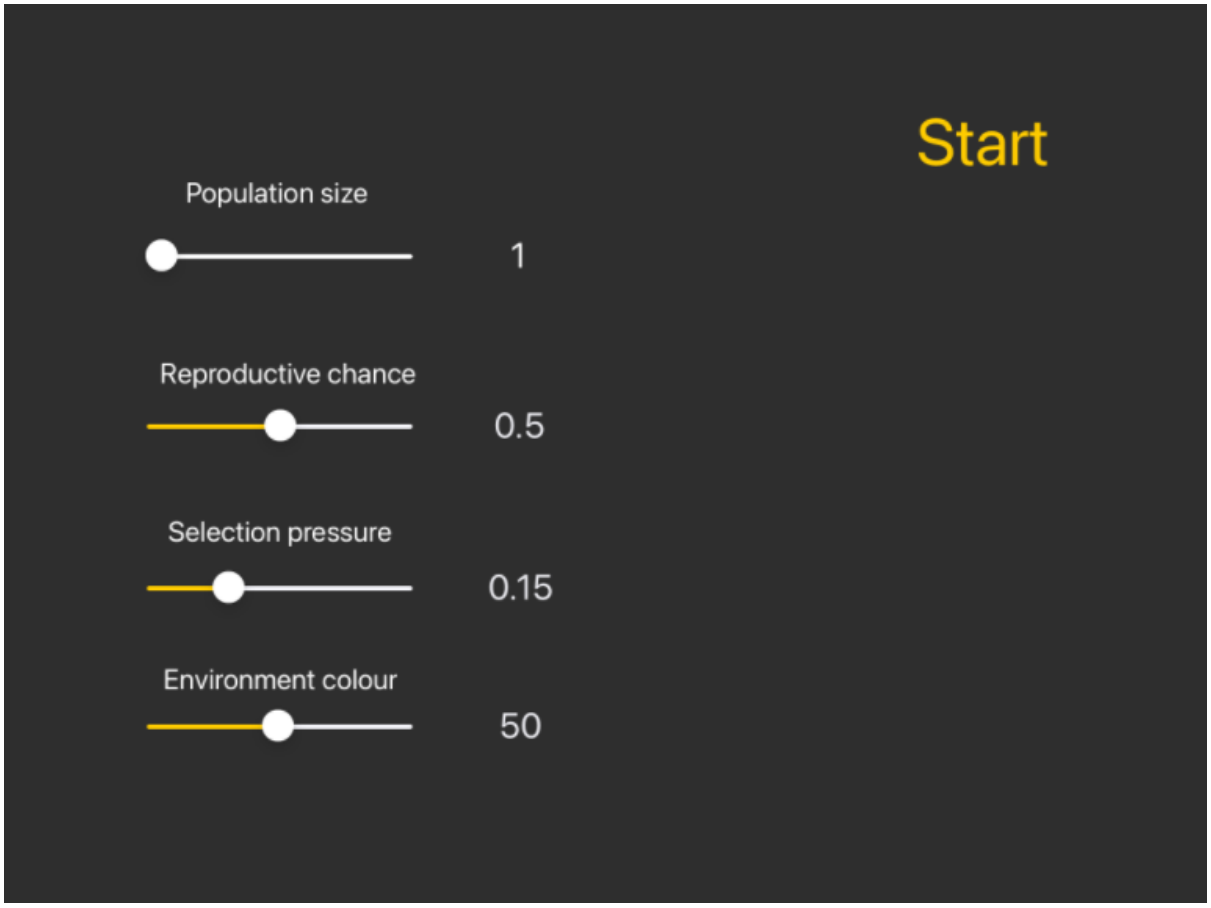
Test

Each sprite is a different shade of grey, so this test passed! This ticket is complete.

S4 T5: Change SKView background colour so creatures blend into background

I want the creatures to blend into the background as they evolve, so instead of the background being red it should be the shade of grey that the creatures are evolving towards. To implement this, I first need to add a slider the user can change so they can customise the background colour. This contributes to success criterion 6.

Code



I need to link this new slider to the parameter input view controller.

```
var environmentColour: Int = 50
@IBOutlet weak var environmentColourSlider: UISlider!
@IBOutlet weak var environmentColourLabel: UILabel!
```

Finally, I need to control what happens when the slider value is changed.

```
@IBAction func environmentColourChanged(_ sender: UISlider) {
    let value = Int(sender.value)
    self.environmentColour = value
    environmentColourLabel.text = String(value)
}
```

To make this slider do anything, I need to instantiate the environment of each simulation with this value of environmentColour.

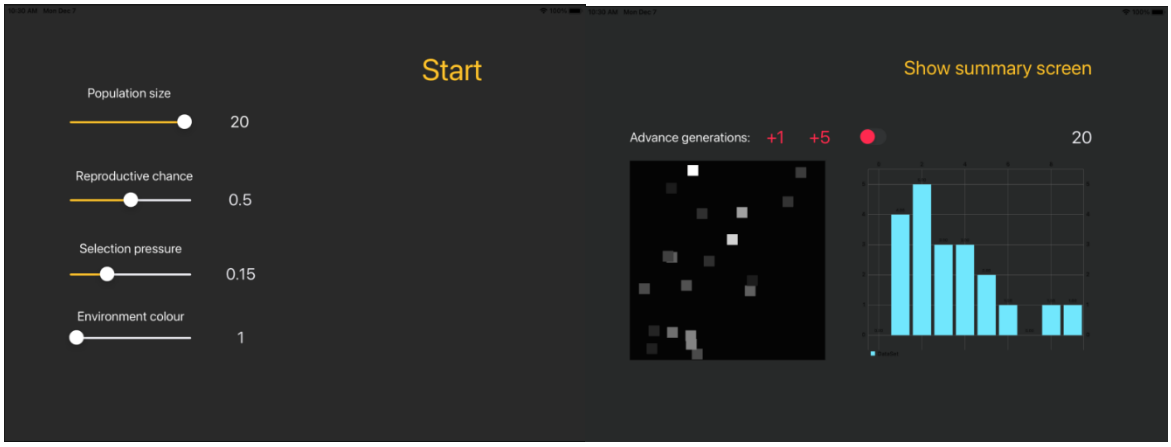
```
let environment = Environment(environmentColour: environmentColour, reproductiveChance: self.reproductiveChance,
    selectionPressure: self.selectionPressure, populationSize: self.populationSize)
```

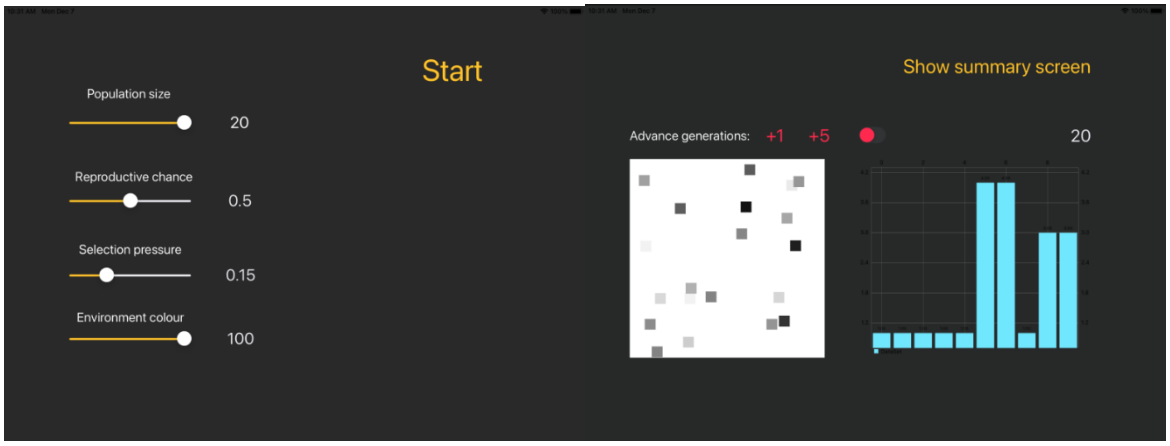
Now, when the SKView is loaded it should have its background colour set to the environment colour using Swift's RGB colour system.

```
func setBackgroundColour() {
    let colour = CGFloat(Double(environment!.environmentColour)/100)
    backgroundColor = UIColor(red: colour, green: colour, blue: colour, alpha: 1)
}
```

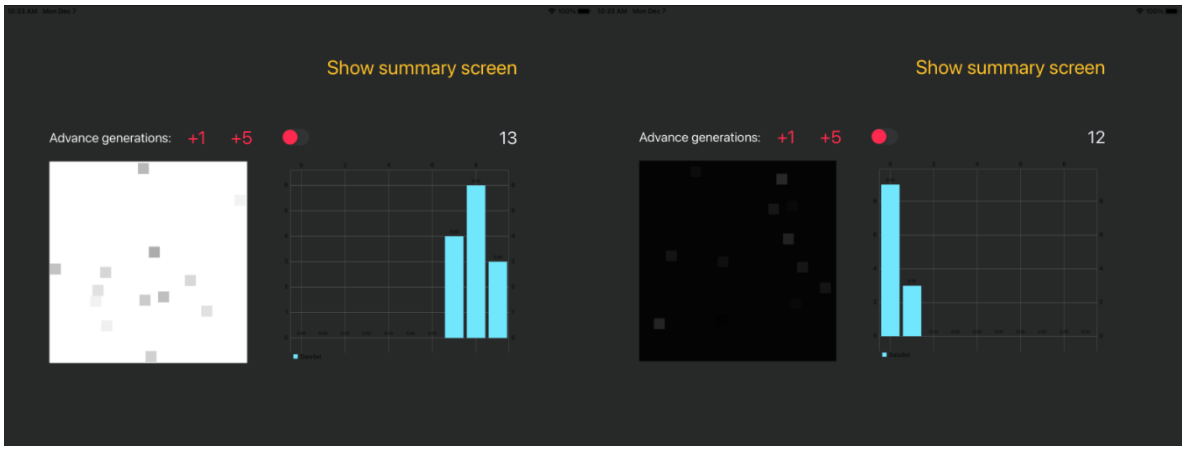
Test

To test the new slider, I simply need to run the app, change the slider to 1 or 100, then check the simulation view has a pure black (for 1) or pure white (for 100) background.





I also need to check that the creatures actually tend towards this colour as they evolve by advancing several generations and checking the creatures' colours converge to pure black or white.



It is obvious both from the live simulation and from the graph that the creatures are converging on the background colour, so this test passed.

S4 T6: Highlight best adapted creatures in simulation

After watching a number of simulations, it didn't look like the creatures were becoming very well camouflaged, even though I knew they must have been from the graph. I realised that this was due to the fact that the user *cannot see the best camouflaged creatures*. For instance, in the screenshot up and right from here (with the black background), there are 12 creatures in the population but only 7 are actually visible. 5 are too well camouflaged for the user to see them. I therefore want to highlight the best camouflaged creatures so the user knows the population is becoming better adapted to the environment – this will contribute to success criterion 7 and 10.

Code

To highlight the creatures, I will simply draw a box around them so it is obvious where they are, even if they perfectly blend into the background.

I created a function to draw boxes around creatures.

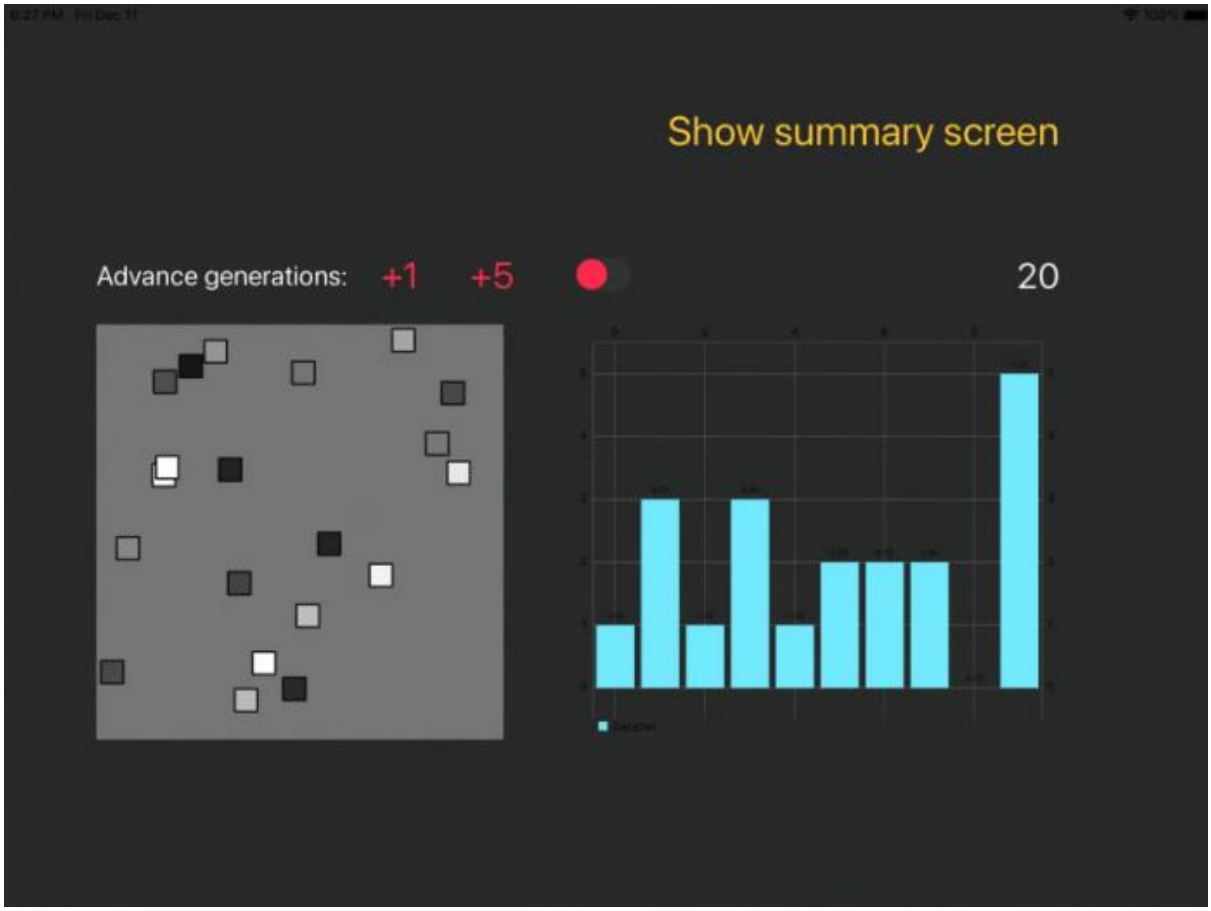
```
func drawBorder(position: CGPoint) {
    // Box around creature has same dimensions as creature
    let borderNode = SKShapeNode(rectOf: CGSize(width: 20, height: 20))
    // Box is centred on the creature
    borderNode.position = position
    borderNode.fillColor = .clear
    borderNode.lineWidth = 2
    borderNode.strokeColor = .black
    addChild(borderNode)
}
```

It is called whenever a new creature is drawn into the scene.

```
for creature in environment!.creatures {
    let sprite = SKSpriteNode(color: colourSprite(creature: creature), size: CGSize(width: 20, height: 20))
    sprite.position = randomSpritePosition()
    addChild(sprite)
    drawBorder(position: sprite.position)
}
```

Test

Testing this simply involves running the app, and checking to see whether a black box is drawn around every creature.



This test passed. Next, I need to change the border colour of the best adapted creatures to highlight them.

Code

```
func drawBorder(position: CGPoint, creature: Creature) {
    // Box around creature has same dimensions as creature
    let borderNode = SKShapeNode(rectOf: CGSize(width: 20, height: 20))
    // Box is centred on the creature
    borderNode.position = position
    borderNode.fillColor = .clear
    borderNode.lineWidth = 2
    borderNode.strokeColor = .black

    // checks if creature colour is within 5 of the environment background
    if -5...5 ~= creature.traitValue - environment!.environmentColour {
        borderNode.strokeColor = .red
    }

    addChild(borderNode)
}
```

Test

The best adapted creatures should now have a red border instead of a black one.

To start with, some of the creatures have red borders.

After 30 generations of evolution, many more have red borders:

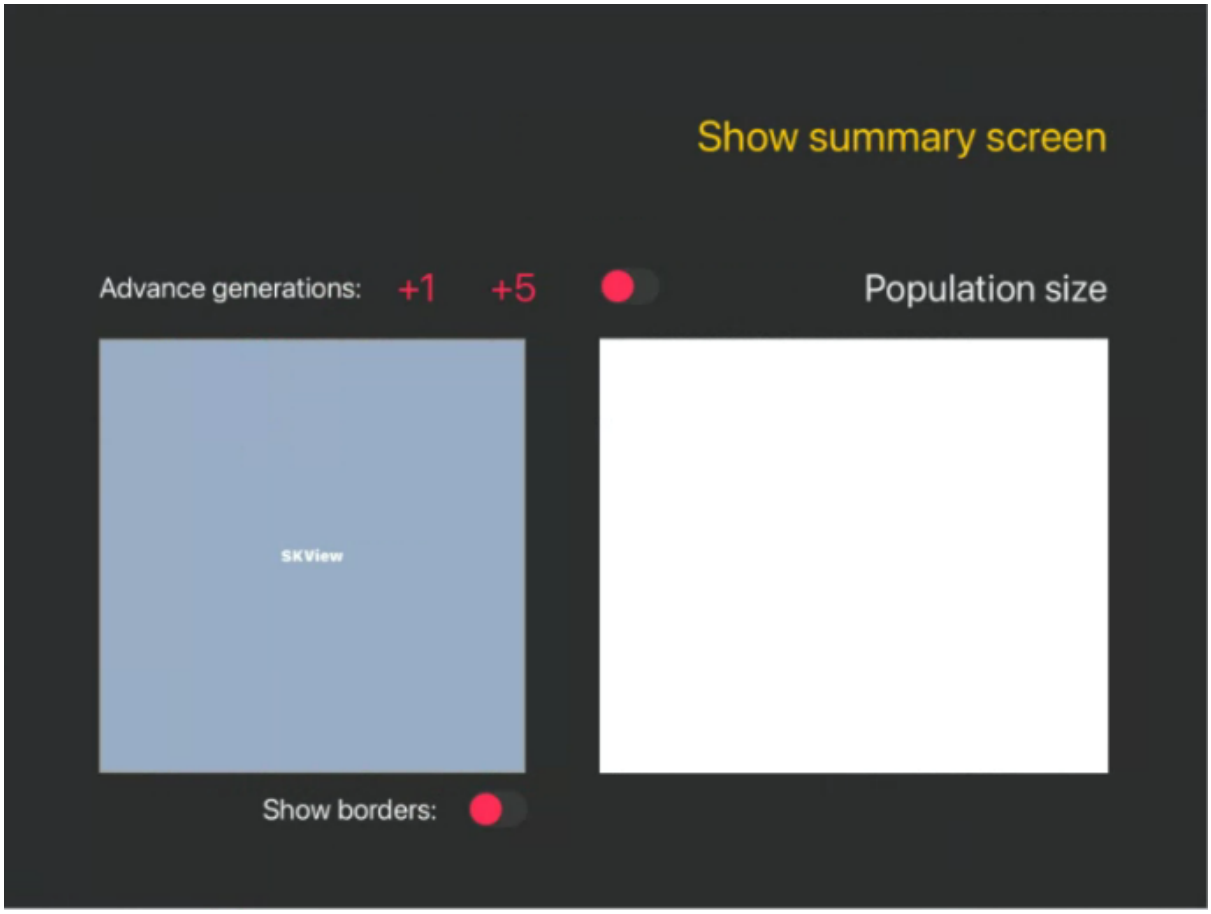
This tells me the creatures are evolving as expected, and that the highlighting works. This test passed!

S4 T7: Allow the user to remove the creature borders

When using the app myself, I wanted to sometimes remove the borders to see if the best creatures are at all visible without them. This would highlight to the user how well adapted the creatures are – without the borders, they are impossible to see.

Code

To make this happen, the first thing I need is a switch in the simulation view.



By default, the borders will not be showing. Next, I need to link this switch to the simulation view.

```
// Shows or hides the sprite borders
@IBOutlet weak var showBorders: UISwitch!
@IBAction func showOrHideBorders(_ sender: Any) {
}
```

I now need to add functions to AnimationScene that add or remove the border sprites. To store the position of each sprite, I added a new variable that can store each border sprite.

```
var borders: [SKShapeNode] = []
```

Each time I add a border to the scene, I will add it to this array.

```
borders.append(borderNode)
addChild(borderNode)
```

Now, I can add the functions that will show and hide the borders as follows:

```
func hideBorders() {
    for border in borders {
        border.removeFromParent()
    }
}

func showBorders() {
    for border in borders {
        addChild(border)
    }
}
```

The last thing to do is call hideBorders() when the switch is turned off, and showBorders() when the switch is turned on.

```
// Shows or hides the sprite borders
@IBOutlet weak var showBorders: UISwitch!
@IBAction func showOrHideBorders(_ sender: Any) {
    if showBorders.isOn == true {
        scene!.showBorders()
    } else {
        scene!.hideBorders()
    }
}
```

Test

When the simulation first loads, I expect the borders not to be there. When I turn the switch on, I expect the borders to appear, and when I turn it off I expect the borders to disappear.

Straight away, this test failed as the borders were there to start with. Worse, when I changed the switch to on, the app crashed and this error was shown.

```
libsystem_kernel.dylib`__pthread_kill:
    0x7fff5dca6330 <+0>:  movl   $0x2000148, %eax          ; imm = 0x2000148
    0x7fff5dca6335 <+5>:  movq   %rcx, %r10
    0x7fff5dca6338 <+8>:  syscall
->  0x7fff5dca633a <+10>: jae    0x7fff5dca6344            ; <+20>  =  Thread 1: "Attemped to add a SKNode which already has a parent: <SKShapeNode> na...
    0x7fff5dca633c <+12>: movq   %rax, %rdi
    0x7fff5dca633f <+15>: jmp    0x7fff5dca0629            ; cerror_nocancel
    0x7fff5dca6344 <+20>: retq
    0x7fff5dca6345 <+21>: nop
    0x7fff5dca6346 <+22>: nop
    0x7fff5dca6347 <+23>: nop
```

This error tells me I somehow need to remove each border's 'parent' after removing it so that a new parent can be set to load it into the scene.

Updated code

Instead of entirely removing each border from the scene then adding it again, I used a function built into SpriteKit for hiding and showing sprites: sprite.isHidden = bool.

```
func hideBorders() {
    for border in borders {
        border.isHidden = true
    }
}

func showBorders() {
    for border in borders {
        border.isHidden = false
    }
}
```

Test

I expect to see the same as I expected in my first test: initially, the borders are hidden, but they appear/disappear when I change the switch.

Again, this test failed because the borders are showing to begin with. However, changing the state of switch does work.

Updated code

```
override func viewDidAppear(_ animated: Bool) {
    scene = AnimationScene(size: skView.bounds.size)
    scene!.environment = environment
    scene!.setBackgroundColour()
    scene!.placeCreatures()
    scene!.hideBorders()
    self.skView.presentScene(scene)
}
```

To fix this, I simply hid the borders when the view first appears.

Test

This time, I expect the borders to be hidden when I first load the view.

This test seemed to pass, but when advancing generations with the switch off, the borders appeared each new generation.

This is not what I expected, so this test failed.

Updated code

```
func advanceOneGeneration() {
    if environment?.advanceOneGeneration() == "extinct" {
        extinctionAlert()
    } else {
        updateViews()
        scene!.placeCreatures()
        if showBorders.isOn == false {
            scene!.hideBorders()
        }
    }
}
```

I fixed this by checking if the switch is off each generation, and calling hideBorders() if it is.

Test

This test finally passed! This ticket is complete.

End of sprint review

This was my final sprint. I am very happy with the progress I made, but I am frustrated I didn't have time to add the description of evolution and an explanation of how the app works – it would be very nice to add these in further development to make the app a better learning tool.

Stakeholder feedback

Questions (sent to Ms. Hicks by email) in black, and answers in red.

1. Which are the most crucial features the app is missing?
   Does it have an explanation section for how the app works?

2. Which features do you like the most?
   Seems to have an intuitive, user friendly face even without an explanation of how it works (it would be great to play around with it myself to truly understand how it works). I like the idea of having graphs running along-side the visual representation.

3. Is this a useful simplification of evolution?
   Difficult to say without a little more explanation of how the numbers and simulation works.

4. Does the app look easy to use? What changes would you make to its design?
   Easy to use, yes. It is very difficult to see any numbers / lettering on the graphs due to the black on black colour scheme – could you use white letters/numbers? Perhaps some actual case studies of organisms and their evolution?

5. How does the app compare to your ideal version of it?
   It would have an explanation, perhaps a tutorial showing how it works – this would be great for students. It could perhaps have some case studies of evolution of camouflage in nature (there are some really interesting examples out there, beyond peppered moths!).

G.L.:

1. Which are the most crucial features the app is missing?
   Instructions explaining how to use the app. It would also be nice to have the creatures move a bit instead of only staying still.

2. Which features do you like the most?
   The graph near the simulation is very cool and I like how the creatures slowly blend into the background.

3. Is this a useful simplification of evolution?
   Overall yes. However it is difficult to understand exactly what is happening in the app and if this is how it would happen in real life. It would be nice to have some text explaining what is going on.

4. Does the app look easy to use? What changes would you make to its design?
   Yes, it looks easy to use. It would be nice if there was some text describing where this has happened in nature, but the design looks nice.

5. How does the app compare to your ideal version of it?
   If I used this to actually learn about evolution, it would be very nice to have a description of how evolution works and how this app works. I'd also really like to see some different animals being evolved – how did big/small animals evolve, and fast/slow.

Conclusion

Unfortunately, due to COVID-19, allowing the biology teacher stakeholder to use the app herself was impossible.

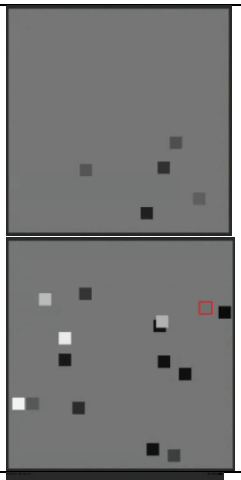
The stakeholder feedback suggests I have designed the app well and it is easy to use. A notable exception to that is the dark font used for the graph axes, which are difficult to see. This would be an easy but very useful change to make with more development time.

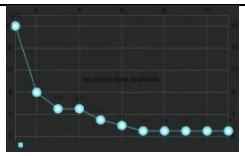
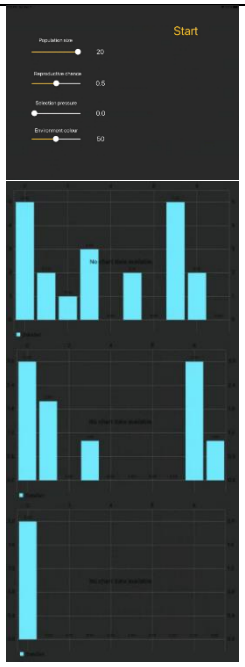
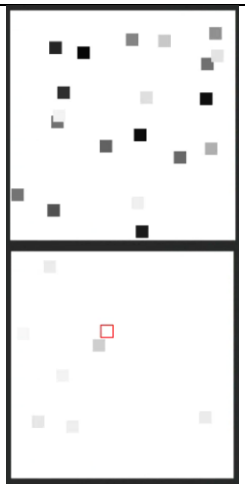
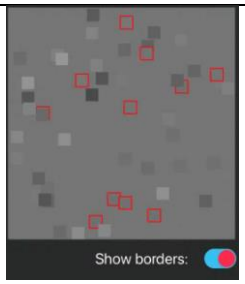
It is stressed that an explanation of how the app works and of how evolution works would improve the app, so I would also like to add those features along with some case studies of evolution that students might themselves come across. Of course, I would also like to add some more traits for the user to select.

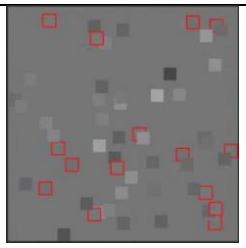
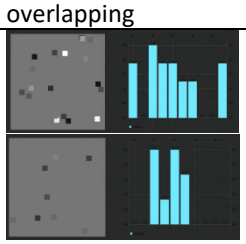
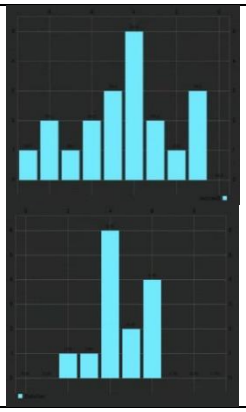
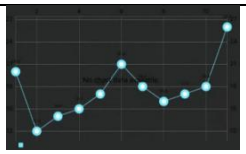
Evaluation

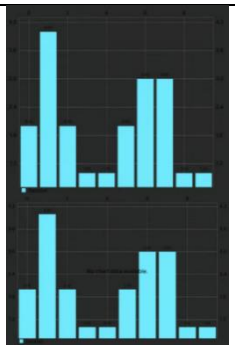
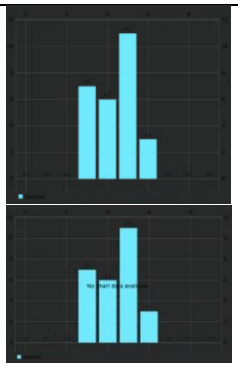
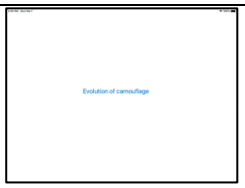
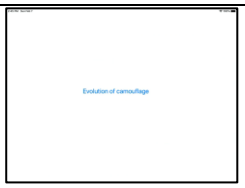
Testing to inform evaluation

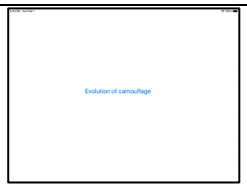
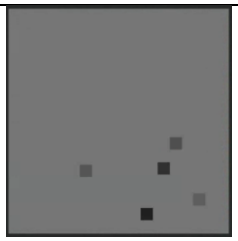
Function tests

Test no.	Description	Reason	Test data	Expected result	Actual result	Passed?
1	Test user can change initial population size	A key feature of the simulator is that the user can change initial conditions to see the impact	Initialise environment with 5 creatures, then with 15 creatures	First environment will have 5 creatures, second will have 15		Yes
2	Test advancing one generation	Advancing generations is central to the function of the app	Create an environment, then press the “+1” button	The population should be updated in distribution and size	 Population changed from 16 creatures to 14 and distribution narrowed	Yes
3	Test advancing five generations	Users should be able to advance generations faster than 1 at a time	Create an environment, then press the “+5” button	Population distribution should change more than advancing one generation	 Population distribution narrowed from random data to steep bell curve	Yes
4	Test auto advance through generations	User should be able to advance automatically through app	Create an environment, then turn auto advance generations on	User should advance one generation per second without pressing anything	 Generations advance automatically once per second	Yes
5	Test user can change reproductive chance	The user should be able to change the creatures’ traits along with the	Initialise environment with 20 creatures and reproductive chance of 0	All 20 creatures should die with 20 generations		Yes

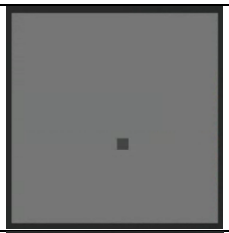
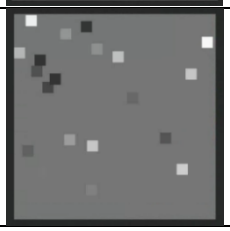
		number of creatures				
6	Test user can change selection pressure	User should be able to change environment conditions as well as the creatures	Initialise environment with 20 creatures and selection pressure of 0. Advance 5 generations. Record distribution of 1st, 3rd, and last generation	Initial random distribution of creatures should **not** converge on environment colour (middle value)		Yes
7	Test user can change environment colour	User should be able to select the colour towards which the creatures converge	Initialise environment with 20 creatures and environment colour of 100, then advance 5 generations. Record first and last generation	Environment should be pure white and creatures should tend towards light colours		Yes
8	Test showing borders	This test ensures the borders around well camouflaged creatures are displayed so that they are visible	Select “Show borders” option	Well camouflaged creatures should be outlined in red		Yes

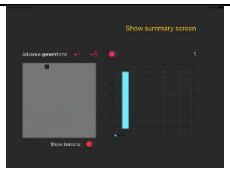
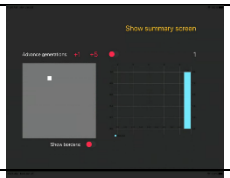
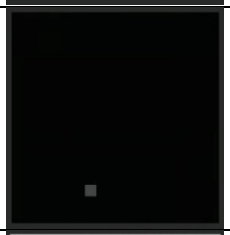
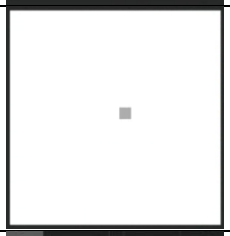
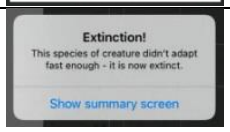
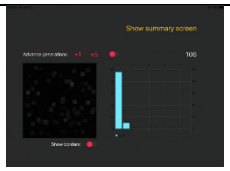
9	Test that creatures do not overlap	This test ensures creatures do not overlap so they are entirely visible to the user	Advance one generation	None of the creatures should overlap	 Creatures are overlapping	No
10	Test that bar graph represents distribution of creatures in simulation screen	Graph should summarise distribution of traits within the population	Initialise environment then advance one generation, recording the graph output	The graph should initially have a roughly even (random) spread, then narrow towards the median value after one generation		Yes
11	Test bar graph updates in real time as simulation runs	Graph should update in real time as per success criterion 11	Initialise environment with 20 creatures, then advance one generation	The bar graph should distinctly change shape to reflect the changing population		Yes
12	Test that user can increase number of bars in bar graph in simulation screen	The user should be able to see finer detail than is shown by default in the bar graph to see smaller changes in population	Increase the number of bars shown in bar graph	The bar graph should be split into more bars	There is no option to increase number of bars	No
13	Test scatter graph displays history of population size	User should be able to see at a glance how the population has grown and shrunk through the simulation	Advance 10 generations, then look at scatter graph in simulation summary screen	Scatter graph should have 11 entries (including initial population), each entry showing number of creatures		Yes

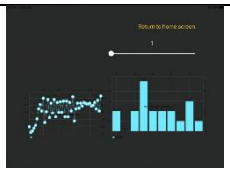
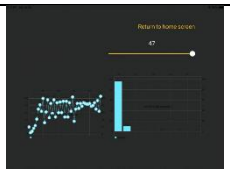
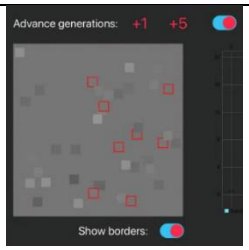
14	Test user can zoom into population scatter graph in simulation summary screen	The user should be able to zoom into the population size history scatter graph to see finer detail	Zoom into a random section of the scatter graph	The entire scatter graph view should be taken up with a small section of the entire graph	No option to zoom into scatter graph	No
15	Test bar graph in simulation summary screen displays distribution of creatures	The bar graph in simulation summary screen should show distribution of creatures in each generation of simulation	Initialise environment, then go to summary screen without advancing any generations	Bar graph in simulation and summary screen should be identical		Yes
16	Test user can user slider to look at distribution of each generation in summary screen	To look at the distribution of different generations, the user should be able to select each generation they progressed through in the simulation	Initialise environment, advance 10 generations, go to summary screen, and select the latest generation to display in bar graph	Population distribution in summary screen should be the same as the final distribution in simulation screen		Yes
17	Test user can see an explanation of evolution within the main menu	Because the simulator is designed to be a learning tool, the user should be able to see a simple explanation of evolution	Select 'Explanation of evolution' button in main menu	A pop-up window will contain a simple explanation of evolution	 There is no explanation of evolution button	No
18	Test user can select different traits in main menu	I wanted the user to be able to select several traits in order to get a better handle on evolution	Select two different traits to see evolve from the main menu	There will be multiple traits inside the main menu. Selecting different traits will bring the user to different parameter input screens	 There is only one trait the user can select	No

19	Test user can select instructions in the main menu	Because this program is designed to be a learning tool, it should be simple to use. It should therefore have instructions to help users use the app	Select the 'Instructions' button in the main menu	A pop-up window will contain instructions describing what the app does and how to use it	 There is no 'Instructions' button	No
20	Test that each trait is given in context by a pop-up window shown after selection	To ensure users understand the analogy with real life, an example of where the trait has evolved should be given for each trait	Select the 'Evolution of camouflage' button	A pop-up window describing where camouflage has evolved like this in nature will appear	 The user is taken straight to the parameter input screen	No
21	Test creatures are shown visually using colour	A key success criterion was to show the evolution of traits using colour and animation	Start a simulation	Creatures should be displayed as different shades of a colour	 Instead of using colour, I used a grayscale	Yes

Robustness tests

Test no.	Description	Reason	Test data	Expected result	Actual result	Passed?
1	Test minimum initial population size	Selecting boundary data for population does not crash app	Initialise environment with population size 1	Simulation will be populated with 1 creature		Yes
2	Test maximum initial population size	Selecting boundary data for population does not crash app	Initialise environment with population size 20	Simulation will be populated with 20 creatures		Yes

3	Test minimum reproductive chance	Boundary data for reproductive chance should not crash app	Initialise environment with reproductive chance of 0 then advance a generation	Simulation screen will be shown		Yes
4	Test maximum reproductive chance	Boundary data for reproductive chance should not crash app	Initialise environment with reproductive chance of 1	Simulation screen will be shown		Yes
5	Test minimum selection pressure	Boundary selection pressure values should not crash app	Initialise environment with selection pressure of 0	Simulation screen will be shown		Yes
6	Test maximum selection pressure	Boundary selection pressure should not crash app	Initialise environment with selection pressure of 0.5	Simulation screen will be shown		Yes
7	Test minimum environment colour	Boundary environment colour should not crash app	Initialise environment with environment colour of 1	Simulation will have pure black background		Yes
8	Test maximum environment colour	Boundary environment colour should not crash app	Initialise environment with environment colour of 20	Simulation will have pure white background		Yes
9	Test population dropping to 0	All creatures in a population ‘dying’ should not crash the app	Initialise environment with reproductive chance of 0 and population size of 1	After several generations the creature should die, and a pop-up should appear describing extinction		Yes
10	Test large population sizes	The app should limit how large a population can grow	Initialise environment with reproductive chance of 1	The creatures should grow to a maximum of around 100 creatures	 The largest the population grew to was 106 creatures	Yes

11	Test minimum generation selected in summary screen	Boundary generation selection should not crash app	Select first generation to view in summary screen	The bar chart should display distribution of first generation		Yes
12	Test maximum generation selected in summary screen	Boundary generation selection should not crash app	Select last generation to view in summary screen	The bar chart should display distribution of last generation		Yes
13	Test borders around well camouflaged creatures are shown while advancing generations	Advancing through generations should not remove the borders around well camouflaged creatures	Select "Show borders" option, then automatically advance through generations	Borders around best camouflaged creatures should always be shown	 Borders were always shown	Yes

Usability tests, and how to address unmet usability features

One of the key ways in which I wanted to make my app usable by a wide range of people is using colours that are easy to distinguish no matter how the user perceives colour – even people with total colour blindness. I therefore settled on using grayscale instead of colours, as the creatures are distinguished only by brightness. This usability test is therefore fully met.

Another usability feature is that the instructions for the app must be a single click from the main screen. This usability feature is unmet because there are no instructions. In further development, I would add a clearly labelled button with some text explaining how to use the app, with the text prompted by my stakeholders.

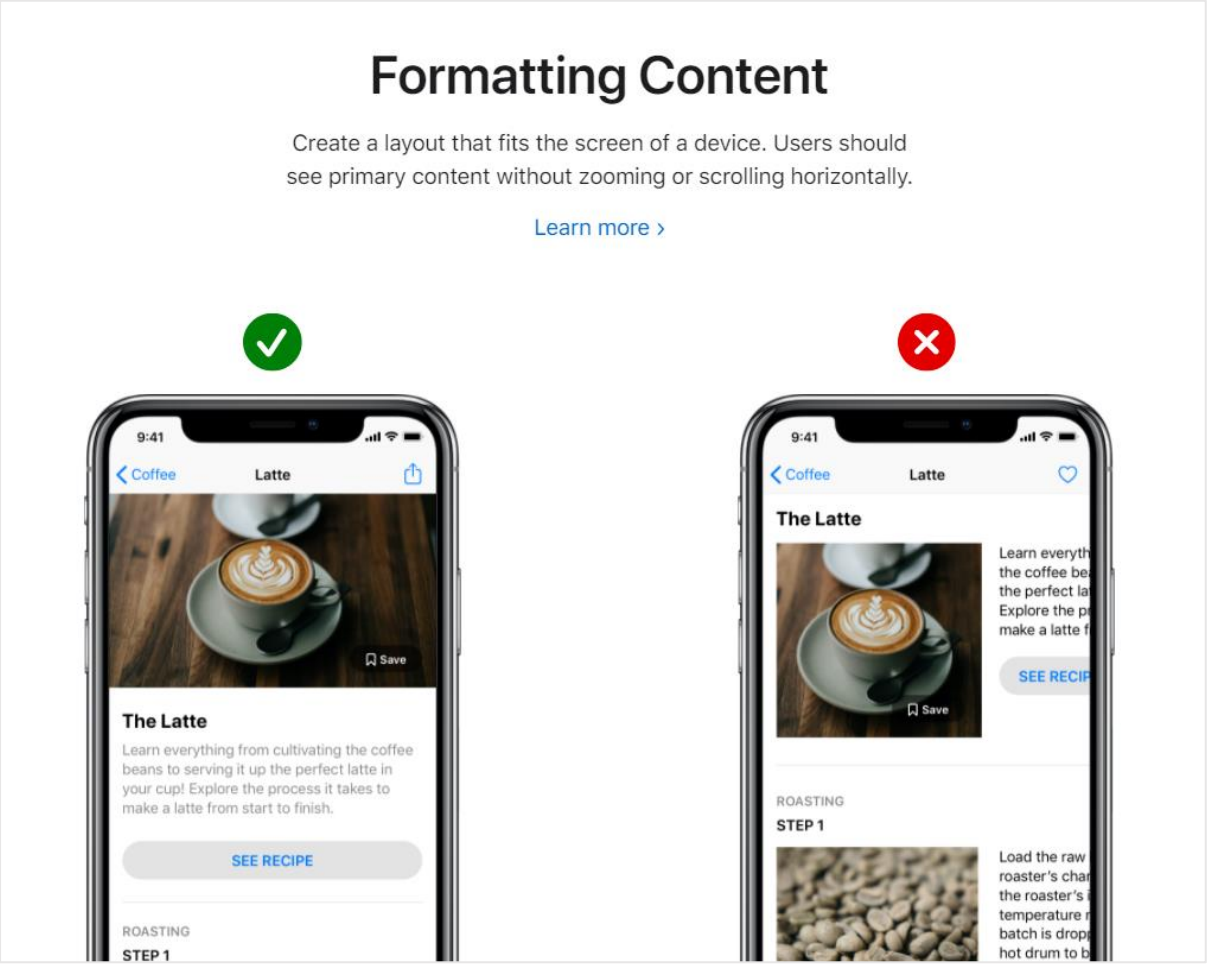
Initially, I wanted an information button in the bottom right of each trait the user could select, which would provide an example of where that trait has evolved before. This would help the user intuitively understand what is happening as the creature evolves in my app. This feature is unmet as there is no information button. In further development, I would look for the simplest examples of where each trait has evolved, and explain in a popup window how that relates to the selected trait.

A key feature that would help people navigate the menus was to have buttons for continuing through the process (e.g. "Start simulation, "Stop", "Close") on the right hand side of the screen and coloured red to signify an irreversible action. The buttons for continuing are in the top right of the screen, but are coloured yellow instead of red to stand out against a dark background. This means that although the feature is technically partially met, my current solution is better than my original solution. I will therefore mark this feature as fully met.

Another way in which I have tried to ensure my app is usable is by sticking to Apple's usability guidelines – what they have called the "UI Design Dos and Don'ts" (find them on the Apple website at https://developer.apple.com/design/tips/). I will therefore go through each of the Dos and Don'ts, commenting on whether my implementation of Apple's usability features have been a success, partial

success, or failure as effective usability features. I will also comment on how partially successful or unsuccessful features could be addressed in further development.

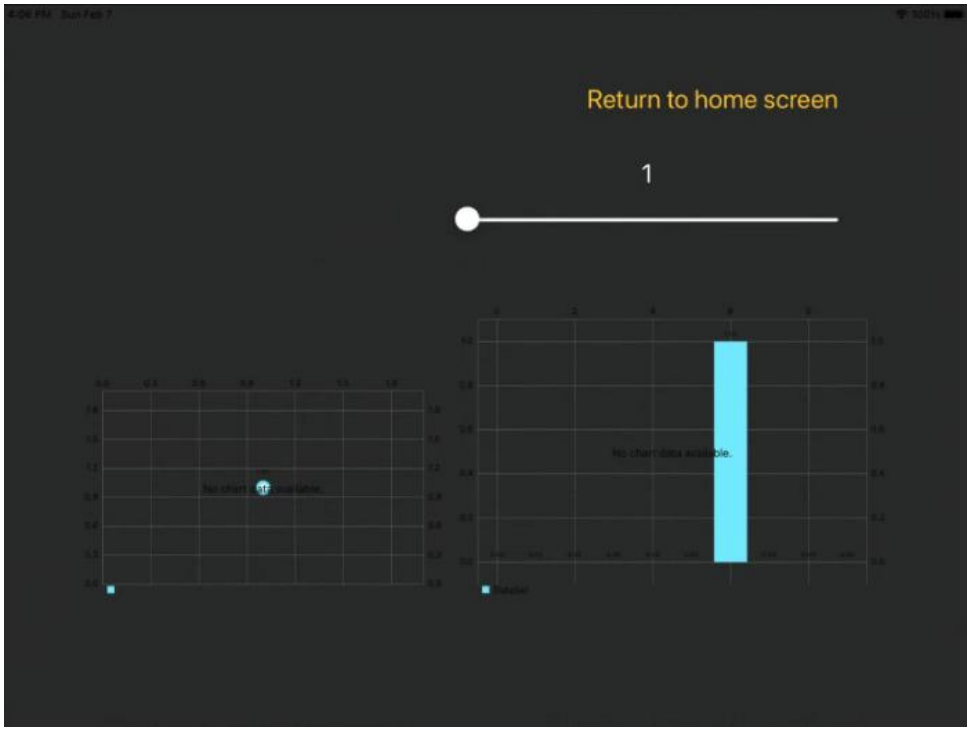
1.



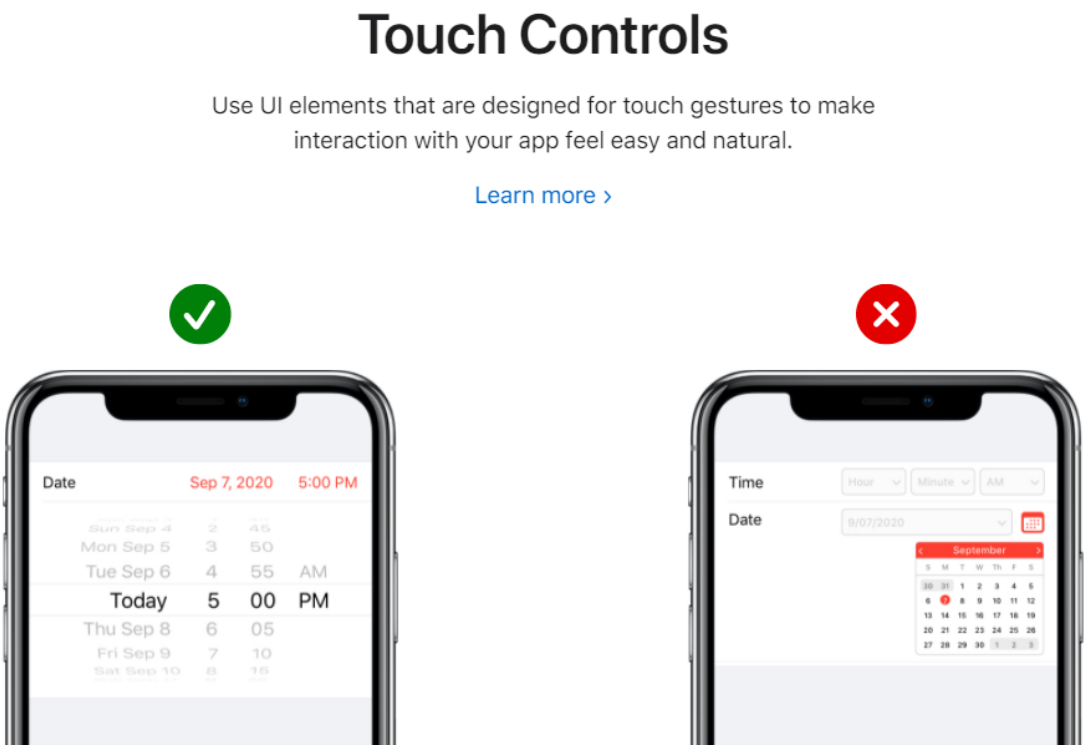
The first guideline tells us that primary content on a page should be visible without zooming or scrolling horizontally. I think my implementation of this guideline is a success because my app follows this guideline in every instance. I have ensured each page displays all its information without needing to zoom or scroll. See three examples below:


Start
Population size
1
Reproductive chance
0.5
Selection pressure
0.15
Environment colour
50


Show summary screen
Advance generations:
+1
+5
1
Show borders:

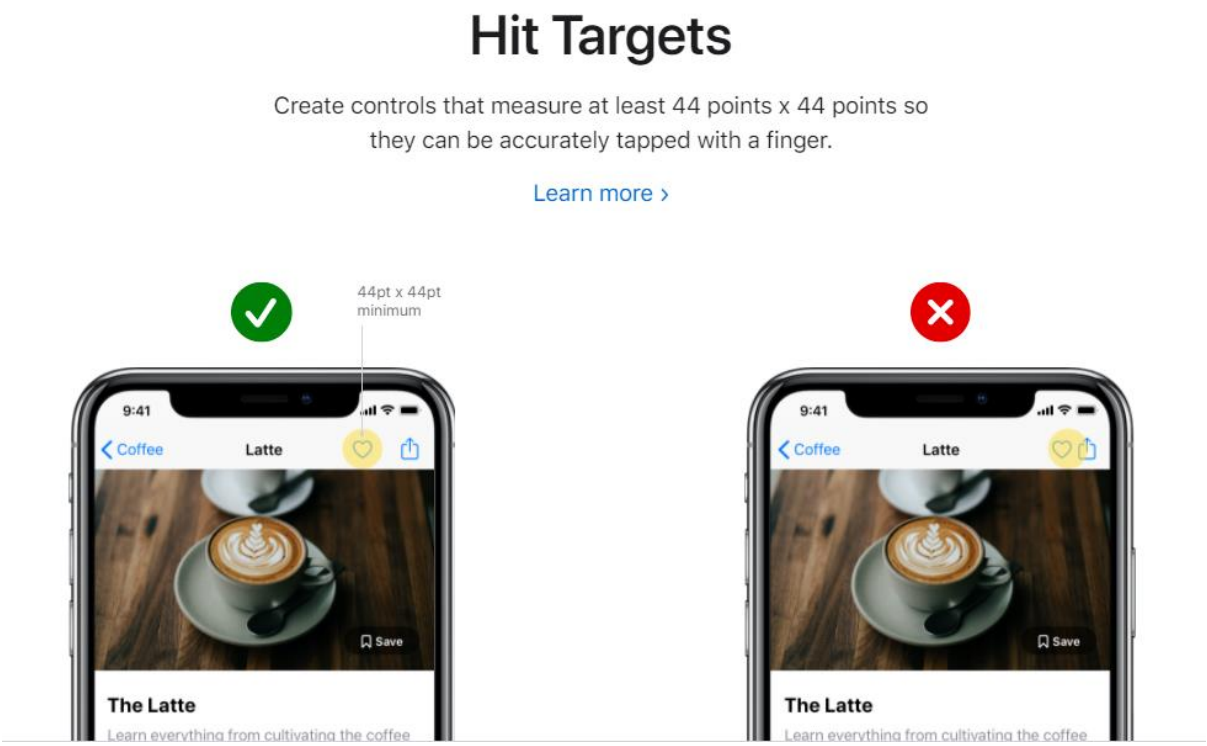


2.



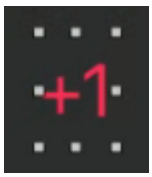
This guideline suggests all UI elements should be designed for touch gestures so that navigating and using the features of the app is intuitive. I have kept to this guideline by exclusively using native UI elements available from within XCode. This ensures the elements match with each other and are easy to use because they are designed by Apple themselves.

3.



This usability feature is only partially met as there are two buttons within my app that are narrower than 44 pixels:

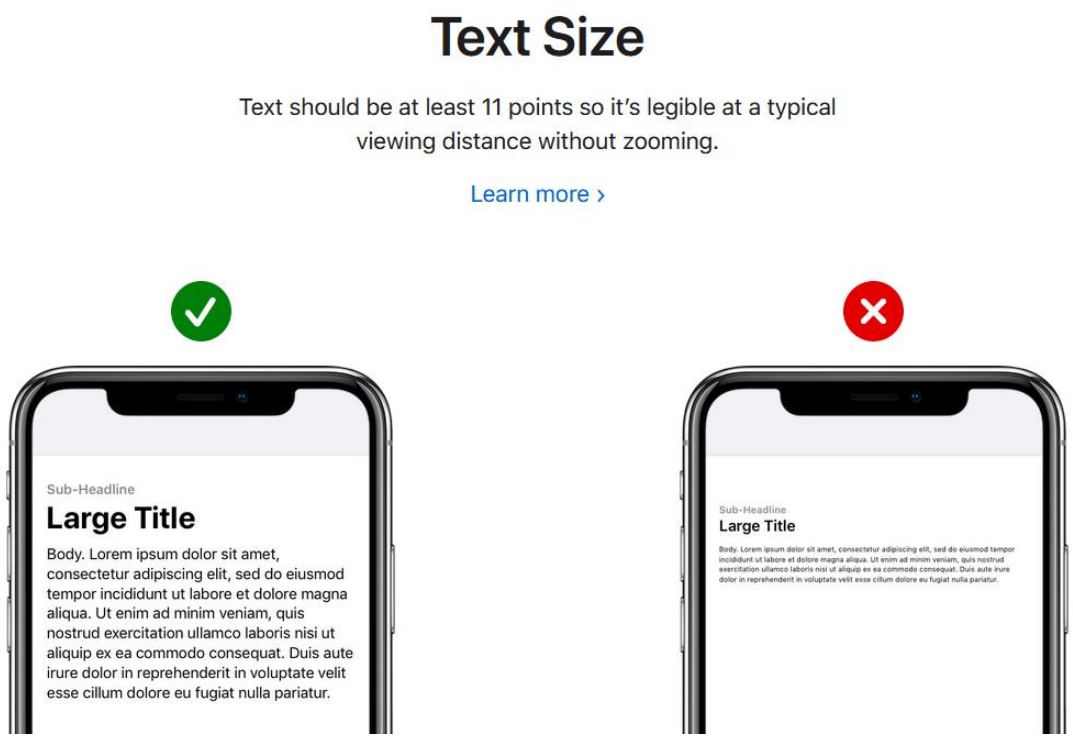
This button is 35 pixels wide:



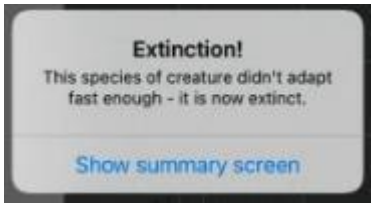
This button is 41 pixels wide:

This partially met usability feature could be addressed in further development by widening the buttons by 9 and 4 pixels respectively. This would place them at the minimum acceptable width of controls according to the Apple guidelines while not making them disruptive to the layout of the rest of the app.

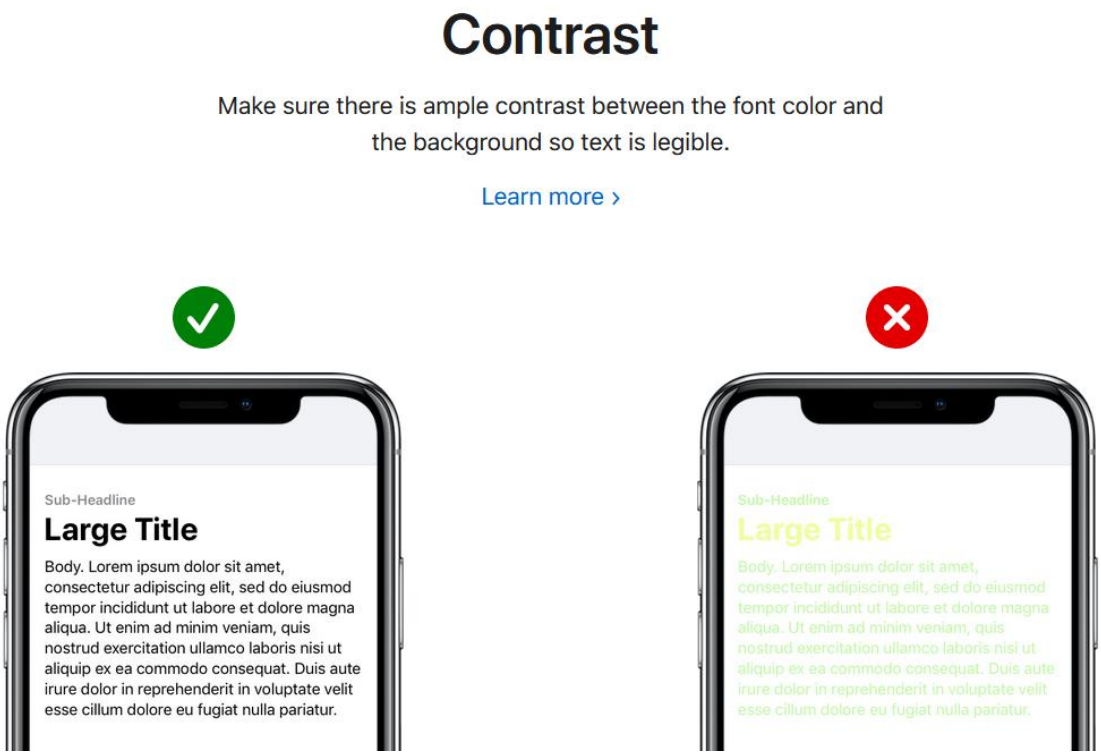
4.



This guideline is met throughout my app, as the smallest text in my app is the default alert text size, which is 14 points.



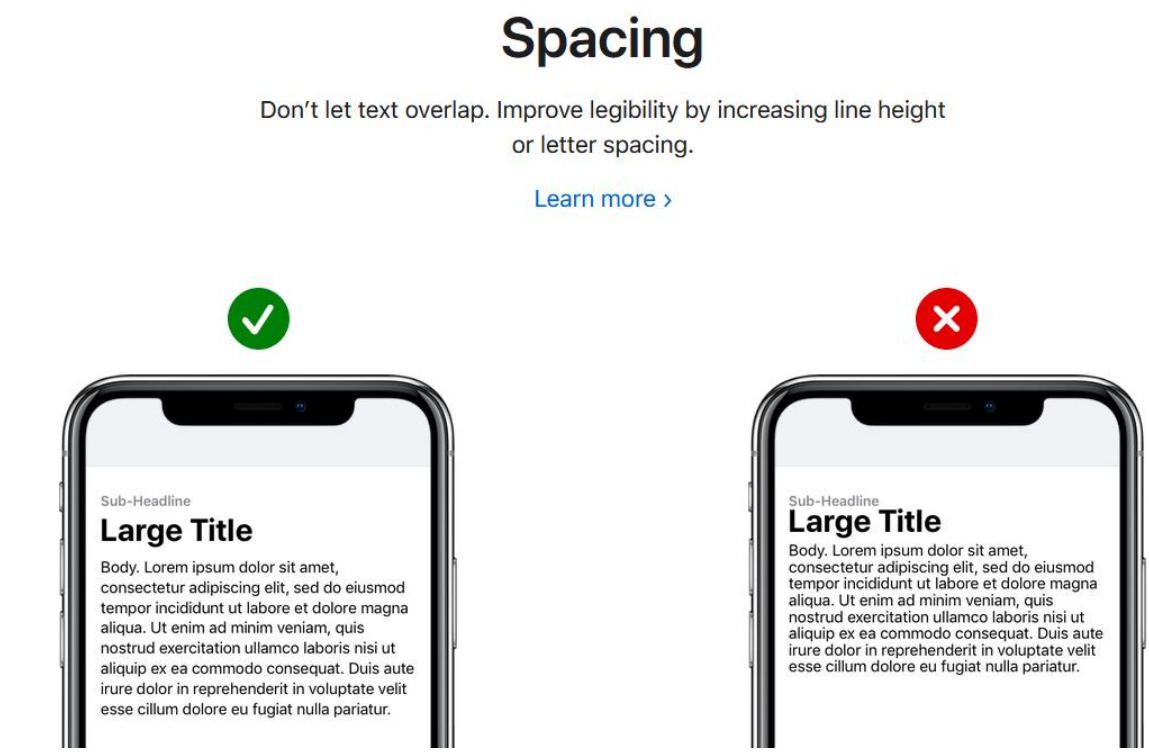
5.



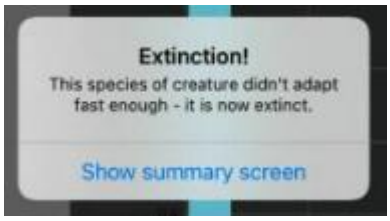
All the text in my app is white on black, red on black, yellow on black, or white on blue. The lowest contrast in my app is the white on blue, but there is still strong contrast:

This usability feature is therefore a success as an effective feature.

6.

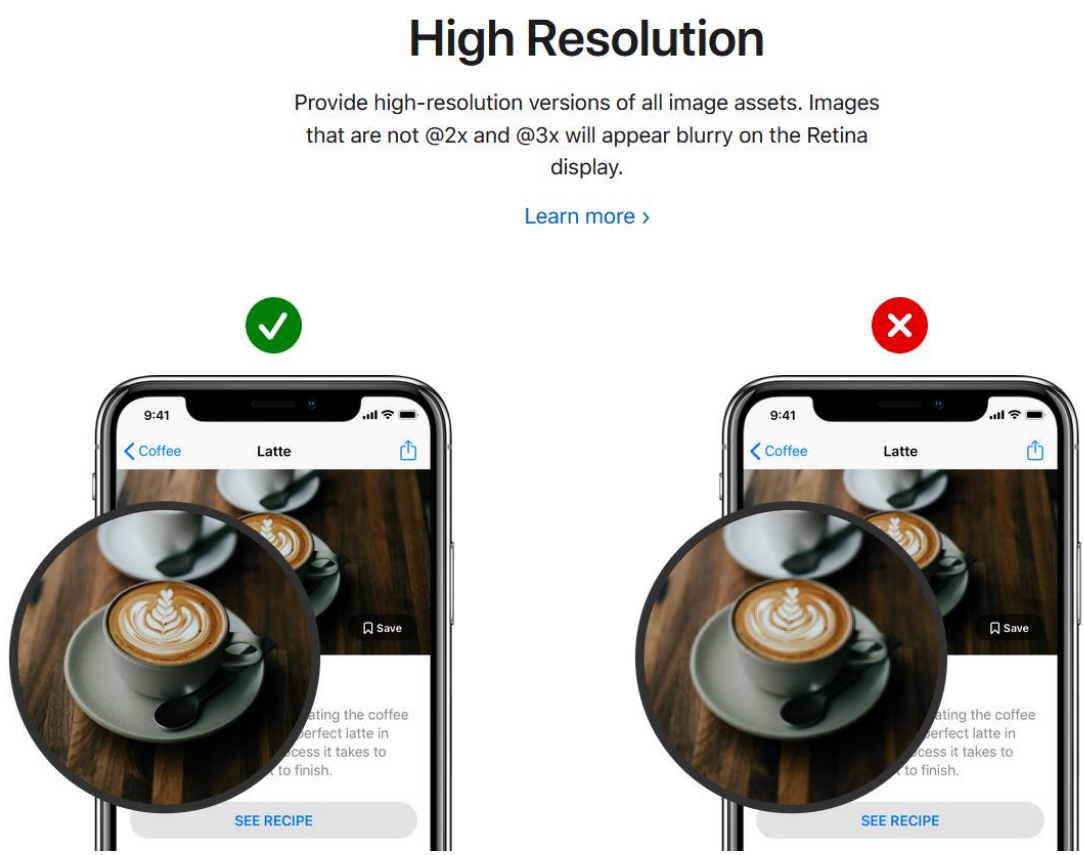


The closest spacing within the app is in the popup window after all the creatures of a population die, shown here:



None of the text overlaps and the space between lines and letters is sufficient for the text to be easily legible. This usability feature is a success.

7.



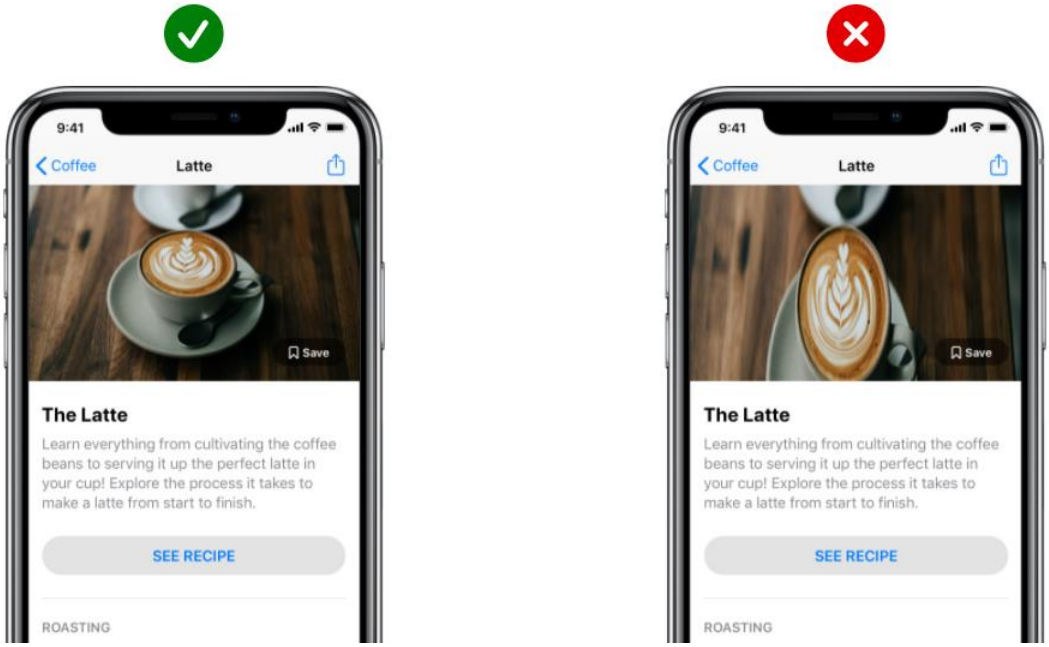
This usability feature does not apply as I have not imported any images into my app.

8.

Distortion

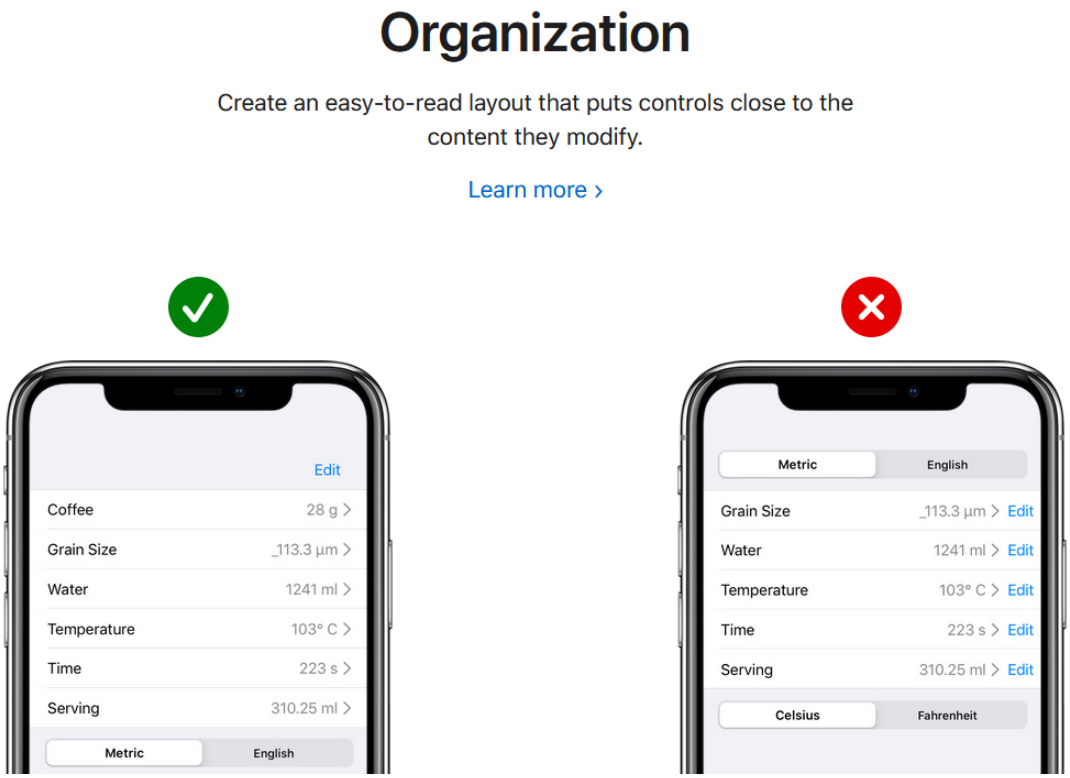
Always display images at their intended aspect ratio to avoid distortion.

Learn more ›



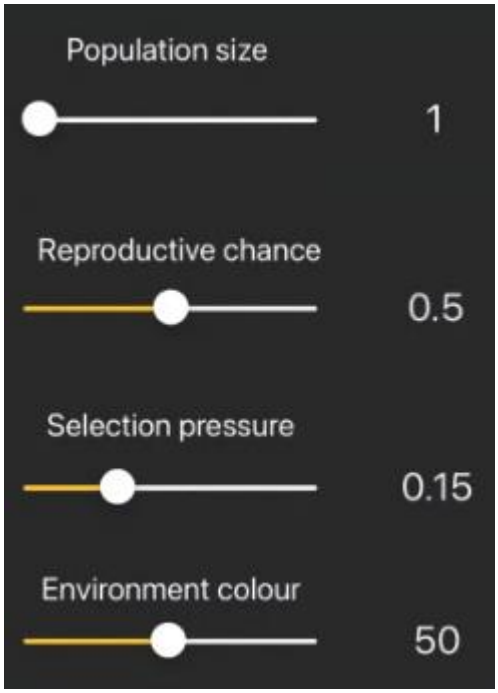
Once again, this usability feature does not apply.

9.



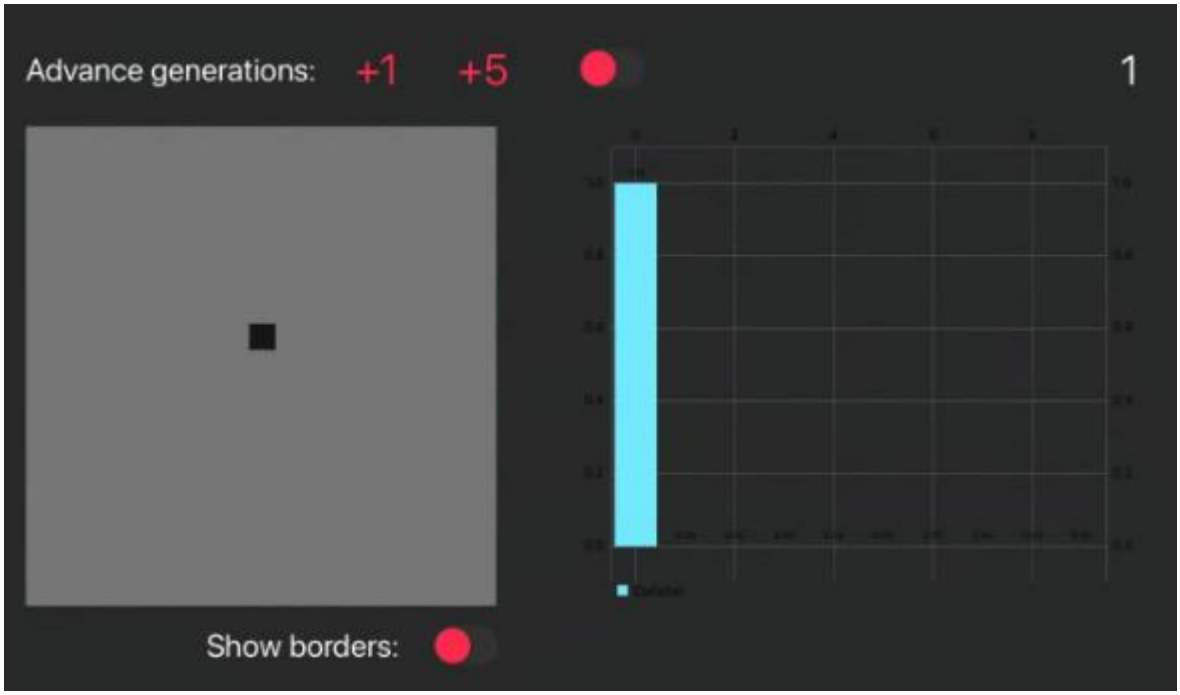
This usability feature is considered in several places throughout my app.

First, in the parameter input screen, each slider is directly below its descriptor and directly across from where its value is displayed:



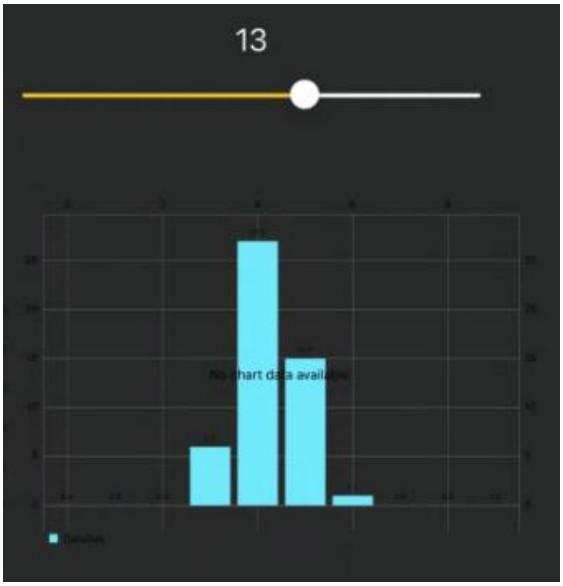
Each title is also physically close to its respective slider, making the layout very clear.

In the simulation screen, however, this usability feature is only a partial success. The “+1” and “+5” buttons are well placed, as is the switch to show borders, but the switch to the right of “Advance generations: ” is not labelled.



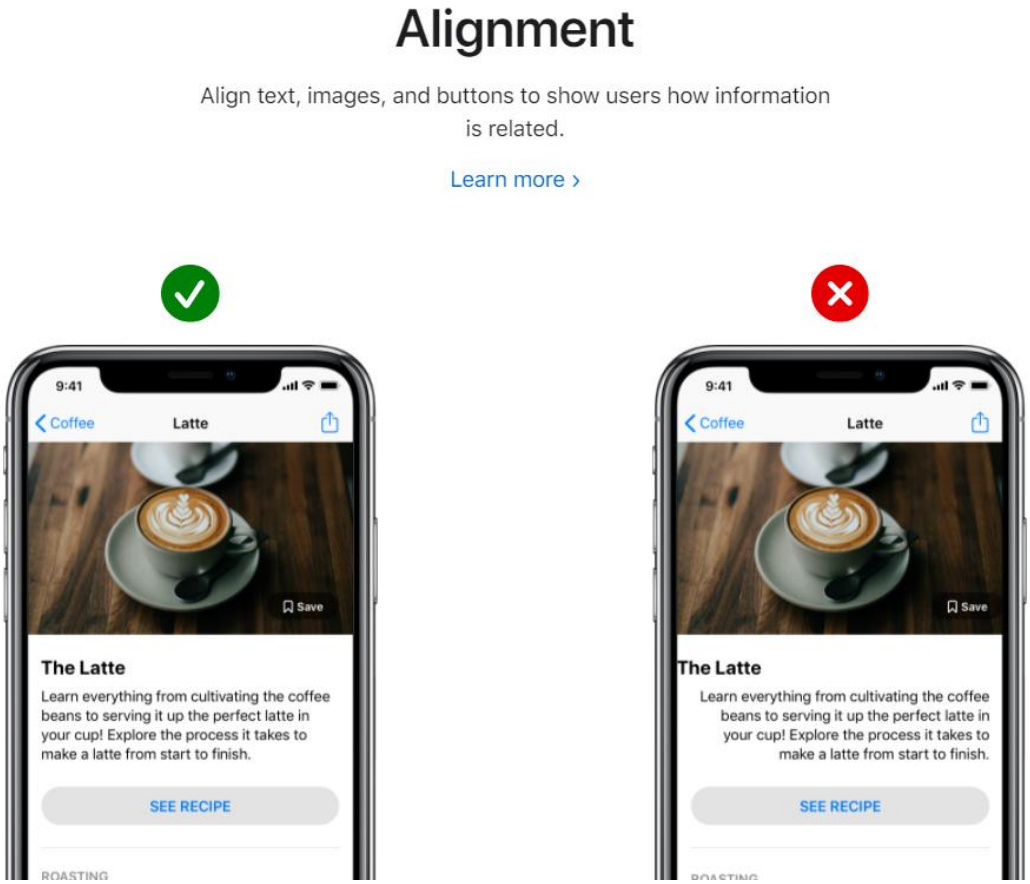
This could be addressed in further development by either adding an extra label on the same line or by placing the switch and an extra label on the line below to visually separate it from the static generation advance buttons.

Finally, in the summary screen, this usability feature is a failure as an effective feature. This is because there is no descriptor to tell the user what the slider does, so they have to figure it out by trial and error.



In further development I would add a label to the left of the value to explain to the user what the slider does. The label could read “Generation: ”.

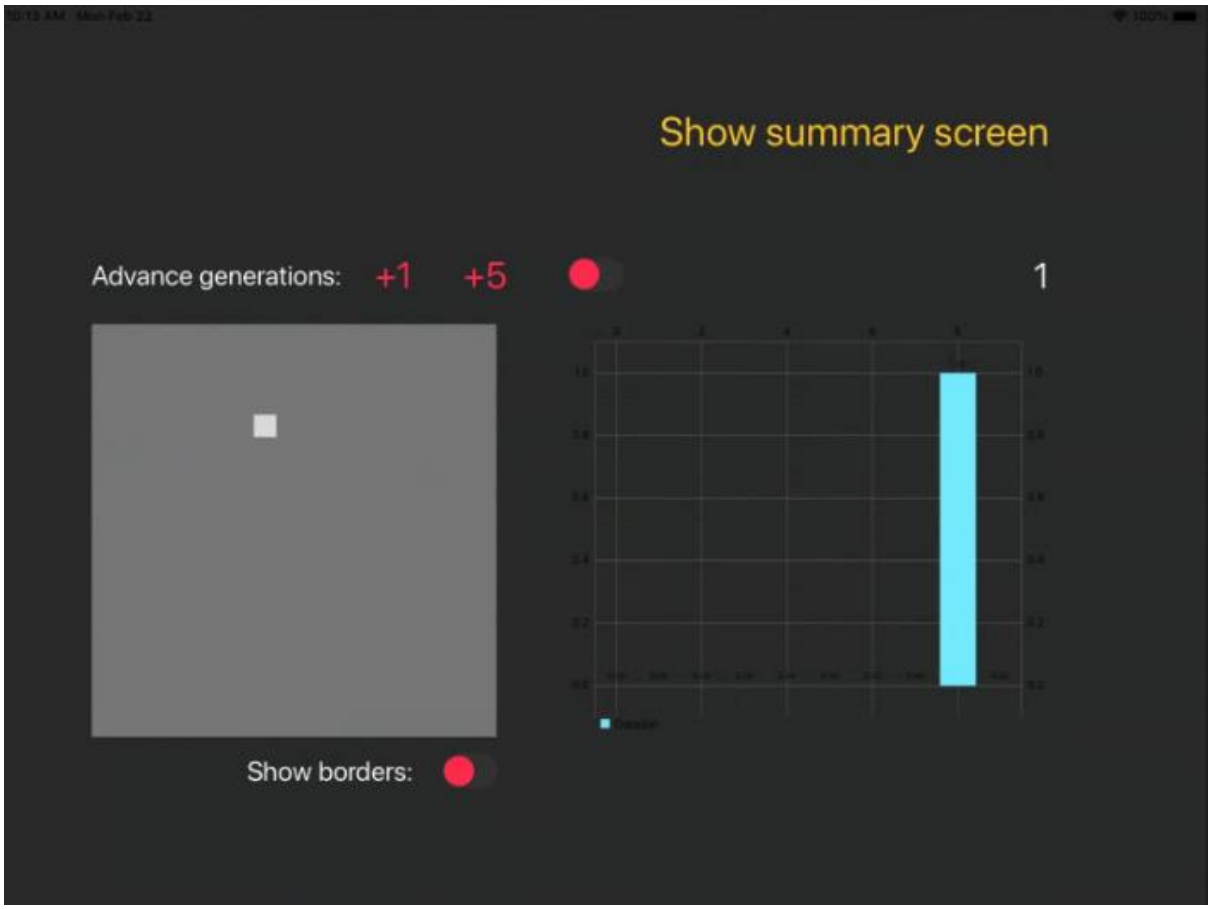
10.

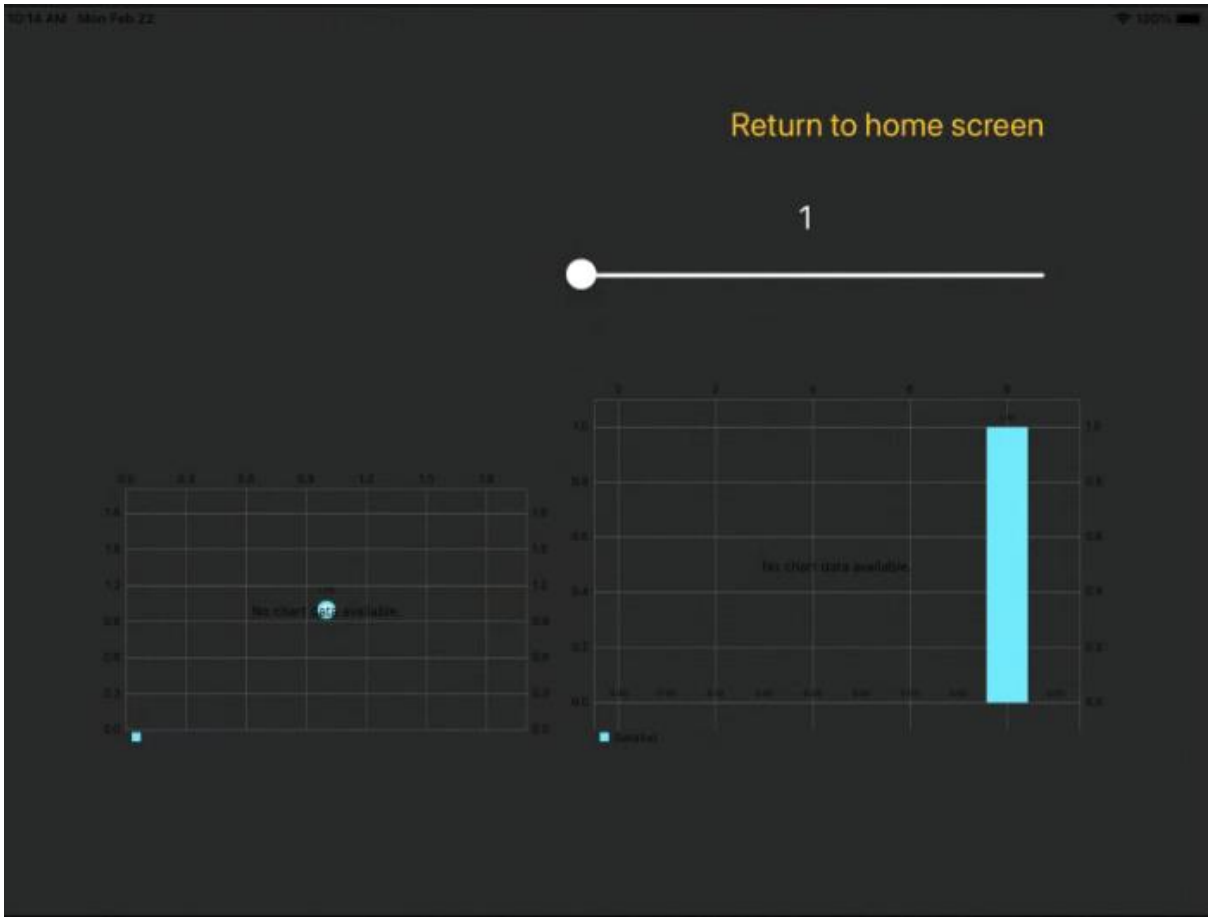


This usability feature is focused on positioning text, images and buttons to make the relationship between the objects intuitive, which makes navigating the ap much easier. This is considered in several places throughout my app.

First and foremost, I have placed the button to progress to the next screen (e.g. “Start simulation”) in the same place on each of the parameter input, simulation, and summary screens. The button is in the centre of the screen on the main menu to indicate to the user it is the main option.


Start
Population size
1
Reproductive chance
0.5
Selection pressure
0.15
Environment colour
50


Show summary screen
Advance generations:
+1
+5
1
Show borders:



Evaluation of solution

Success criteria review

In this section of my documentation, I will evaluate how effective the product I have built is using the success criteria listed earlier in my project. For each partially or unmet criteria I will comment on how that success criteria could be addressed in further development.

1. A series of options to see the evolution of different traits

As per the recommendation of Ms Hicks, my biology teacher stakeholder, I wanted there to be several possible traits the user could choose to see the evolution of. In my app, however, there is only one trait the user is able to select as shown in function test 18. Although there is only one option, i.e. the evolution of camouflage in pepper moths, the evolution process is covered in detail.

Conclusion: partially met.

To fully meet this success criteria in further development, I would implement a totally separate and non-physical trait, such as altruism, to demonstrate the evolution of. This would give the user a wider range of traits to see evolving which would in turn give them an intuitive understanding of how evolution can bring about behavioural changes.

2. A clearly labelled option in the menu for instructions

Instructions on how to use the app would make the app easier to use and make it more viable for use in a classroom. However, there are no instructions on the home screen as shown in function test 19.

Conclusion: not met.

To meet this criteria, I would add a set of simple instructions that outline what each screen is for, and a short description of how to use the parameter input screen.

3. An option in the menu for an explanation of evolution

An explanation of evolution in the app itself would make it a more self-contained learning tool. There is no explanation of evolution in the app. See function test 17.

Conclusion: not met.

This would be a very simple criteria to meet in further development. I would add a small button labelled “What is evolution?” to the main menu, which would open a popup containing a short summary of evolution.

4. After the user clicks on a trait, show a popup describing the circumstances of each trait’s development to provide context

The evolution of camouflage button is called “Evolution of camouflage in pepper moths”, which does provide some context and allows the user to research the animal themselves. However, there is no popup and there’s no in-depth explanation of what pepper moths are, so this success criteria is not fully met. See function test 20.

Conclusion: partially met.

Again, this would be a very simple criteria to meet during further development. I would simply display a popup window with the relevant context when the user clicks on a trait which the user can close whenever they would like.

5. Emphasise randomness of process in ‘explanation of evolution’ page and on the popup after selecting a trait

This should have been combined with criteria 3. As there is no explanation of evolution, this criteria is unmet. See function test 17.

Conclusion: not met.

In further development, I would add the explanation and emphasise the randomness of the process.

6. Be able to change at least one parameter of each simulation’s environment

In my simulation, there are four separate parameters the user can change – initial number of creatures, chance of reproduction each generation, selection pressure, and the environment colour. The environment colour parameter would be removed for the evolution of other traits, but the other three are general parameters that could apply to any simulation. See function tests 1, 5, 6, and 7 for tests related to the parameters.

Conclusion: fully met.

7. Traits described visually using colour in animation

I used grayscale instead of colour to describe the changing trait in the simulation to accommodate users with total colour blindness. However, the point of the success criteria was not to use colour specifically, but visually differentiating the creatures. See function test 21.

Conclusion: fully met.

8. Graphs visually near the simulation summarising the state of the population

The key graph I wanted in my app was a bar graph that summarised the distribution of the creatures, which I have in my app. I considered adding a line graph to represent the population as it changed, but the user interface became too cluttered with both graphs. I therefore removed it, and added it to the summary screen instead. See function tests 10, 11, and 12 for tests of the simulation screen bar graph, then tests 13 through 16 for tests of the bar and scatter graph in the summary screen.

Conclusion: fully met.

9. Uncluttered user interface

I have tried to keep the user interface as simple as possible so that the app is easy to use. I have done this by spacing different aspects of the UI to clearly separate them, and by placing related components near to each other. See more detail in “Usability tests” under Apple’s guidelines 1, 6, and 9.

Conclusion: partially met.

In further development I would add labels describing what certain buttons, switches, and sliders do that are not explicitly explained.

10. Starting the simulation changes the population according to the selected parameters

This is the core of the app, and the simulation works exactly as expected. See function tests 2, 3, and 4 for different tests of this.

Conclusion: fully met.

11. Graph updates in real time as simulation runs

The graph does update in real time as simulation runs. See function test 11.

Conclusion: fully met.

12. A clearly labelled button to stop the simulation at any time

This shouldn’t have been its own success criterion (I wouldn’t add it if I were doing the project again), but the button in the top right of the simulation screen that takes the user to the summary screen acts as a stop button.

Conclusion: fully met.

13. Interactive post-simulation window summarising the evolution of the selected trait

The summary screen is an interactive post-simulation window that summarises the evolution of the trait, but the graphs in the summary screen are not polished and it does not contain all the features I would like. For tests of the components of the summary screen, see function tests 13-16.

Conclusion: partially met.

In further development I would improve the summary screen by allowing the user to zoom into the scatter graph, change the amount of detail shown in the bar graph (increase the number of bars), and show the user a snapshot of the selected generation in a small window. I would also like to add a vertical bar to the scatter graph that moves left to right as the user selects different generations so it is clear where in the scatter graph a particular generation lies.

Limitations

The largest limitation of the current program is that the user can only see the evolution of one relatively simple trait. I wanted the user to be able to see the evolution of complex features and behaviours like kindness, perhaps even simulating two creatures at once like a predator-prey system.

The program is limited as a teaching tool because it doesn't explain what evolution is or what the program does. To be useful for students learning about evolution, they therefore need a teacher familiar with the software to explain what evolution is and then how to use the program.

Finally, the simulation is an over-simplification of real evolution in that the creatures start very different to one another then converge on an ideal form, all within a static environment. A more useful simplification would be that the creatures start well adapted to the environment, but the *environment then changes* and the creatures have to adapt. This is closer to how pepper moths evolved and better demonstrates evolution.

How to deal with limitations

A wider variety of traits could be added to the program with more development time. The underlying framework would be the same, so it would only require the details of the simulation to change for each trait.

An explanation of how evolution works and what the app does could easily be added with slightly more development time and an iterative approach to find how the information can be conveyed most clearly.

To more correctly depict the evolution of camouflage, I could use the same framework and allow the user to change the environment during the simulation. This would be the most flexible solution as the same framework can solve a wide variety of problems, but implementing this in practice may cause some difficulty where the discrete generations and continuous variation of the environment collide.

Maintenance

Because the program is built to be self-contained, it doesn't need to be maintained as external software (e.g. cloud-based file storage) is updated. However, if new versions of Swift are released the code will need re-writing to match the new syntax, and if new Apple devices are released the UI will need resizing to match the new screen sizes.

If new features are added or changes are made the code will of course need changing, but these changes will be relatively easy to make because the program is very modular. Also, the code is annotated to make maintenance easier, and the inputs and outputs of almost all subroutines are

commented so a new developer can understand the code and interact with the subroutines immediately.

Potential improvements

The most important thing to add to the program moving forward is an explanation of evolution and of what the app does. This would be easy to do as it does not require changing any part of how the program works, so it would just require an interview with a biology teacher and student, then adding the text. Once the text is added it does not need updating, as the mechanisms of evolution are very well established.

To make the graphs more useful, I would like to change the font colour in the graphs to white so the labels and scales of the axes are legible. This was specifically suggested by Ms. Hicks, who found the graphs less useful than they could be as a teaching tool because extracting specific data from the graph rather than trends is quite difficult.

In the longer term, it would be great to have a wider range of traits the user can select so they can get a better grip on how subtle traits evolve. It would also be nice (if very superficial) to have the creatures move in the simulation and to have them as smoother models than squares, created (for instance) in Blender or another graphics software toolset.

Overall conclusion

For the most part, this project has been extremely fun and rewarding. I have created a product I am proud of and I have learnt a huge amount in the process. However, due to the time constraints of the coursework, it hasn't been possible to completely finish the app and create what I had originally imagined. In particular, I would have liked to add an explanation of how evolution works and to show the evolution of a more abstract trait such as altruism.

If I were to restart the project, I would like to work more closely with my stakeholders to produce a solution that fits their needs as closely as possible. The iterative process would take more time because I would have to show each new feature to the stakeholders and potentially change it based on their feedback, but each part of the solution would be better suited to its purpose and I wouldn't spend time developing unimportant features.

I would also order the features I wanted to build so that the most important and useful features were built first. For instance, I would have spent more time making the app educational, as per the suggestion of my stakeholders, and less time implementing the borders around creatures, which was a useful feature but certainly wasn't vital. The main point of this app was to learn about simulations and using models of physical systems to gain some insight into the system, and I certainly achieved this.

Final code

GitHub repository

Link to GitHub repository: https://github.com/BeanVlasto/Programming-project.

This repository is currently private due to the security implications surrounding the coursework, but access to the repository can be requested by emailing me at “Vlasto.B@etoncollege.org.uk”.

Files

AnimationScene.swift

```
//
//  AnimationScene.swift
//  Evolution simulator
//
//  Created by Vlasto, Benedict (JDN) on 16/11/2020.
//

// This class controls what is displayed in the simulation page animation

import UIKit
import SpriteKit

class AnimationScene: SKScene {

    var environment: Environment?
    var borders: [SKShapeNode] = []

    required init?(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)
    }

    // This initialiser is called if a CGSize is provided in the AnimationScene call, which it always is
    override init(size: CGSize) {
        super.init(size: size)

    }

    func setBackgroundColour() {
        let colour = CGFloat(Double(environment!.environmentColour)/100)
        //backgroundColor = UIColor(red: 0.961, green: 0.784-0.259*colour, blue: 0.259, alpha: 1)
        backgroundColor = UIColor(red: colour, green: colour, blue: colour, alpha: 1)
    }

    func placeCreatures() {
        borders = []
        for child in children {
            child.removeFromParent()
        }
        for creature in environment!.creatures {
            let sprite = SKSpriteNode(color: colourSprite(creature: creature), size: CGSize(width: 20,
height: 20))
            sprite.position = randomSpritePosition()
            addChild(sprite)
            drawBorder(position: sprite.position, creature: creature)
        }
    }

    func randomSpritePosition() -> CGPoint {
        let xPos = CGFloat(arc4random_uniform(UInt32(self.size.width - 20))) + 10
        let yPos = CGFloat(arc4random_uniform(UInt32(self.size.height - 20))) + 10

        return CGPoint(x: xPos, y: yPos)
    }

    func colourSprite(creature: Creature) -> UIColor {
        let colour = CGFloat(Double(creature.traitValue) / 100)
        //let colour = UIColor(red: 0.961, green: 0.784-0.259*colourValue, blue: 0.259, alpha: 1)
        let spriteColour = UIColor(red: colour, green: colour, blue: colour, alpha: 1)
        return spriteColour
    }

    func drawBorder(position: CGPoint, creature: Creature) {
        // Box around creature has same dimensions as creature
        let borderNode = SKShapeNode(rectOf: CGSize(width: 20, height: 20))
        // Box is centred on the creature
        borderNode.position = position
```

```
        borderNode.fillColor = .clear
        borderNode.lineWidth = 2
        borderNode.strokeColor = .clear

        // checks if creature colour is within 1 of the environment background
        if -1...1 ~= creature.traitValue - environment!.environmentColour {
            borderNode.strokeColor = .red
        }

        addChild(borderNode)
        borders.append(borderNode)
    }

    func hideBorders() {
        for border in borders {
            border.isHidden = true
        }
    }

    func showBorders() {
        for border in borders {
            border.isHidden = false
        }
    }

}
```

Creature.swift

```
//
//  Creature.swift
//  Evolution simulator
//
//  Created by Vlasto, Benedict (JDN) on 19/09/2020.
//

import Foundation

class Creature {

    // Trait value is an integer between 1 and 100 that represents how much of the trait being evolved a
creature has, such as how fast/slow a creature is.
    var traitValue: Int

    // traitValue is not passed in when creatures are first initialised, but is passed in when creatures
reproduce.
    // This is why traitValue is optional.
    init(traitValue: Int?) {
        if let creatureTraitValue = traitValue {
            self.traitValue = creatureTraitValue
        } else {
            self.traitValue = Int.random(in: 1...100)
            //self.traitValue = 90
        }
    }

    func reproduce() -> Creature {

        let mutationVariable = Int.random(in: -10...10)
        var childTraitValue = self.traitValue + mutationVariable

        // if statements handle the case of childTraitValue being outside bounds of 1...100
        if childTraitValue > 100 {
            childTraitValue = 200 - (mutationVariable + self.traitValue)
        } else if childTraitValue < 1 {
            childTraitValue = 2 - (mutationVariable + self.traitValue)
        }

        let newCreature = Creature(traitValue: childTraitValue)
        return newCreature

    }

}
```

Environment.swift

```
//
//  Environment.swift
//  Evolution simulator
//
//  Created by Vlasto, Benedict (JDN) on 21/09/2020.
//
```

```
import Foundation

class Environment {

    // Reproductive chance is a double between 0 and 1 that represents the chance a creature will
reproducee each generation. It is a constant per simulation.
    let reproductiveChance: Double

    // environmentColour is an integer in range 1...100. This is what the creatures' traitValues are
compared against. If creature traitValue is very far from environmentColour, it is more likely to die.
    let environmentColour: Int

    // selectionPressure changes how punishing an environment is, and so how narrow/wide a creature
distribution will be. Higher value of selection pressure gives narrower distribution.
    let selectionPressure: Double

    // Stores each generation of creatures during simulation
    var creatures: [Creature] = []

    // This is the array used in the summary screen to access the entire history of creatures. This allows
every generation's distribution to be displayed in the summary screen bar chart.
    var populationHistory: [[Creature]] = []

    init(environmentColour: Int, reproductiveChance: Double, selectionPressure: Double, populationSize:
Int) {
        self.environmentColour = environmentColour
        self.reproductiveChance = reproductiveChance
        self.selectionPressure = selectionPressure
        for _ in 1...populationSize {
            let creature = Creature(traitValue: nil)
            self.creatures.append(creature)
        }
        self.populationHistory.append(self.creatures)
    }

    func calculateCreaturesAdaptation(creature: Creature) -> Int {
        var difference = creature.traitValue - self.environmentColour
        if difference < 0 {
            difference = -difference
        }
        let adaptation = 100 - difference
        return adaptation
    }

    func calculateChanceOfDeath(creature: Creature) -> Double {
        let adaptation = Double(calculateCreaturesAdaptation(creature: creature))
        let dampingFactor = 0.2 * Double(self.creatures.count)
        let denominator = 1 + pow(2.71828, -self.selectionPressure * (adaptation - (80 + dampingFactor)))
        let chanceOfSurvival = 1/denominator
        return 1 - chanceOfSurvival
    }

    func advanceOneGeneration() -> String? {

        guard self.creatures.count > 0 else {
            return "extinct"
        }

        // Killing creatures
        var tempArray: [Creature] = []
        for i in 0 ..< self.creatures.count {
            let fraction = Double.random(in: 0...1)
            let chanceOfDeath = calculateChanceOfDeath(creature: self.creatures[i])
            if fraction > chanceOfDeath { // then keep creature alive
                tempArray.append(self.creatures[i])
            }
        }
        self.creatures = tempArray

        guard self.creatures.count > 0 else {
            return "extinct"
        }

        // Reproducing creatures
        for creature in self.creatures {
            let fraction = Double.random(in: 0...1)
            if fraction < self.reproductiveChance {
                self.creatures.append(creature.reproduce())
            }
        }

        self.populationHistory.append(self.creatures)
        return nil

    }

}
```

ParameterInputViewController.swift

```
//
//  ParameterInputViewController.swift
//  Evolution simulator
//
//  Created by Vlasto, Benedict (JDN) on 27/09/2020.
//

import UIKit

class ParameterInputViewController: UIViewController {

    // All variables and IBOutlets manage the sliders being changed in parameter input screen
    var populationSize: Int = 1
    @IBOutlet weak var populationSizeSlider: UISlider!
    @IBOutlet weak var populationSizeLabel: UILabel!

    var reproductiveChance: Double = 0.5
    @IBOutlet weak var reproductiveChanceSlider: UISlider!
    @IBOutlet weak var reproductiveChanceLabel: UILabel!

    var selectionPressure: Double = 0.15
    @IBOutlet weak var selectionPressureSlider: UISlider!
    @IBOutlet weak var selectionPressureLabel: UILabel!

    var environmentColour: Int = 50
    @IBOutlet weak var environmentColourSlider: UISlider!
    @IBOutlet weak var environmentColourLabel: UILabel!

    override func viewDidLoad() {
        super.viewDidLoad()

        environmentColourSlider.value = 50
    }

    // Button that shows simulation screen
    @IBAction func showSimulationVC(_ sender: Any) {

        let environment = Environment(environmentColour: environmentColour, reproductiveChance:
self.reproductiveChance, selectionPressure: self.selectionPressure, populationSize: self.populationSize)

        if let vc = storyboard?.instantiateViewController(identifier: "simulationVC") as?
SimulationViewController {
            vc.environment = environment
            vc.modalPresentationStyle = .fullScreen
            present(vc, animated: true, completion: nil)
        }
    }

    // All IBActions here manage sliders being changed in parameter input screen
    @IBAction func populationSizeChanged(_ sender: UISlider) {
        let value = Int(sender.value)
        self.populationSize = value
        populationSizeLabel.text = String(value)
    }

    @IBAction func reproductiveChanceChanged(_ sender: UISlider) {
        let value = Double(sender.value)
        let roundedValue = round(value * 100)/100
        self.reproductiveChance = value
        reproductiveChanceLabel.text = String(roundedValue)
    }

    @IBAction func selectionPressureChanged(_ sender: UISlider) {
        let value = Double(sender.value)
        let roundedValue = round(value * 100)/100
        self.selectionPressure = value
        selectionPressureLabel.text = String(roundedValue)
    }

    @IBAction func environmentColourChanged(_ sender: UISlider) {
        let value = Int(sender.value)
        self.environmentColour = value
        environmentColourLabel.text = String(value)
    }

}
```

SimulationSummaryViewController.swift

```
//
//  SimulationSummaryViewController.swift
//  Evolution simulator
//
//  Created by Vlasto, Benedict (JDN) on 27/09/2020.
//

import UIKit
import Charts

class SimulationSummaryViewController: UIViewController {

    // stores the Environment from the simulation screen
    var environment: Environment?

    // For bar chart
    @IBOutlet weak var displayGenerationNumber: UILabel!
    let xValues: [String] = ["1-10", "11-20", "21-30", "31-40", "41-50", "51-60", "61-70", "71-80", "81-
90", "91-100"]
    @IBOutlet weak var generationSelector: UISlider!
    @IBOutlet weak var simulationSummaryBarChartView: BarChartView!

    // For line chart
    @IBOutlet weak var simulationSummaryLineChartView: LineChartView!

    override func viewDidLoad() {
        super.viewDidLoad()

        // Sets slider bounds, bar chart, and line chart
        generationSelector.minimumValue = 1
        generationSelector.maximumValue = Float(environment!.populationHistory.count)
        setBarChart(generation: 1)
        setLineChart()
    }

    func setBarChart(generation: Int) {

        displayGenerationNumber.text = String(generation)

        var yValues = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
        var dataEntries: [BarChartDataEntry] = []

        for creature in environment!.populationHistory[generation-1] {
            if 1...10 ~= creature.traitValue {
                yValues[0] += 1
            } else if 11...20 ~= creature.traitValue {
                yValues[1] += 1
            } else if 21...30 ~= creature.traitValue {
                yValues[2] += 1
            } else if 31...40 ~= creature.traitValue {
                yValues[3] += 1
            } else if 41...50 ~= creature.traitValue {
                yValues[4] += 1
            } else if 51...60 ~= creature.traitValue {
                yValues[5] += 1
            } else if 61...70 ~= creature.traitValue {
                yValues[6] += 1
            } else if 71...80 ~= creature.traitValue {
                yValues[7] += 1
            } else if 81...90 ~= creature.traitValue {
                yValues[8] += 1
            } else if 91...100 ~= creature.traitValue {
                yValues[9] += 1
            }
        }

        for i in 0 ..< xValues.count {
            let dataEntry = BarChartDataEntry(x: Double(i), y: Double(yValues[i]))
            dataEntries.append(dataEntry)
        }

        let chartDataSet = BarChartDataSet(dataEntries)
        let chartData = BarChartData(dataSet: chartDataSet)
        simulationSummaryBarChartView.data = chartData

    }

    func setLineChart() {

        var xValues: [Int] = []
        for i in 1 ... environment!.populationHistory.count {
            xValues.append(i)
        }

        var yValues: [Int] = []
```

```
        for i in 0 ..< environment!.populationHistory.count {
            yValues.append(environment!.populationHistory[i].count)
        }

        var dataEntries: [ChartDataEntry] = []
        for i in 0 ..< xValues.count {
            let dataEntry = ChartDataEntry(x: Double(xValues[i]), y: Double(yValues[i]))
            dataEntries.append(dataEntry)
        }

        let lineChartDataSet = LineChartDataSet(entries: dataEntries, label: nil)
        let lineChartData = LineChartData(dataSet: lineChartDataSet)
        simulationSummaryLineChartView.data = lineChartData

    }

    // Called when generationSelector slider is changed
    @IBAction func selectedGenerationChanged(_ sender: UISlider) {
        let value = Int(sender.value)
        setBarChart(generation: value)
    }

    //
    @IBAction func showMainMenuVC(_ sender: Any) {
        if let vc = storyboard?.instantiateViewController(identifier: "mainMenuVC") as? ViewController {
            vc.modalPresentationStyle = .fullScreen
            present(vc, animated: true, completion: nil)
        }
    }


}
```

SimulationViewController.swift

```
//
//  SimulationViewController.swift
//  Evolution simulator
//
//  Created by Vlasto, Benedict (JDN) on 27/09/2020.
//

import UIKit
import Charts
import SpriteKit

class SimulationViewController: UIViewController {

    // stores the Environment initialised in parameter input screen
    var environment: Environment?

    // timer used for automatic advance of generations
    var timer: Timer?

    // Displays population size
    @IBOutlet weak var populationSizeLabel: UILabel!

    // For bar chart
    let xValues: [String] = ["1-10", "11-20", "21-30", "31-40", "41-50", "51-60", "61-70", "71-80", "81-
90", "91-100"]
    @IBOutlet weak var simulationBarChartView: BarChartView!


    // For animations
    @IBOutlet weak var skView: SKView!
    var scene: AnimationScene?

    override func viewDidLoad() {
        super.viewDidLoad()

        updateViews()
    }

    override func viewDidAppear(_ animated: Bool) {
        scene = AnimationScene(size: skView.bounds.size)
        scene!.environment = environment
        scene!.setBackgroundColour()
        scene!.placeCreatures()
        scene!.hideBorders()
        self.skView.presentScene(scene)
    }

    // Updates bar chart and population size label
    func updateViews() {
        setChart()
```

```
        populationSizeLabel.text = String(self.environment!.creatures.count)
    }

    func setChart() {

        var yValues = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
        var dataEntries: [BarChartDataEntry] = []

        for creature in environment!.creatures {
            if 1...10 ~= creature.traitValue {
                yValues[0] += 1
            } else if 11...20 ~= creature.traitValue {
                yValues[1] += 1
            } else if 21...30 ~= creature.traitValue {
                yValues[2] += 1
            } else if 31...40 ~= creature.traitValue {
                yValues[3] += 1
            } else if 41...50 ~= creature.traitValue {
                yValues[4] += 1
            } else if 51...60 ~= creature.traitValue {
                yValues[5] += 1
            } else if 61...70 ~= creature.traitValue {
                yValues[6] += 1
            } else if 71...80 ~= creature.traitValue {
                yValues[7] += 1
            } else if 81...90 ~= creature.traitValue {
                yValues[8] += 1
            } else if 91...100 ~= creature.traitValue {
                yValues[9] += 1
            }
        }

        for i in 0 ..< xValues.count {
            let dataEntry = BarChartDataEntry(x: Double(i), y: Double(yValues[i]))
            dataEntries.append(dataEntry)
        }

        let chartDataSet = BarChartDataSet(dataEntries)
        let chartData = BarChartData(dataSet: chartDataSet)
        simulationBarChartView.data = chartData

    }

    // Called by advanceOneGeneration() if population size hits 0
    func extinctionAlert() {
        let alert = UIAlertController(title: "Extinction!", message: "This species of creature didn't adapt
fast enough - it is now extinct.", preferredStyle: .alert)
        alert.addAction(UIAlertAction(title: "Show summary screen", style: .default, handler: { action in
            self.showSummaryVC()
        }))
        self.present(alert, animated: true)
    }

    func advanceOneGeneration() {
        if environment?.advanceOneGeneration() == "extinct" {
            extinctionAlert()
        } else {
            updateViews()
            scene!.placeCreatures()
            if showBorders.isOn == false {
                scene!.hideBorders()
            }
        }
    }

    // Button that calls advanceOneGeneration once
    @IBAction func advanceOneGenerationButton(_ sender: Any) {
        advanceOneGeneration()
    }

    // Button that calls advanceOneGeneration five times
    @IBAction func advanceFiveGenerations(_ sender: Any) {
        for _ in 1...5 {
            advanceOneGeneration()
        }
    }

    // Button that starts/stops auto advance of generations
    @IBAction func startStopAutoAdvance(_ sender: Any) {
        if timer == nil {
            timer = Timer.scheduledTimer(withTimeInterval: 1, repeats: true) { timer in
                self.advanceOneGeneration()
            }
        } else {
            timer!.invalidate()
            timer = nil
        }
    }
```

```
    // Shows or hides the sprite borders
    @IBOutlet weak var showBorders: UISwitch!
    @IBAction func showOrHideBorders(_ sender: Any) {
        if showBorders.isOn == true {
            scene!.showBorders()
        } else {
            scene!.hideBorders()
        }
    }

    // Button that ends simulation and shows summary screen
    @IBAction func showSummaryVC(_ sender: UIButton? = nil) {
        if let vc = storyboard?.instantiateViewController(identifier: "simulationSummaryVC") as?
SimulationSummaryViewController {
            if timer != nil {
                timer!.invalidate()
                timer = nil
            }
            vc.environment = environment
            vc.modalPresentationStyle = .fullScreen
            present(vc, animated: true, completion: nil)
        }
    }

}
```

ViewController.swift

```
//
//  ViewController.swift
//  Evolution simulator
//
//  Created by Vlasto, Benedict (JDN) on 19/09/2020.
//

// This is the view loaded when the app first runs

import UIKit

class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()

        // Do any additional setup after loading the view.
    }

    // Button to access parameter input screen
    @IBAction func showParameterInputVC(_ sender: Any) {
        if let vc = storyboard?.instantiateViewController(identifier: "parameterInputVC") as?
ParameterInputViewController {
            vc.modalPresentationStyle = .fullScreen
            present(vc, animated: true, completion: nil)
        }
    }

}
```