EGYPT
Farmers to Pharoahs

Derrick Diana
Harry Heathcock
Kaedon Jon Williams

September 10, 2019

Abstract

1 Introduction

Studying ancient civilisations can be difficult due to incomplete information and an inability to test hypothoses. This is a problem which can be partially solved by simulation, if the simulation can be shown to be accurate. A programme was created to reimplement an older simulation model for ancient Egyptyian farming communities in a new language to allow for extensibility and speed improvements, as well as improving the user experience. Another goal was to test whether the model was resilient to slight changes that would be resultant from a reimplementation.
The simulation itself could provide assistance to researchers attempting to study ancient Egypt, allowing them to view how the conditions that were discovered came about in the first place. This could provide insight as to whether slight changes in initial conditions would create a different environment entirely.
The simulation itself is an agent based model, where households claim and farm fields around the Nile River for the grain which they need to survive. Households are each part of a Settlement, which can contain many households. Households grow and thrive, while others die out.
The design process used followed a few steps. The first step was to read and understand the original source code, collecting required functionality and trying to piece together how the system worked as a whole. Once the system was understood, step two began, in which the basic pieces were implemented into an object-oriented python version. This was split into the simulation and the UI, with the simulation very closely following the original code and the UI using the original as a style guide. Once the basics of the simulation and the UI had been created, a set of available data was agreed on and the two systems were brought together. The UI code instantiated a copy of the simulation code, pulled the data required for rendering each year and called the function which causes the simulation to continue for another year. From there, the final step was just to continue programming features into the simulation and UI seperately until both fulfilled the original requirements.

2 Requirements

2.1 Functional

2.2 Non-Functional

2.3 Usability

2.4 Use Cases

3 Design Overview

The programme was designed such that the only interactions between the simulation itself and the UI are in instantiation, the increase of years and the current state of the simulation. This was done for encapsulation purposes and to ensure the integrity of the simulation at all times. The display is split up into the UI and the rendering of the simulation, this was performed for ease of upkeep, as combining the two would result in a monolithic structure which is difficult to work with and maintain. The basic structure is shown in Figure 1.
Due to this being a reimplementation of work that has already been done, the algorithms and data structures were pre-determined. All changes made to the overall algorithm were due to efficiency concerns, or strange behaviour which was corrected.
Data was stored almost exclusively in lists, as order needed to be randomised each tick to remove possible biases due to ordering of operations being performed on the households.

3.1 Changes

A number of changes in the implementation were made, some due to what seemed to be implementation errors in the original code and others due to functionality not making sense with what was trying to be achieved. Those which affect functionality have been implemented as explained, however when the Legacy Mode setting is enabled, these changes are reverted back to how they were in the original implementation.

3.1.1 Death Due to Lack of Grain

In the original NetLogo code, if a household did not have enough grain to survive a year then a single person would die and the grain would be set to zero. In reality this doesn't make sense, as if a household has no food, then no one would survive. The reimplementation changed this so that only the family members who can be fed will survive.

3.1.2 Population Increases

In the original code, if the total current population is less than or equal to the historical projected population then the population can increase. At first glance this seems to be correct, however this allows the population to always grow slightly faster than the projected historical value, and thus in the reimplementation this is changed so that the population will only increase if it is less than the historical projected population.

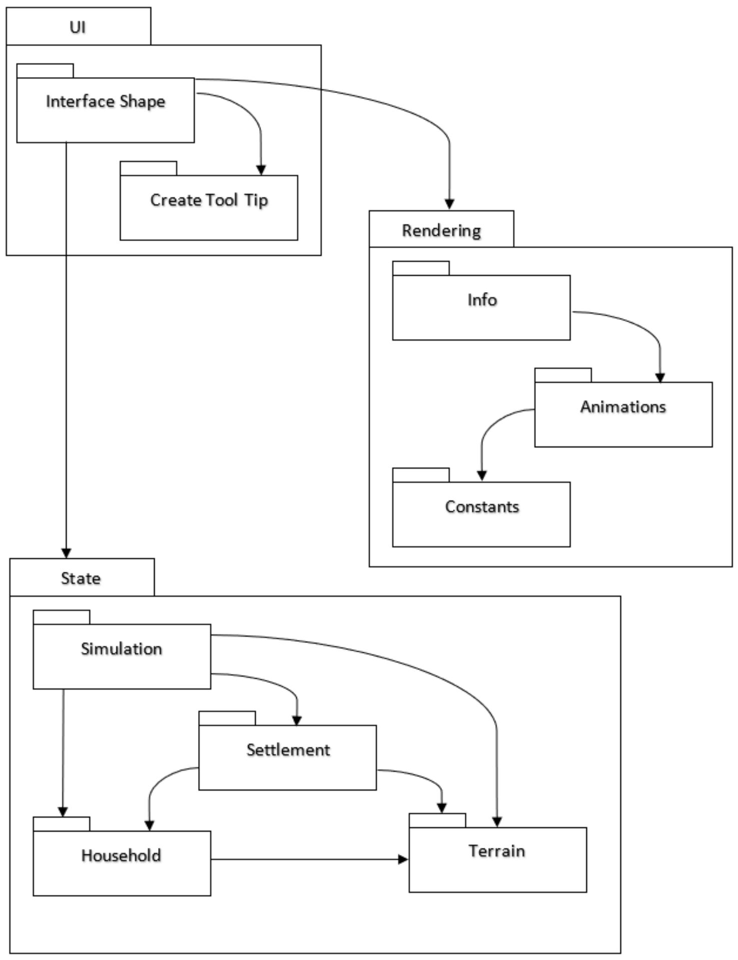


Figure 1: Architecture diagram for the system, showing how the UI checks the state of the simulation and passes this to be rendered.

3.1.3 Generational Changes in Ambition and Competency

In the original code, if the ambition or competency for a household moved out of the bounds set (minimum, up to 1) then a new value would be generated until it fell within the bounds. This means that the computational time for the ambition and competency changes are unbounded. In the reimplementation, a value within the possible range is generated.

3.1.4 Order of Field Claimants

In the original code, households claimed fields in ascending order of how much grain they currently have. This seemed inconsistent with every other operation being independant of grain or ordering, in the reimplementation, this ordering is randomised each year.

4 Implementation

As can be seen in Figure 2, InterfaceShape is the controller class which invokes other methods in all other classes. InterfaceShape contains the code which sets up the GUI, which consists of the components explained below.
A canvas object on which the main simulation grid is drawn. The main simulation grid displays the households, fields and rivers distinctly and clearly. A panel, that consists of slider and

check-box objects, which allows the user to initialise parameters for the simulation. Two FigureCanvasTkAgg() objects which act as widgets and each create a matplotlib-style interactive graph. These graphs can be zoomed, scrolled, and can save their displays as images. These are live representations of figure objects within the matplotlib.pyplot library. Several other buttons and sliders which allow the user to interact with the program in minor ways.

The GUI has a method called MainLoop, which performs the primary actions corresponding to one year of the simulation which includes calling the following methods, in order:

- Simulation.tick()
  Which causes the simulation class to simulate a single year, generating and setting fertility values for the entire set of land, allowing households to claim, farm and rent land, as well as consuming grain, dying out when not enough food is present, allowing for population growth as well as shifts in the household competency and ambition levels.

- Info.drawGridSimulation()
  Which accesses the simulation object, finding the positions and values for fields and settlements in order to render them on the canvas correctly.

- Info.plotData()
  Which reads the necessary data out of the simulation object and appends all the data needed for each graph to a list which stores the lists required for each graph. This outer list consists of 11 inner list, each corresponding to a single graph.

- Info.updateGraphs()
  Which takes the data which has already been stored and adds it to the pyplot.figure objects which the FigureCanvasTkAgg() represent. In order to reduce the time taken to draw all the points, only points which have been added since the last draw are drawn each cycle. This is performed until the user decides to change which graph is being displayed, in which case the entire graph must be redrawn, or many lines have already been drawn, in which case the graph is redrawn as a single line. The second part is a speed optimisation, as plotting many points at once is a slow process, however the fragmentation that is created when many smaller lines have been drawn can cause the process to slow even further.

- Info.showGraphs()
  Which calls show on both of the FigureCanvasTkAgg() objects to show the data to the user.

The MainLoop method then calls itself again in a non-recursive manner using the tkinter method after(), which calls the MainLoop function again without blocking after a given delay in milliseconds. This number of milliseconds for the delay is determined by a slider that is present in the GUI and the delay is calculated each frame to achieve the desired frames per second. If the desired framerate cannot be achieved, then the programme does not sleep.

5 Test Cases

6 Conclusions

The application as designed seems to run much faster than its predecessor, while also being written in a more extensible manner. For clients, there are quality of life features that were implemented which were not in the original, for example being able to set up multiple runs of the same settings to automatically continue as well as being able to easily save graphs which are

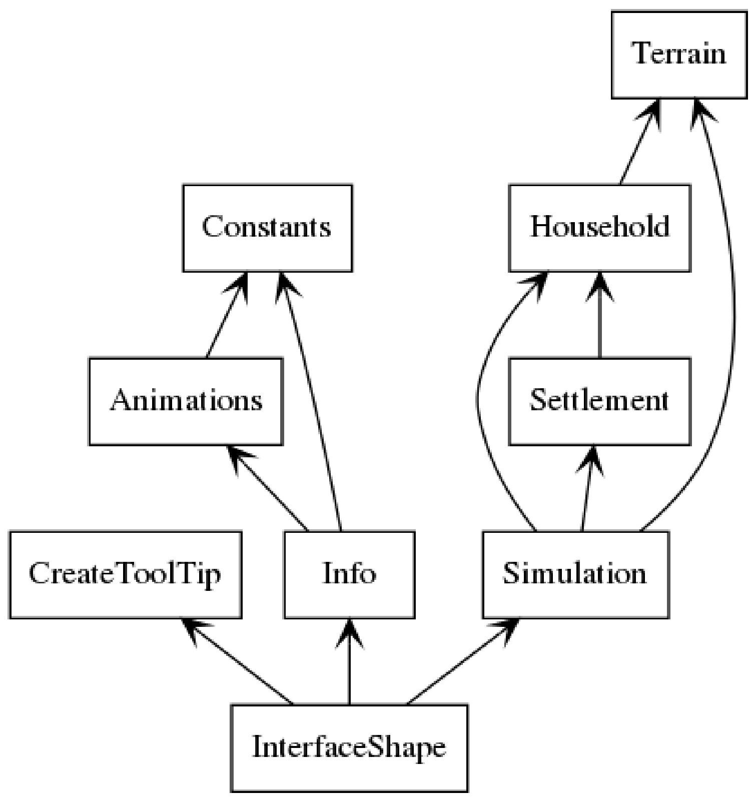


Figure 2: Diagram showing the relationship between all the classes involved in the programme.

generated. In the original, a single graph could be saved at a time. In the reimplementation, all graphs could be saved at once, or a graph could be zoomed in on before being saved.

References

A Manual

A.1 Sliders

The left hand side of the interface contains a set of sliders which may be adjusted to give different parameters to the program. On adjustment, the sliders automatically reset the simulation with the changed values. Below sliders are check boxes, which function similarly - resetting the simulation on being changed. One check box asks the users for a seed and then passes this to the simulation.

A.2 Simulation display

The central portion of the interface displays the current state of the simulation, in which the user may right click any block to see a description of the current parameters associated with that block. To clear the display of information when no longer required, the user may left click. The information displayed depends on what type of block was clicked on, as follows:

A.2.1 Settlement

Settlements are represented by houses, where the size of the house represents the total population of the settlement.

When right clicked on, settlements display their total population, and how many households are contained within.

A.2.2 Fields

Fields are represented by a circle (unharvested) or a piece of barley. The colour of the symbol is related to the colour of the household which has claimed the field, with purple representing the top 33% of wealth, blue representing the middle 33% and yellow representing the bottom 33%. When right clicked on, fields will display whether or not they have been harvested, their fertility, their potential harvest for the year and the number of years it has been since the field has been harvested.

A.2.3 Land

The shade of green of each of the block of represents its fertility, where a darker colour means that it is more fertile. Right clicking on a piece of land displays its fertility and what its potential harvest would be if it were to be claimed and harvested.

A.3 Graphs of data

Two displays are available on the right hand side of the interface. Each is controlled by a drop down box beneath it which when clicked allows the user to choose one of the eleven graphs to be displayed. There is also a tool bar underneath each graph which allows the user to interact with the graph through actions such as panning or zooming. To save an individual graph one may click the save button in the toolbar. To save all the graphs into a folder of your choosing, click Save all figures and enter the name of the folder.

B Technical API Manual

This section will assume a familiarity with how the simulation is meant to function as well as the code of Simulation, Settlement, Household and Terrain, although a brief description of their functionality will be described. If additional variables are required in Settlement, Household or Terrain, note that these variables must first be added to the "__slots__" list at the top of the class in order to be used. This is to provide speedup to accessing variables and reduce memory usage overall and is opposed to the usual python dictionary which is used to store variables within classes.

B.1 Simulation

The Simulation class forms a wrapped for all of the other simulation objects, holding data that is required globally throughout. It also controls some of the order of operations. For extensibility, the simulation class would probably not provide much help outside of changes to instantiation. If one were to want different households or settlements to run with different parameters, this is where one would make the changes.

B.2 Settlement

The Settlement class holds the households which exist in a specific area together, controls the removal of households when their population reaches 0 and ensures that the households within run their yearly simulations in a random order. If one were to want to change the order in which the simulation is run, or change behaviour of household removal, this is the file in which changes should be made.

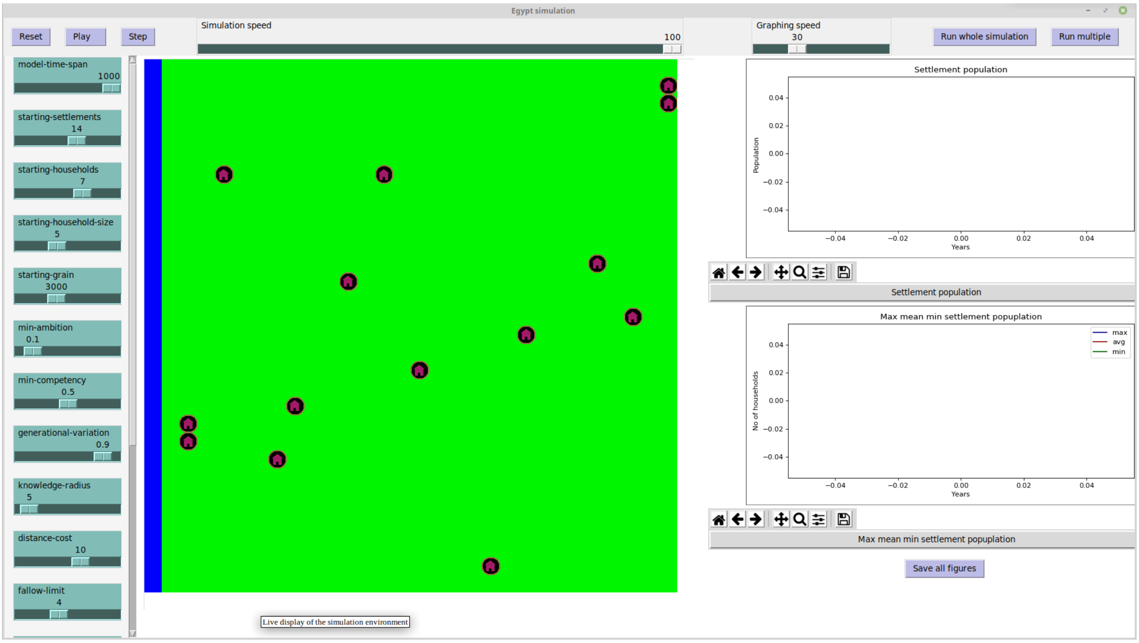
B.3 Household

The Household class controls the year to year activities of individual houses, including farming, using food, population increases and changes to ambition and competency over generations. If one were to want to change any general behaviour of the individual households, this is likely to be the correct area to look. Due to how python treats functions, functions can be replaced in certain households without affecting how all households function. If this is done, it must be ensured that the population of the settlement and total population are adjusted when the population is adjusted, and that the grain of the household at the end of the year is added to the total grain.

B.4 Terrain

The Terrain class maintains information about the state of the land in a specific position, as well as containing information about what its position is. It is unlikely that extensibility would come from altering this class, but if it is necessary, then it must be ensured that the years not harvested is incremented each year, harvested is set to false at the start of the year, the fertility and harvest are set every year, and the claim and unclaim functions need to maintain their current functionality.

C Screenshots



Simulation speed 100

Graphing speed 30

model-time-span 1000

starting-settlements 14

starting-households 7

starting-household-size 5

starting-grain 3000

min-ambition 0.1

min-competency 0.5

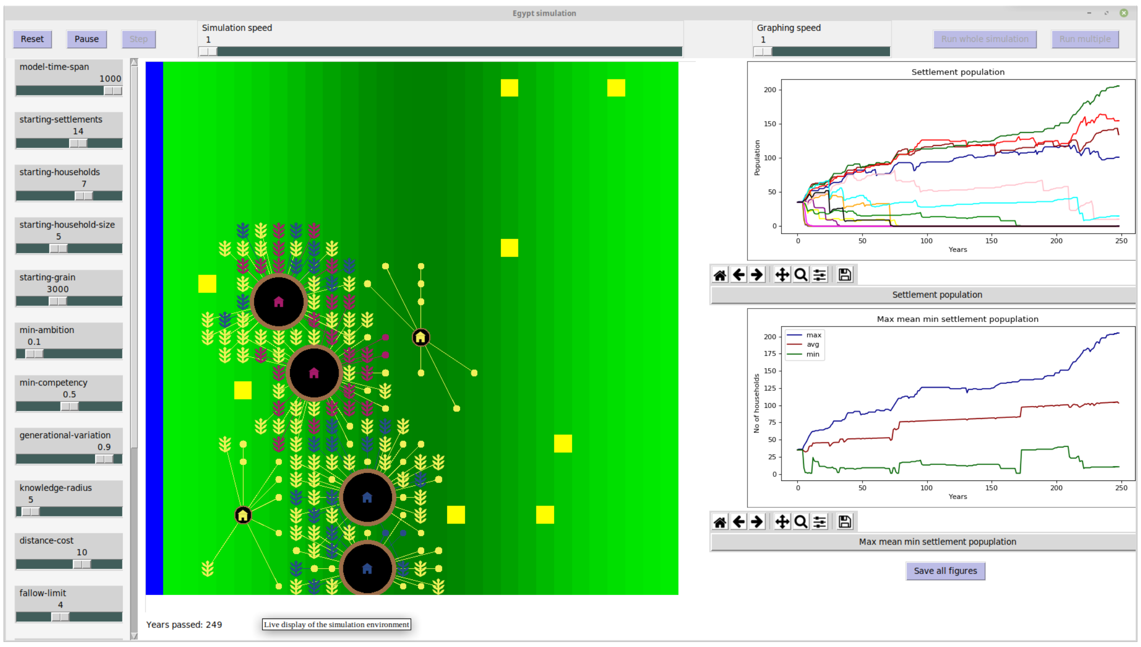
generational-variation 0.9

knowledge-radius 5

distance-cost 10

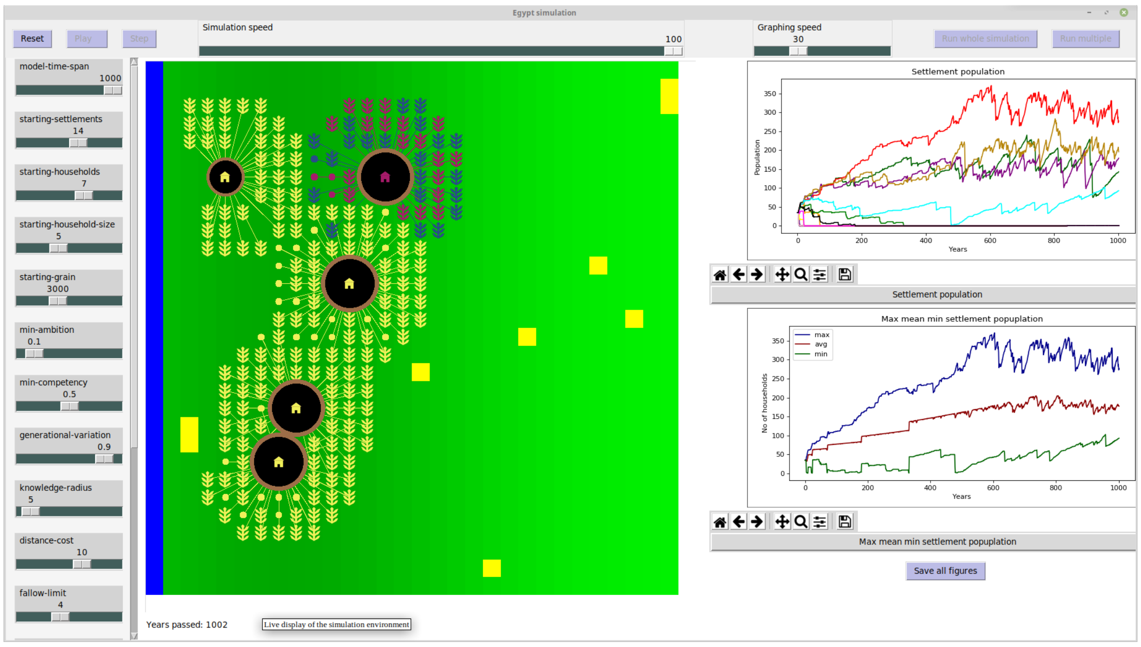
fallow-limit 4

Figure 3: Screenshot of the application once opened, without any actions being performed.



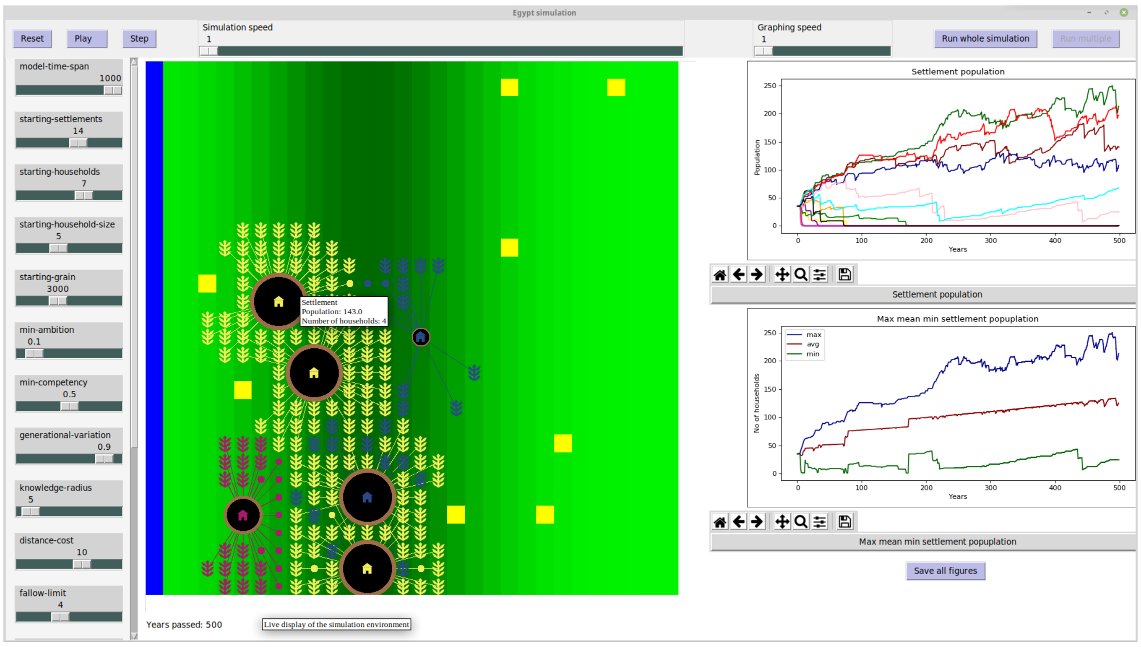
Simulation speed 1
Graphing speed 1
model-time-span 1000
starting-settlements 14
starting-households 7
starting-household-size 5
starting-grain 3000
min-ambition 0.1
min-competency 0.5
generational-variation 0.9
knowledge-radius 5
distance-cost 10
fallow-limit 4
Years passed: 249
Live display of the simulation environment

Figure 4: Screenshot of the application once a run has begun, taken at a slow simulation speed to prevent screentearing during a screenshot.



Simulation speed 100
Graphing speed 30
model-time-span 1000
starting-settlements 14
starting-households 7
starting-household-size 5
starting-grain 3000
min-ambition 0.1
min-competency 0.5
generational-variation 0.9
knowledge-radius 5
distance-cost 10
fallow-limit 4
Years passed: 1002
Live display of the simulation environment

Figure 5: Screenshot of the application once a run has completed.

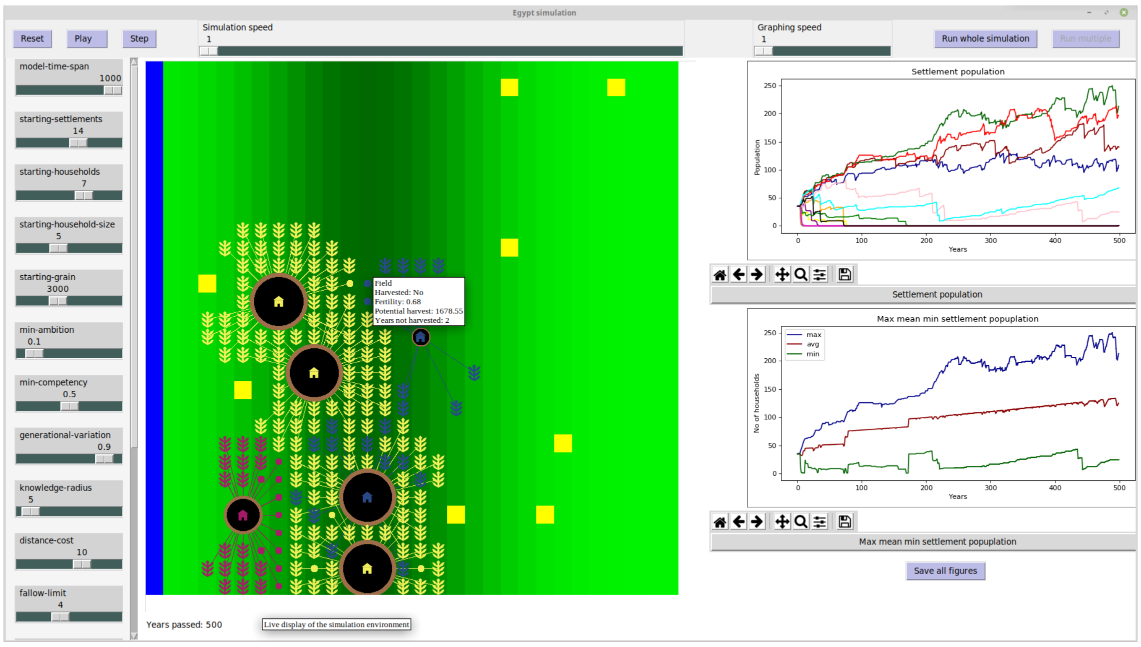


Simulation speed 1
Graphing speed 1
model-time-span 1000
starting-settlements 14
starting-households 7
starting-household-size 5
starting-grain 3000
min-ambition 0.1
min-competency 0.5
generational-variation 0.9
knowledge-radius 5
distance-cost 10
fallow-limit 4

Settlement
Population: 143.0
Number of households: 4

Years passed: 500

Figure 6: Screenshot of the application during a run, with the simulation paused and the output of right clicking on a settlement shown.

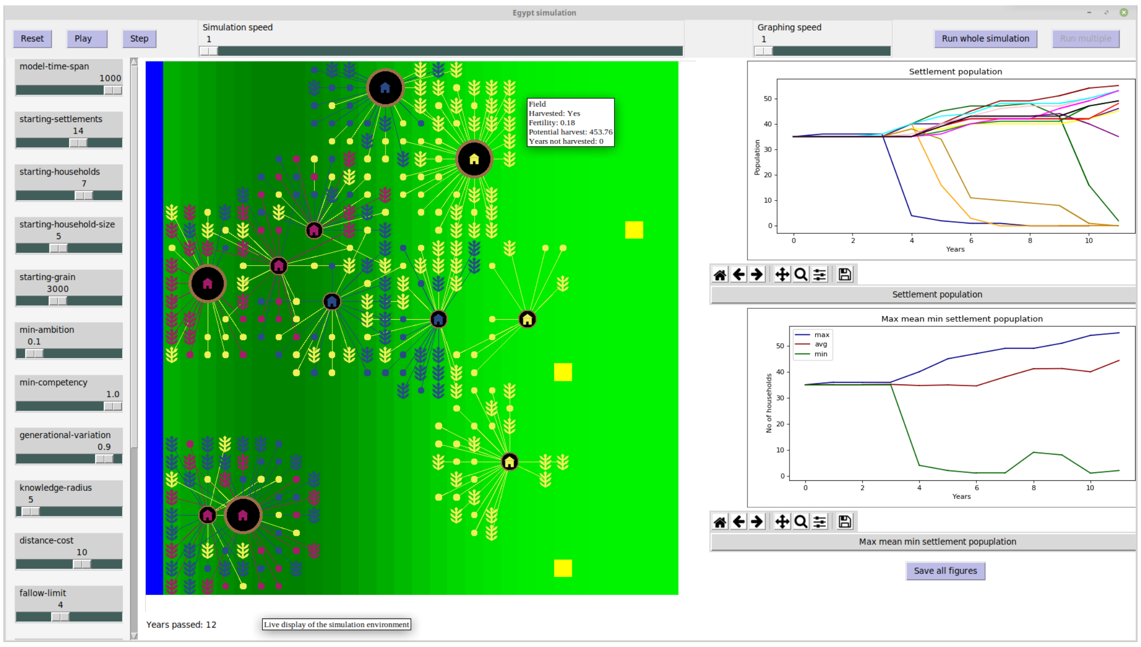


Simulation speed 1
Graphing speed 1
model-time-span 1000
starting-settlements 14
starting-households 7
starting-household-size 5
starting-grain 3000
min-ambition 0.1
min-competency 0.5
generational-variation 0.9
knowledge-radius 5
distance-cost 10
fallow-limit 4

Field
Harvested: No
Fertility: 0.68
Potential harvest: 1678.55
Years not harvested: 2

Years passed: 500

Figure 7: Screenshot of the application during a run, with the simulation paused and the output of right clicking on an unharvested field shown.



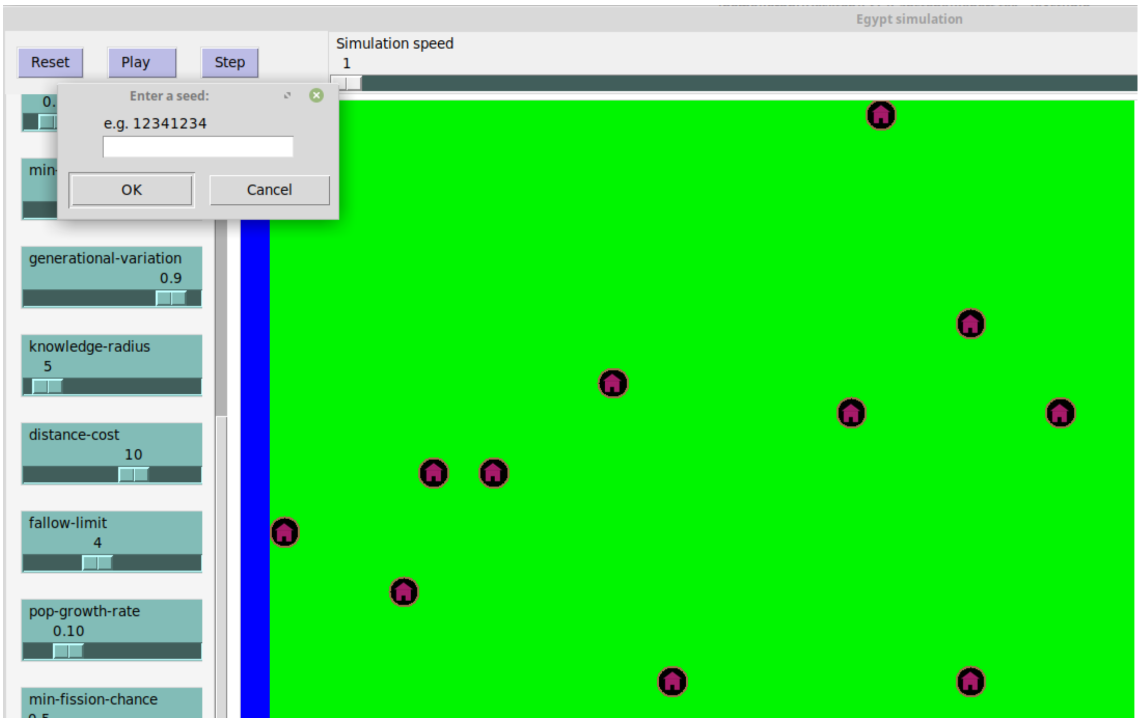
model-time-span 1000
starting-settlements 14
starting-households 7
starting-household-size 5
starting-grain 3000
min-ambition 0.1
min-competency 1.0
generational-variation 0.9
knowledge-radius 5
distance-cost 10
fallow-limit 4

Simulation speed 1
Graphing speed 1

Field
Harvested: Yes
Fertility: 0.18
Potential harvest: 453.76
Years not harvested: 0

Years passed: 12

Figure 8: Screenshot of the application during a run, with the simulation paused and the output of right clicking on a harvested field shown.



Simulation speed 1

Enter a seed:
e.g. 12341234

generational-variation 0.9
knowledge-radius 5
distance-cost 10
fallow-limit 4
pop-growth-rate 0.10
min-fission-chance

Figure 9: Screenshot of the application showing how a manual seed is entered.

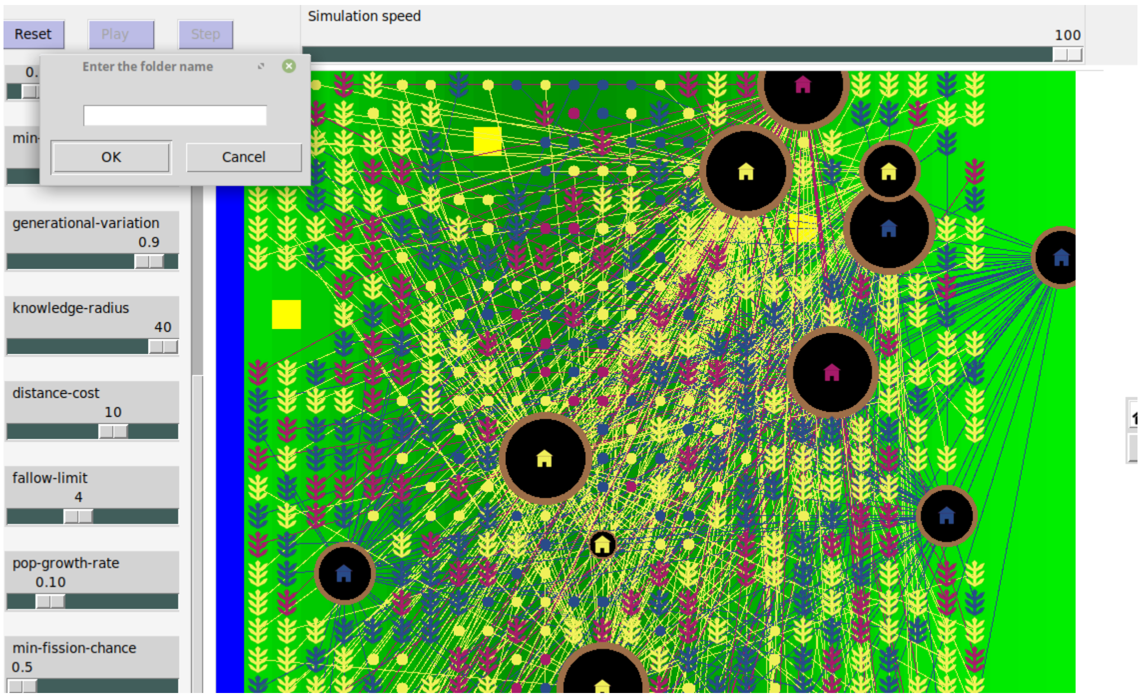


Figure 10: Screenshot of the application showing the dialog asking for a foldername to save multiple figures in for the "Save all Figures" button.

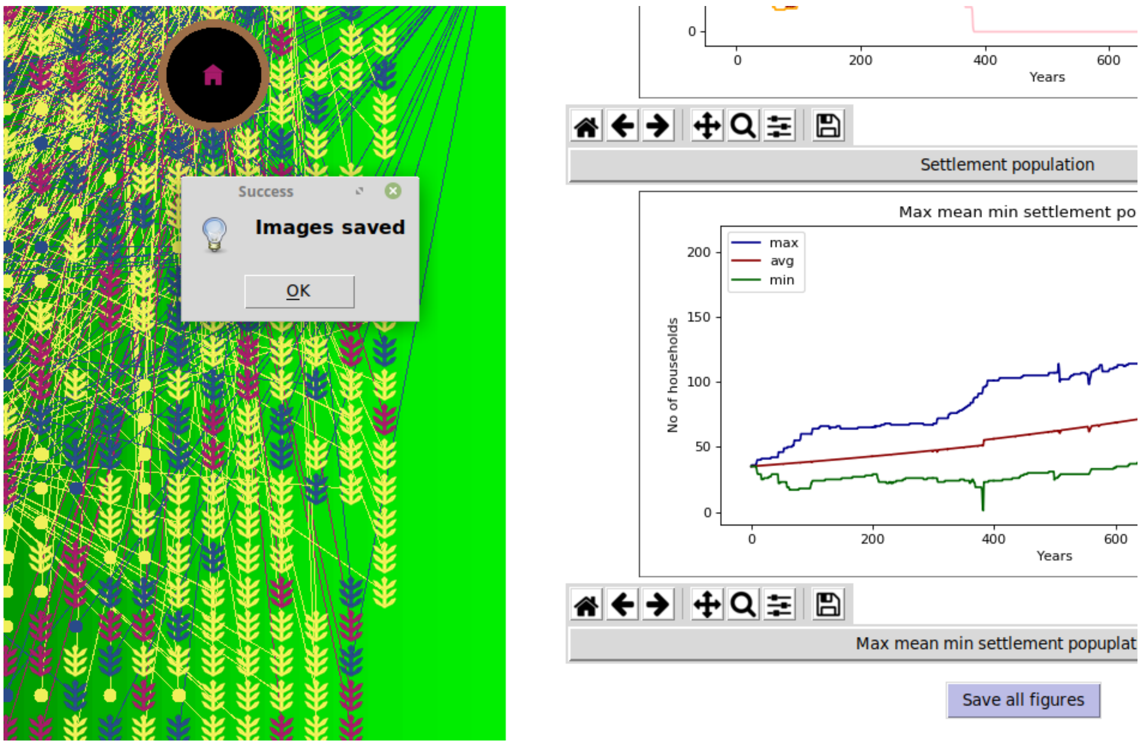


Figure 11: Screenshot of the application showing output of the "Save all Figures" button once all figures have been saved.