CSC3003S Capstone Project: EcoViz

James Burness
Department of Computer Science
BRNJAM019@myuct.ac.za

Jacq van Jaarsveld
Department of Computer Science
VJRJAC003@myuct.ac.za

Owen Franke
Department of Computer Science
FRNOWE001@myuct.ac.za

Computational biomodeling is a field in science that is concerned with building computational models that further the investigative perspective of biological systems. These models incorporate various visualization tools allowing botanists, biologists and other field professionals to explore simulated ecosystems, perform statistical analysis and observe the effects of different environmental conditions, in order to make predictions about the future and act accordingly. We have developed an application: "EcoViz"; a biomodeling application that serves as a visualisation tool to gather statistical information upon a subject botanical ecosystem. The Java-based application is capable of receiving input data representing a real-world biome and generating an interactive visual model. In an overview, the application enables the user to easily navigate and filter a 2D environmental model, consisting of a grayscale elevation map overlaid with colour-coded plant life, gather plant information and simulate the impact of fires and potential firebreak locations. The program incorporates an interactive graphical user interface, allowing for real-time dynamic manipulation of the environmental model and visual analysis. Built using data gathered from various biomes in Sonoma Country, where wildfires are a common occurrence, our fire simulations clearly show the devastating effect that such fires would have upon these areas.

1. Introduction

This project was undertaken at the request of a group of botanists from Sonoma County, USA, who required a biome visualization tool in order to analyse ecosystems and to simulate the effects of climate change and wildfires, which are a common occurrence in the area. The benefit of this program lies with its ability to allow for in-silico experiments and simulations on the model, without having to harm real-world ecosystems in any way, or having to wait for statistical data real fires.

Our biomodelling application, henceforth referred to as EcoViz in this report, was designed to facilitate this desired biome visualisation, by accepting formatted plant, species and terrain data gathered from any real-world ecosystem and constructing a visual model of this data, presented to the user through an interactive graphical user interface. The main component of this interface is a 2D top-down view of the mapped area, showing terrain elevation, plant size and species spread.

Our application is capable of simulating the spread of wildfires, starting at any point on the given model, taking wind speed and direction into account and indicating the resulting damage caused. Additionally, it provides a means to assess the mitigating effect of different possible firebreak locations.

Besides fire control, the application can be used for extracting useful botanical information, such as species elevation distribution, plant clustering or the effects of alien/invasive vegetation. The colour coding of different species as well as various filtering options allow a user to gain a greater understanding of the biome in question. The model can be viewed with differing granularity, initially providing an overview, but allowing a user to zoom in and observe specific isolated areas.

The development of this project consisted of a fairly standard evolutionary approach. The initial stages involved thorough planning and consultation with our primary stakeholder, Professor. James Gain, who acted as the representative of the aforementioned group of botanists. We began with an analysis of the project scope, noting any specific requirements and possible risks, before designing an early prototype, documented with use case narratives and a class diagram. Regular meetings with our stakeholders

allowed us to receive feedback on progress and adapt to any changing requirements. Our development process consisted of a fairly horizontal approach at first, initially implementing many features at surface level, before focusing on each in turn, evolving the prototype into the final product.

2. Requirements Captured

2.1. Requirements

Functional requirements the application achieves

- Support for multiple species types: The application can support multiple species of flora in the same simulation, each represented by a distinct colour. As plants may overlap, the plant renderings are semi-transparent, thus allowing low-growing undergrowth species to be viewed at the same time as tall canopy species.
- Minimap/position guide for perspective: The application includes a mini-map, which provides the necessary perspective when the user uses the zooming feature. The minimap provides context as to how the current view of the model fits into the entire environment as a whole.
- Details on demand: The user is able to click on/select a specific plant, with the program displaying the individual plant's details and highlighting it. The colour of the highlight can be customised by the user. The user can view all plants of the selected species if desired, as well as a real-world image example.
- Zooming capabilities: The user is able to zoom into and isolate any section of the ecosystem, to get a closer look at a specific section of interest, using the mouse scroll wheel. The simulation constantly re-renders, providing high quality visuals of the plants and terrain at any zoom level.
- Dragging/Traversal capabilities: At a zoomed-in perspective the user can traverse/pan across the landscape, by dragging their mouse, allowing them to view different areas without having to zoom out.
- Fire impact simulation: The user is able to simulate fire-spread, from a seed position, taking wind speed and direction into account. The affected trees are highlighted with a red colour scheme. The fire-spread simulation mimics the spread of a real wildfire.
- Firebreaks: The user can remove all vegetation from a desired area of the model in order to simulate a firebreak, preventing fire spread to that area during a simulation.
- Filtering: The user is able to filter the view by removing plants outside of a certain radius surrounding a selected individual, outside of a certain height/canopy-radius range, removing an entire plant layer or entire species/group.

Non-functional requirements the application achieves

- The fire simulation runs at an average frequency at least 10hz.
- The application can support up to 25 species in a single model.
- The zoom feature operates with a response time delay of less than 25 ms.

Usability requirements the application achieves

- A User friendly GUI: Every possible user action is intuitive and the user should never be confused as to how to use the application.
- User manual: A manual is provided which allows a brand new user to navigate your application successfully.
- Stable and smooth performance: The zooming, dragging/scrolling, fire simulation, and any other visual manipulations of the view are smooth, and do not stutter or freeze.
- Immediate response/ Dynamic queries: All allowed user actions produce immediate responses in the model, where possible.

2.2. Use cases

As can be seen from Figure 1, our application contains a large number of features. For this reason, our use case narrative section was rather lengthy, and so we have included it as Appendix B.

2.3. Major Analysis Artefacts

Our two main analysis artefacts are the design class diagram and architecture diagram, both shown below in 2.3. The class diagram details the entire class structure/ hierarchy and all relationships between them, displaying the main methods and variables for each class, which are further discussed in Section 4.3. The

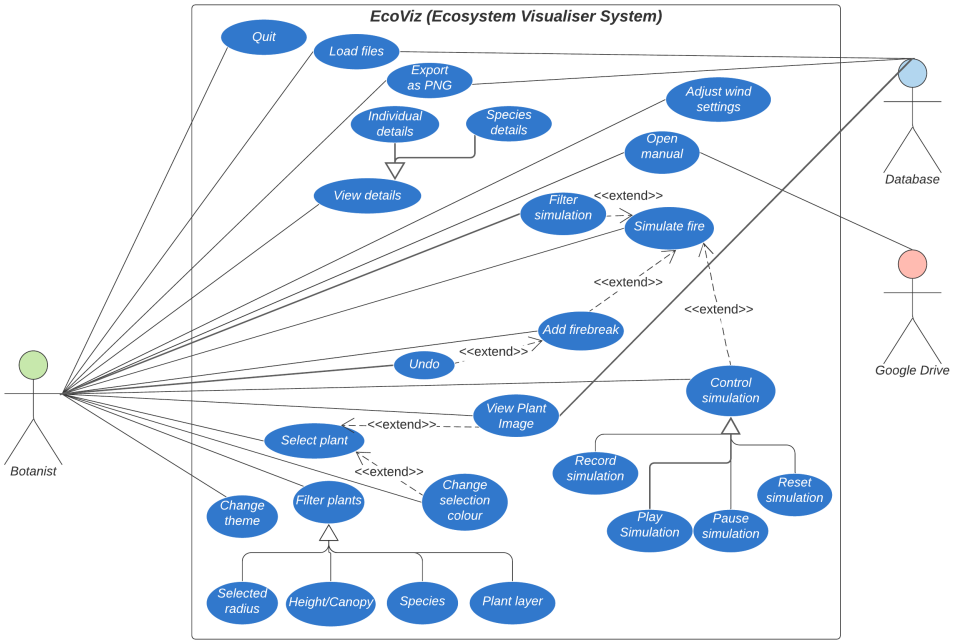

Figure 1: Use Case Diagram

architecture diagram, which follows this, details our system from a higher level, showing how we used the MVC architecture, which classes fit into each component/layer of our architecture and how the three components interact.

3. Design Overview

In order for our application to provide a user-friendly interactive experience, we needed a custom designed, robust user-interface. For this reason, we separated the view presented to the user into three classes, GUI, ImagePanel and MiniMap. The Gui class is responsible for all graphical elements, such as buttons and colouring, aside from the main dedicated panel which displays the 2D model of the biome. As this visualisation is the most important aspect of our system, it is managed by a dedicated class, ImagePanel, handling all image painting, and colour overlays, in order to produce high quality visuals. The MiniMap class displays a smaller image, scaling down the larger biome visual and providing a position guide to the user.

Another important aspect of our user interface was dynamic user interaction, and direct manipulation of the visualized environment. For this reason, we had a dedicated Controller class, responsible for creating and handling action listeners for every interactive Gui component. Each action has a corresponding internal method, which performs some manipulation on one or more of the Model (data storage) classes. The controller cannot be directly accessed by the user enabling our system to safely and efficiently manipulate and change the data model and provide immediate user feedback on the Gui. We chose to have add a FileController class to manage all file I/O processing and validation, in order to separate functionality and allow for a cleaner, more dedicated Controller class.

Finally, the data required as input to this application takes the form of four separate files, each including a variety of multivariate data describing the biome under study. With such data, we thought it best to categorize the data into separate classes, such as species, terrain, plant etc., allowing each class to store related pieces of data, and have a focused purpose, rather than trying to store all data in a single database/class.

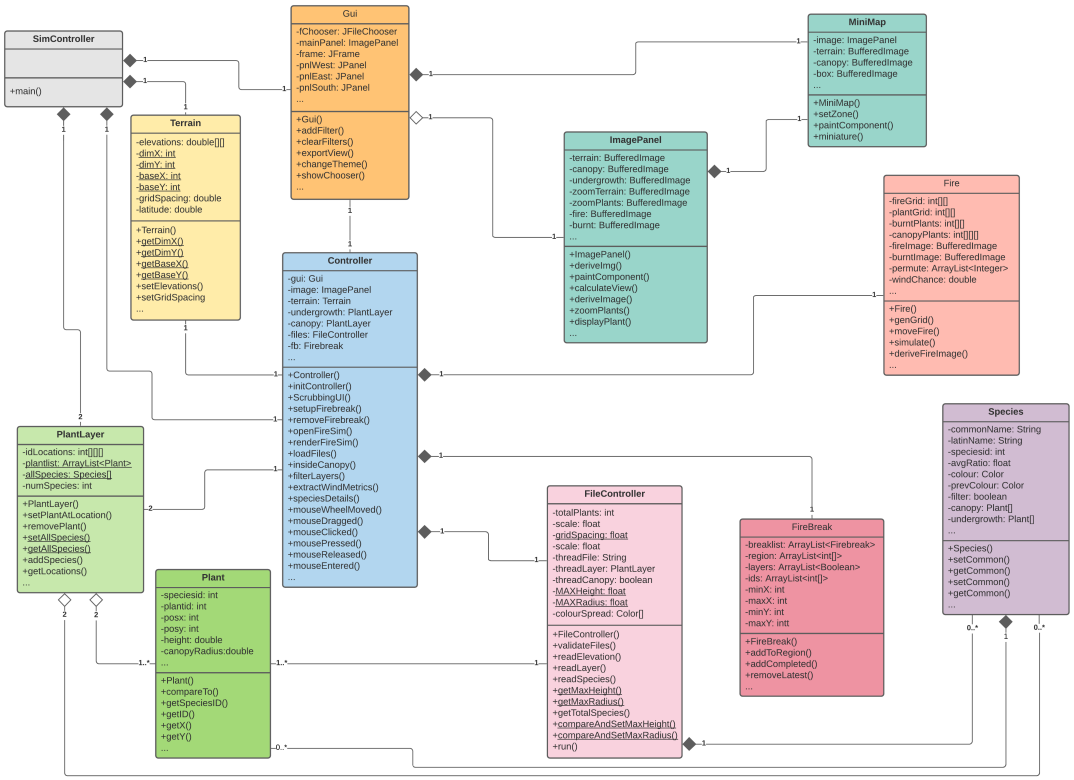

Figure 2: Design Class Diagram

3.1. Architecture Used

The EcoViz application uses the Model-View-Controller architecture pattern. This separates our system class hierarchy into 3 main components: Model, Controller and View. As the main requirement for this project was to produce a view/visualization from an underlying dataset, with interactive user elements manipulating and changing the view, based on changes in the data model, the MVC is a perfect fit. In the case of our simulation application, all user actions and input was handled by our controller, all Graphical User Interface elements were created and maintained by our View classes, and all classes concerned with the storage and trivial manipulation of the input biome data, formed the Model. The manipulations and experiments performed by users of our application are not supposed to change the actual gathered field data, and so a model in which the user interface is abstracted and separated from direct interaction with the data, such as the MVC pattern, was ideal. Below we describe each component of the MVC and how our system has been fitted to this architecture.

- *Model*
  The model represents the objects/classes whose purpose it is to store data, to be displayed in the view. The model objects in our application are as follows: “Fire.java”, “Firebreak.java”, “Plant.java” , “PlantLayer.java”, “Species.java” and “Terrain.java”. The method in which data is organised between these classes is discussed below.
- *View*
  The view includes all classes/objects responsible for the display and visual representation of our data model. In our case, this forms our program’s graphical user interface, one of its most important features. The classes/sections of our system responsible for the view are: “Gui.java”, “ImagePanel.java” and “MiniMap.java”.
- *Controller*
  The controller component represents the object(s) whose purpose it is to recognise user input/ interaction with the View, and to translate these actions into manipulations of the data Model, which are then

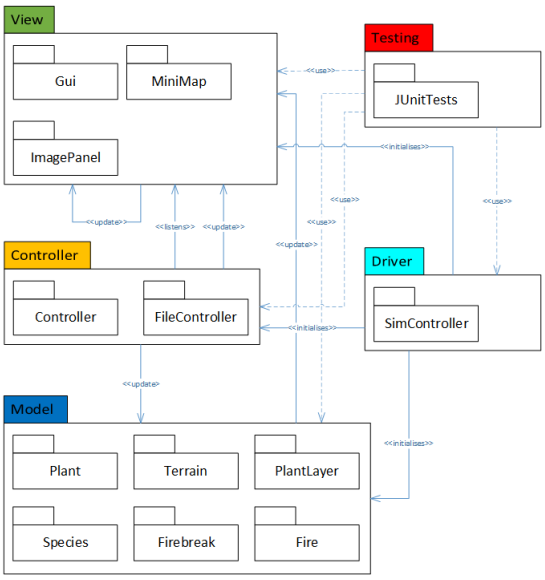

Figure 3: Package Diagram

reflected back to the user by the View. In our system, the controller consists of: "FileController.java" and "Controller.java".

Two classes do not explicitly fit the MVC model. The first is the SimController class; our driver file, which creates the MVC components individually, before assigning control to the Controller class. The second, is our JUnitTests class, also a driver file, but does not run our application, instead running a number of defined unit tests. A full explanation of the tests included in this file is described in Appendix C.

3.2. Integral Algorithms Used

- *Circle Drawing Algorithm*
  This algorithm was implemented due to a severe delay caused by using the fillOval method provided with Java's Graphics2D library. Our algorithm generates all pixel coordinates (x,y) that would form part of a circle, given a center 'x','y' and radius 'rad', by using euclidean distance and checking a square box around the center pixel (dimensions rad+1 x rad+1). In the code shown below, it was used to populate a 2D grid with 1s at all positions inside the circle. This algorithm was also used to draw the coloured circles representing each plant in our 2D visual environment.

```
// in Fire.java
int specId = underPlants[y][x][0]; // Species ID
int plantID = underPlants[y][x][1]; // Plant ID
Plant[] uPlants = specieslist[specId].getCanopyPlants();
double rad = uPlants[plantID].getCanopy();
double temp = Math.round(rad);
int boundary = (int) temp + 1;

for (int j = y - boundary; j < (y + boundary + 1); j++) {
   for (int i = x - boundary; i < (x + boundary + 1); i++) {
      if (j < Terrain.getDimY() && j > 0 && i < Terrain.getDimX() && i >
          0) {
         double dist = Math.sqrt(Math.pow((x - i), 2) + Math.pow((y - j),
             2));
         if (dist <= rad) {
            plantGrid[i][j] = 1;
         }
```

```
        }
    }
}
```

- *Inside Canopy*
  This algorithm takes an (x,y) point, and a Plant object as arguments, and makes use of the basic circle formula to determine if the point falls inside the plant's radius. This algorithm was useful for determining if a user clicked on a specific individual plant, in order to display details on demand.

```
// in Controller.java
   private boolean insideCanopy(Point point, Plant plant) {
      int x = (int) Math.round(point.x / image.getZoomMult() +
         image.getTLX());
      int y = (int) Math.round(point.y / image.getZoomMult() +
         image.getTLY());
      return (Math.pow((x - plant.getX()), 2) + Math.pow((y -
         plant.getY()), 2)) <= Math.pow(plant.getCanopy(), 2);
   }
```

- *Zoom (Re-Rendering)* When zooming in and out of the model, the visualised plant data, represented as coloured circles on the main panel, needed to be re-rendered, to ensure a consistent high quality visual experience. This involves calculating a new view (either wider or narrower than the previous) and calculating the new positions and scaled dimensions of each plant that falls within this view. To perform this task, the system tracks the top-left corner coordinates (used as x/y offsets) of the view for every change in zoom, as well as the scaled dimensions, based on a zoom multiplier value.
- *Drag/Pan (Re-Rendering)*
  When panning across the terrain at some zoom level, the distance that the dragged mouse cursor has moved from its starting point is constantly recorded and updated as an x/yDiff. These values are added to the offset in order to determine the new translated view, changing the position of the top left corner of the view window in the original full model.
- *Plant Inside Rectangle*
  This algorithm is used by the ImagePanel class to determine whether any part of a plant's canopy falls within a rectangular view. The rectangle is defined by the coordinates (x,y) of its top left corner, as well as its width and height. The algorithm uses the maximum and minimum x,y coordinates covered by the plant's canopy, checking if these fall within the area covered by the rectangle. This algorithm is used by the ImagePanel to detect which plants to re-render at any particular zoom level.

```
   //Check if plant is in give rectangular view
   public boolean plantInRect(int x, int y, double rad, int tlx, int tly,
       int w, int h){
      if((x + rad > tlx) && (x - rad < tlx+w)){
         if((y + rad > tly) && (y - rad < tly+h)){
            return true;
         }
      }
      return false;
   }
```

- *Collections.sort - merge sort*
  All plants recorded in the dataset are stored in one large ArrayList. This ArrayList is sorted based on plant height, in ascending order, to impose correct ordering for mixing of plant colours. To sort this ArrayList, the sort() method available in the Java Collections library is used. This function uses a modified MergeSort algorithm. While slower than quicksort, this algorithm provides a stability

and robustness that quicksort does not. The delay, as shown in Section 5.1 is minimal and a custom algorithm was deemed unnecessary.

```
Collections.sort(PlantLayer.getPlantList());
```

3.3. Data Organisation Used

Here we will discuss the basic organisation techniques used to separate the multivariate data generated by botanists surveying these different biomes, into our different model classes.

- *Plant and Species Data Storage*
  Each plant in the environment is represented by a unique Plant object. These objects store plant-specific values, such as dimensions, location etc, as well as two identifiers, a plantID and speciesID.
  Each species is represented by a single unique Species object. These objects have fields for the common name, latin name etc. Each Species object is identified by a unique speciesID and contains two different plant lists; one for each plant layer.
- *PlantLayer Data Storage*
  Each .pdb file processed represents one two plant layers; the undergrowth or canopy. Details of each layer are stored in a unique PlantLayer object. These details include the number of plants in the layer, as well as a 2D grid of plant locations. The PlantLayer class stores a static ArrayList of Plant objects, which contains every plant found in ALL layers, and a static species list, containing all species present across both layers.
- *Terrain Data*
  Our terrain class stores a 2-dimensional list of elevation values, in order to populate the grayscale terrain image, as well as the basic width and height dimensions of the geographical area being mapped.
- *Fire and Firebreak Data*
  The Fire class's main purpose in terms of data organisation is to populate and store a fire grid and plant grid, in order to facilitate the fire spread algorithms. These grids store the locations of all instances of fire generated in a fire simulation. Each firebreak, generated by the user painting/dragging their mouse over the model (when the system is in fire simulation mode) is stored in a static ArrayList managed by the Firebreak class. Each one of these are stored as a Firebreak object, which stores the region of the model covered by a user-created fire break, including any plants that have been removed.

4. Implementation

4.1. Data Structures used

Our application's data source takes the form of four formatted data files. Our main data structures are concerned with storing this data in a meaningful and structured way, to allow easy manipulation and graphical visualisation of the Model.

- *PlantLayer static field - plantlist:*
  This plantlist stores every Plant object created during the initial processing of the data set, across all species and layers. No count of the total number of plants in the given environment is provided and so the dimension of this list cannot be predetermined. Thus, an ArrayList of Plant objects was chosen for this data structure, allowing dynamic resizing and allowing new Plant objects to be appended as they are created. In order to reduce the processing delay of large plant database files, we made use of threading to read both of the .pdb files concurrently. To ensure that both threads had access to this shared ArrayList data structure, we made it a static member of the PlantLayer class. This ArrayList, once complete, is sorted in ascending order according to PlantHeight. For this reason, data structures such as a binary tree were considered, but ruled out due to the powerful random access and indexing that ArrayLists provide.
- *PlantLayer static field - allSpecies:*
  The allSpecies array stores details of all the different species found within the ecosystem, up to a maximum of 25 species (non-functional requirement). Thus, the total number of species can be easily determined by a quick pass of the .spc file. Hence, a standard Species object array is used, as the length

of the list required to store all the Species objects is determined before instantiation. However, as the .pdb files are read in parallel, and these files contain additional species data, the two file-reading threads both still require access to the same allSpecies array, hence it is also static. The chronological order imposed on the data file format results in this list being implicitly ordered by ascending speciesID.

- *PlantLayer field - idLocations:*
  The idLocations structure is a 3D array. Its purpose is to store a pair of unique integer ID's identifying each Plant entity i.e the speciesID and plantID, at the specific x,y coordinate that the plant can be found in the layer. Thus the first two dimensions are determined by the width and height of the geographical area being studied, while the 3rd dimension is always a 2 element integer array, containing first the speciesID and plantID.
- *Species fields - canopy & undergrowth:*
  Both of these structures are easily implemented as simple Plant object arrays, as each is populated from a separate database file, which specifies the number of plants of each species found in that layer.
- *Terrain field - elevations*
  A 2D array of floating point values - each dimension of the array corresponds to a dimension of the landscape indicated in the header lines of the elevation datafile.
- *Fire fields - fireGrid & plantGrid*
  These arrays represent the same 2D grid landscape mentioned extensively above, and so required no extra explanation.
- *Firebreak static field - breaklist*
  This list contains all of the fully completed Firebreak objects created by the user. As there is no limit to the number of firebreaks that can be drawn, an ArrayList was required, allowing the size to be increased according to the user's wishes.
- *Firebreak fields - layers & ids*
  These two structures are related, as they together identify each Plant object removed by a given firebreak. Thus ArrayLists were once again used, in order to allow for firebreaks of varying sizes.

4.2. Graphical User Interface

- *The decision to use Java Swing instead of JavaFX:*
  The application makes use of the Java Swing and Java AWT libraries. JavaFX is a more modern library that may have UI components with a better look and feel, but still falls short to the quantity and polish of Swing components in present time. JavaFx is far more suitable for 3D development than Java Swing and AWT, however this does not fit into the scope of our project as our visualisation only required 2D graphical development. To overcome the aesthetic limitations of the Java Swing look and feel, we simply made use of an open source library 'FlatLaf' which is discussed in Section 4.5.
- *The use of Java AWT*
  Java AWT provides a key role in our applications visualisations. It is an application programming interface that allows us to paint graphics and images to Java Swing containers.
- *Initialisation Frame and JFileChooser:*
  The start of the application shows a JFrame that presents an aesthetic welcome screen, which requests the user to load in files. When the user does so it prompts a JFileChooser, which allows the user to visually navigate through their personal directories and load in the required files.
- *East panel:*
  The east side panel of the application is a JPanel container, used to display a number related components for a number of different functions. It contains a JTabbedPane, consisting of four panes: Details, Config, Filters and Species. This helps to hide content shown on screen to the user, which helps to apply a minimalist approach. The panel also contains another JPanel, used to display the fire simulation controls. Lastly it contains the mini map.
- *ImagePanel (Class):*
  The ImagePanel class extends JPanel, a generic lightweight container. The class overrides the paintComponent method, which takes in a Graphics parameter and can be used to draw Java BufferedImage objects onto the JPanel container. This class was used to visualise fire movement by repainting the

panel rapidly to create the illusion of movement using still images. This is the phi phenomenon. The panel also controls the displaying of zoomed perspectives as well as filtered data.

4.3. Significant class methods

- *Sim Controller:*
  The SimController class only contains the 'main' method, or program entry method of our application. This method creates a new Gui and Terrain object, two PlantLayer objects (canopy and undergrowth) and a Controller object, before passing control of the system to this controller instance.
- *Controller:*
  initController() - The initController method is responsible for creating and adding action, change and mouse listeners to all interactive Gui components, and to the ImagePanel itself. This method then calls the initView() method allowing the user to begin interacting with the application.

  loadFiles() - This method retrieves the files selected by the user through the JFileChooser, and passes these filenames to the FileController class for validation and processing. The method performs a simple initial check to determine whether there are the correct number of files (4). The method then invokes the 'read' methods of the FileController objects, managing the parallel 'multithreaded' reading of the two plant database files, before calling the refreshView() method. This method is called whenever new files are loaded in.

  refreshView() - The purpose of this method is to reset all Gui components to their default values, and then refresh the Gui, to display a new model based on newly processed data files. This method is called by the loadFiles() method described above.

  resetAllFilters() - This method is called when the user enters the 'fire simulation mode'. The ecosystem visual should be set back to the overview (100%) zoom and all filter components should be set back to their initial default values and disabled, displaying all vegetation to the screen.

  mouseWheelMoved(), MouseDragged(), MouseClicked() - These three methods are overridden due to the Controller class implementing the various Java Mouse Listener interfaces. The mouse wheel listener facilitates the zoom functionality, the mouse clicked listener is used for Details on Demand and the mouse dragged listener is used to calculate distance the mouse has been dragged across the landscape, in order to pan the view.
- *FileController:*
  readElevation(), readLayer() and readSpecies() - These three read methods are used to receive input data from the user in a specified data file format. The data is parsed and stored in their respective objects for use by the rest of the program.
- *Gui:*
  All of the important dedicated Gui class components are created in the default constructor, including an ImagePanel object and MiniMap object.
- *ImagePanel:*
  deriveImg() - This method is used to create the initial grayscale BufferedImage representing the terrain elevation values of the landscape, using normalized height values to create a scale to shade pixels in between white (high elevation) and black (low).

  zoomPlants() - This method iterates through the ArrayList of all Plants, checking them for a variety of filter values, which must all pass in order for them to be displayed to ImagePanel. The plant location is then checked to determine whether it should be visible within the current view, given the current zoom level and view offset. Finally, if the plant is to be visualised, the x,y coordinates of the plant are scaled accordingly and the radius enlarged or decreased depending on the direction of zoom. This method performs the functionality required to re-render the plant circles, maintaining high quality visuals at any zoom.

  drawFirebreak() - This method is invoked when a user drags their cursor across the screen while in 'firebreak mode'. The basic purpose of the method is to display the area the firebreak will cover, by mimicking a red circular brush around the user's cursor. In reality, the method uses a circle drawing algorithm to repeatedly draw red circles at the current mouse position. All pixel coordinates painted red by this method are then added to the new Firebreak object being created, which is passed in as a

method parameter.

- *MiniMap:*
  setZone() - This is a mutator method, used to set the new position (determined by the coordinates of the top left corner) and dimensions of the position guide box, which provides context to the user about how their current view fits into the overall environment. This method then causes a minimap repaint, causing the updated red-box position guide to be displayed to the user.
  miniature() - This method is used to scale down the BufferedImages collected from the companion ImagePanel class, to allow them to be displayed on a 200x200 panel. The method makes use of the Java Graphics2D library, as well as the Image.getScaledInstance() method in order to uniformly compress the images, whilst retaining the key visuals.
- *Fire:*
  moveFire() - This method is used to determine the path of fire, taking into account a random chance of moving on the terrain regardless of vegetation data. The method also applies the effects of wind.
  genGrid() - For fire simulation, the locations of plants on the terrain is imperative. This method uses euclidean geometry and the radius of each tree in the database to fill cells in a 2D grid that represent pixels that contain trees on the visualised model.
  simulate() - Simulate controls a randomised permutation of the flow of fire. It checks whether fire is at a specific location and calls the moveFire() method if there is.
- *Firebreak:*
  addToRegion() - This method adds a specific coordinate (x,y) pair to the region ArrayList, increasing the area covered by the firebreak to be used to determine which plants should be removed.
  inFirebreak() - Given a Plant object, this method determines whether or not the center of the plant falls within the region of the calling Firebreak object, by comparing the x,y position of the plant base, to each x,y coordinate in the Firebreak region.
- *Plant:*
  compareTo() - The Plant class implements the comparable interface, and thus overrides this method, allowing Plant objects to be compared by height and sorted.
- *Species, PlantLayer, Terrain:*
  All methods of the Species,PlantLayer and Terrain classes are simple accessor and mutator methods used to acquire and manipulate the data.
- *JUnitTests:*
  The methods of the JUnitTest class are discussed extensively in 5.1

4.4. Special class relationships

Our SimController class acted as a driver file, instantiating Gui, Terrain, PlantLayer and Controller objects, before passing control to the Controller class. The Controller is the only class that has access to both the Model and View, and can manipulate and call methods in both.

4.5. Additional Techniques and Libraries

- *Threading:*
  Java supports parallel programming through the Runnable interface and Java Thread library, both of which were used in our system. The FileController class implemented the Runnable interface in order to allow both plant database files to be read concurrently, in two separate Threads. We also made use of TimerTasks, which are abstract classes implementing the runnable interface, for our fire simulation control.
- *FlatLaf:*
  FlatLaf is a free open-source “Look and Feel” library for use in Java Swing applications. This gave our application a modernised graphics overhaul for a better aesthetic.
- *JUnitTests - junit library*
  The org.junit library is used to provide a convenient unit testing framework. This allowed us to test the functionality and accuracy of our classes, using methods such as ‘assertEquals’.

5. Program Validation and Verification

Table 1: Summary of Testing Plan

Process	Technique
1. Unit Testing: consisted of testing methods and state behaviour of the applications classes.	White-Box Unit Testing was implemented through the use of the Java unit testing framework. The tests ensured the accuracy of the internal structures present in the classes.
2. Integration and Benchmark Testing: testing the performance of main data structures and algorithms of classes.	Comparing benchmark tests between past and present implementations, a speedup could be calculated.
3. Validation Testing: test whether customer requirements are satisfied.	Client meetings were conducted as user acceptance test scenarios to showcase progress of the system and receive input.

5.1. Description of testing:

As noted in our above Software Testing Summary, we performed a large number of Unit Tests. Due to the verbose nature of these descriptions, they have been added as Appendix C. The remainder of this section is concerned our describing some of our benchmark tests used for determining the improvement offered by newly integrated algorithms/features as well as our acceptance and random behavioural tests. For cases such as response time, in which we were aiming to meet a maximum limit set by non-functional requirements, worst case testing using the large 1024x1024 dataset was the best option, whereas for certain other tests, averaging results across all datasets proved more useful.

One of our functional requirements specifies that the zoom feature should operate smoothly and respond with minimal delay. We conducted zoom response time tests whenever a new implementation of the zooming algorithm was added. The results of the most recent tests are shown in Appendix D? In summary, the worst case scenario zoom response time was achieved zooming in and out at the lowest level of zoom (furthest perspective). The older implementation produced an average (over several tests) worst case response time of 2414 ms. This delay was due to this method using the Java Graphics library's fillOval method. Due to this sluggish response, and feedback during a user acceptance test performed during a stakeholder demo meeting, a custom circle drawing algorithm was created, which appears in the final implementation. This algorithm produced a worst case average response time of 116ms. This is a 20.81x speedup.

Our second test measures the speed of the initial visualisation generation, on a 'non warmed-up' system. This is in-effect the slowest possible render. By comparison of these start-up display times, we could directly measure the speedup offered by our new algorithm. For the largest dataset of 729727 plants, the average speedup was 5.5x (1771 ms to 321 ms). The duration of the provided Collections.sort() method was tested on all three datasets, as can be seen in Table 3 in Appendix D.3. The worst case average sort time was 420ms and with this being less than a 0.5s delay in the worst case, we decided that creating a more optimized sorting algorithm would not be worth the time spent.

In addition to such tests conducted using the given data, test data files were created to test details on demand, high quality zooming, species selection and species filtering etc. (any other features not quantitatively measurable). These simple data files described five species, each consisting of five identically sized plants. At one point, the plant database and species files were changed to reflect 25 species each of 1 plant each, to test our choice of colour spread, and ensure that the requirement of our application being able to handle up to 25 species had been achieved.

As mentioned briefly above, user acceptance testing was performed, by demonstrating the progress of our evolutionary prototype to our stakeholder, Prof. James Gain, a number of times throughout the development process. This allowed us to get user feedback, identify key areas which were higher priority

than others, adapt to changing requirements as well as determine which features were acceptable and which ones still required work. Another testing process that was followed, mentioned briefly in our software testing plan outlined above, was random stress testing of other team-members' contributions. Each member of the development team tested the robustness and validity of any new feature added by the other developers. A number of these tests are outlined below.

Table 2: Summary of Random Behavioural and Stress Tests.

Data Set and reason for its choice	**Test Cases**		
	Normal Functioning	*Extreme boundary cases*	*Invalid Data*
File loading (validation)	Passed	Tested with files of different data sets together	Passed
Filtering combinations (validation)	Passed	Tested with input, slider and checkbox filters simultaneously	Passed
Filtering reset on file load (visual/system)	Passed	Tested with combinations of all filters applied	Passed
Filtering plants that are out of view (visual)	Passed	n/a	Passed
Fire simulation resets (system)	Passed	Tested before, during and after simulation, with firebreaks, and with wind	n/a
Simulation recording resets (system)	Passed	Tested before, during and after simulation, with firebreaks, and with wind	n/a
Colour theme changes (visual)	Passed	Tested during fire simulation	n/a
Maximum zoom level (visual)	Passed	n/a	n/a

5.2. Discussion

Correctly validating the application ensures the correct behaviour of the system. Each data set tested fulfills an imperative role in the applications functionality. File loading required proper validation, as it is the backbone of the application with all features stemming of it. The filtering aspects of the application were tested with varying input combinations, thus providing the stability and accuracy of enabling a user to explore the ecosystem in a personalised and efficient manner. All simulation tests resulted in a progressively smooth in-silico experience through each iteration in development. Visual tests were also considered as important tests, due to the importance of providing a seamless navigation experience on the user interface.

Our tests above show we have produced a robust and high quality application, focused on satisfying each and every user requirement. The program has been repeatedly tested and validated by user feedback, and regularly validated for correctness by developers. Our aim was to create a User-friendly graphical user interface-based system for displaying and analysing gathered botanical data, presenting users with an overview, filtering options, details on demand and allowing immediate dynamic manipulation.

As shown in the above testing results, our application fulfils the non-functional requirements detailed by our stakeholder. The system can handle up to the specified maximum of 25 species per biome and on average, when zooming in and out of the environment (not just the worst case initial and final zooms) our program achieves response times of around 20-40ms. The Fire-spread simulation has been optimized and is capable of running at the 10Hz frame rate required, and can be sampled at around 10 frames a second to perform scrubbing and video analysis.

Through a combination of unit testing, random edge case testing by the developers, user feedback and acceptance testing through demonstrations and finally numerical measurements and benchmark testing of any new features added, along with a focus on keeping an up to date bug detection report during the entire development cycle by use of a Kanban board, we have ensured that our project will meet the

expectations of any future user. Additionally, our project is extensible, as we followed a clear MVC architecture structure, with a clear and modular design for each class that forms part of our system.

6. Conclusion

Our client requested a fully functional Java based, ecosystem visualisation tool. Our application, EcoViz fulfills the clients scope requirements and includes a few additional features.

The application needed to follow a relevant architecture design pattern, for which we chose the Model -View-Controller. The design pattern is useful for a fast development process that catered for easy modifications without greatly affecting existing code. Following object-oriented techniques and this design pattern, our application has been developed to be fully modular. This proved vital in our development technique, as our team was able to conveniently work independently with the use of source control.

EcoViz provides a platform for intuitive use with a navigable interface. The application allows for convenient and realistic in-silico wildfire simulations with well implemented fire breaking options that give its users an efficient and intuitive environment for testing different environmental effects and the viability of certain mitigation options, in order to make predictions based off of the statistical analysis possible with this powerful tool.

The program is extendable in use, capable of reading different files for any ecosystem, as long as the input data follows the required formats. Due to the modularity of object-orientation, the application can also be modified to cater towards future requirements that the client might have. Additional functionalities, such as extra filters, and a database of species images, have been added, beyond the original scope provided by our client, to improve upon the existing requirements and leave any future user with a broad and polished software product.

This system holds major importance for the field of computational biomodeling, and shows what computers can make possible in terms of analysing and processing even organic data. A final presentation of EcoViz to our stakeholder Prof. James Gain was conducted, in which he expressed his satisfaction with our final result.

A. User Manual

EcoViz has an intuitive user interface, however it may be the case that a user may have a different thought process to some other, and wish to consult a guide. A link to our user friendly and themed manual can be found here.
However, for a more in-depth insight, we have created a more formal user manual. This user manual can be seen across (Appendix A.1 - A.5)

A.1. Loading in Files

To begin the visualization experience, you first need to load in data files. This can be done by selecting ”Load Files” on the initialization frame at the start of the application. To choose your input files, navigate to your data directory, and perform a <Shift> or <Control> select to highlight the four files. The four file types required are: 1 x .elv 1 x .spc.txt 1 x canopy.pdb 1 x undergrowth.pdb After successfully loading in the files, the application is prepared to visualise your data. Closing the initialisation frame and then showing the main frame (Figure 4).

If you would like to load different files without restarting the application, navigate to the top left corner and <Click> “File”, then <Click> “Load Files”. You can now reload new files using the same technique described above (Figure 5).

A.2. Learning how to navigate the East Panel

The east panel is used for viewing details, applying filters and running simulations. If you wish to view the details pane <Click> on “Details” tab located on the top right of the application. To view “Species” pane <Click> on the “Species” tab located on the top right of the application. To view the “Filter” pane, <Click> on “Filter” tab located on the top right of the application. Lastly, to view the “Config” pane <Click> on the “Config” tab also located on this top right bar (Figure 6).

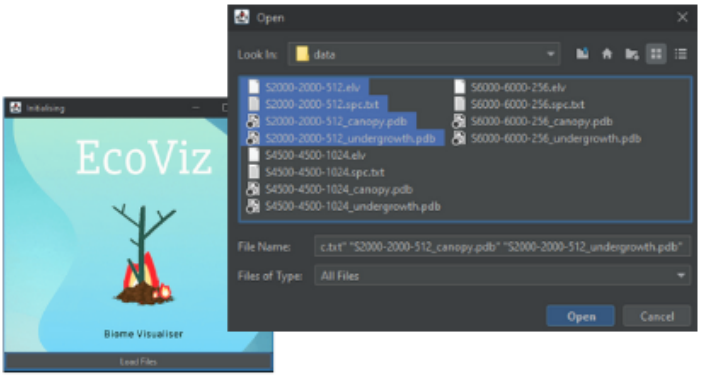

Figure 4: Loading files at initialisation

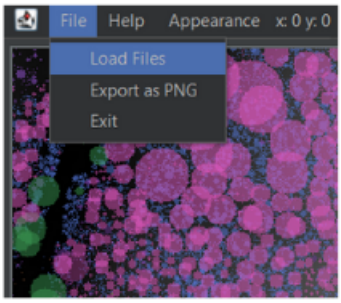

Figure 5: Load in new files

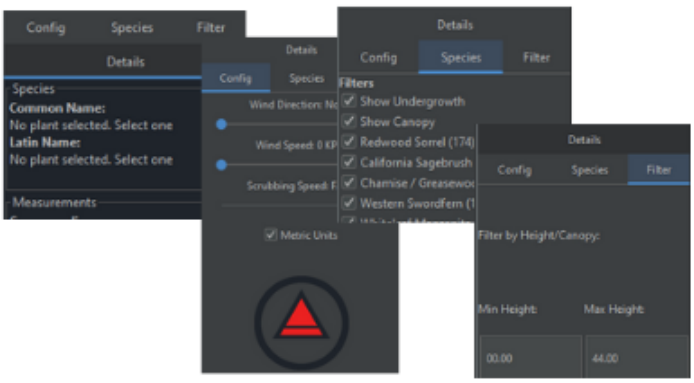

Figure 6: Navigating the east panel

A.3. Viewing Details

- *Zooming and dragging:*
  Use your mouse's scroll wheel to increase/decrease the zoom on the visualised model and use <Click and Drag> to shift the perspective.
- *Plant details:*
  <Click> on a "Color Fill" to set the species colour, when you click on a plant. <Click> on a plant to view its specific details. The details show on the east side panel, in three categories: "Species", "Measurements" and "Stats". <Enable> the "Full Details" to view the details of all of the selected species. You will now be able to view the image of the species as well. <Click> the image to see an enlarged version.

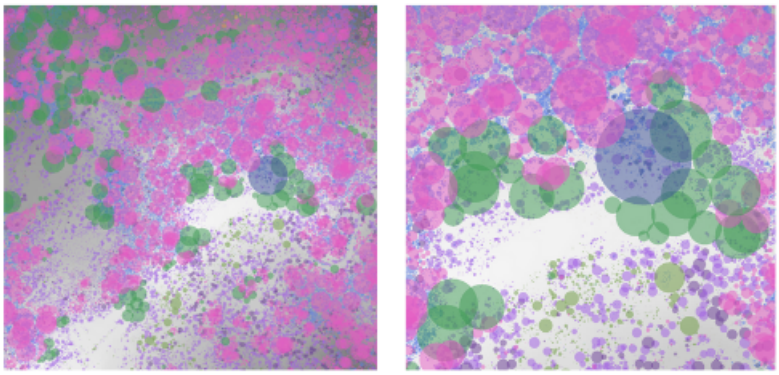

Figure 7: Zooming

A.4. Filtering

- *Filter species:*
  <Enable> or <Disable> the checkboxes on the "Species" tab on the east panel, to show or hide specific species. This will also update the number of each species that are visible in your perspective.

Figure 8: Filtering Species

- *Filter by height or radius:*
  Input the minimum heights of radius and heights on the “Filter” tab. After inputting each value, the visualised model is automatically updated.

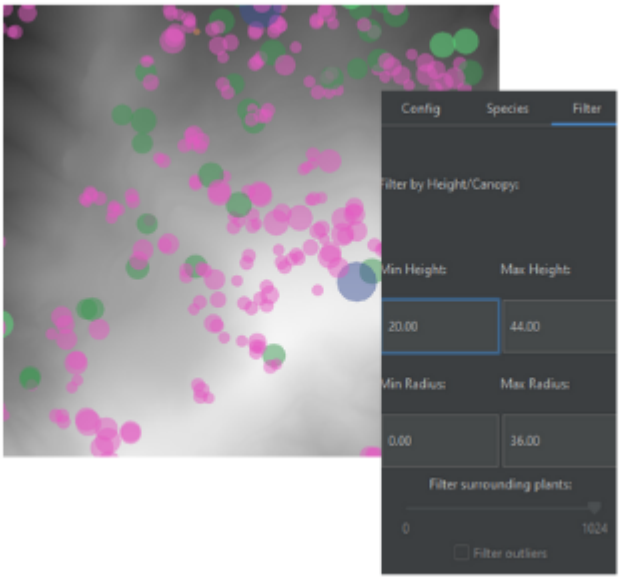

Figure 9: Filtering by height and radius

- *Select with radius:*
  After selecting a plant you are able to reduce the number of plants shown on the map. To do this <Enable> the “Filter outliers” checkbox. You can now shift the slider “Filter surrounding plants:” to show or hide plants within a radius of the selected plant.

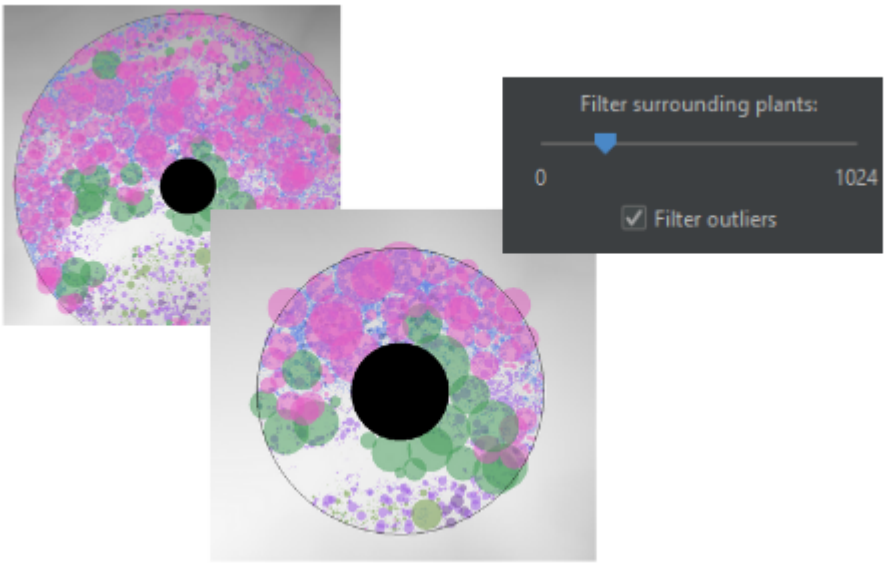

Figure 10: Selecting with a radius

A.5. Fire Simulation

- *Rendering a fire simulation:*
  Open Simulate Fire by performing a <Click> on the button ”Simulate Fire” that is located on the east

panel. Set a tree on fire by performing a <Click> within its canopy radius. The tree will now be highlighted 'yellow'. <Click> "Run" to begin the simulation.

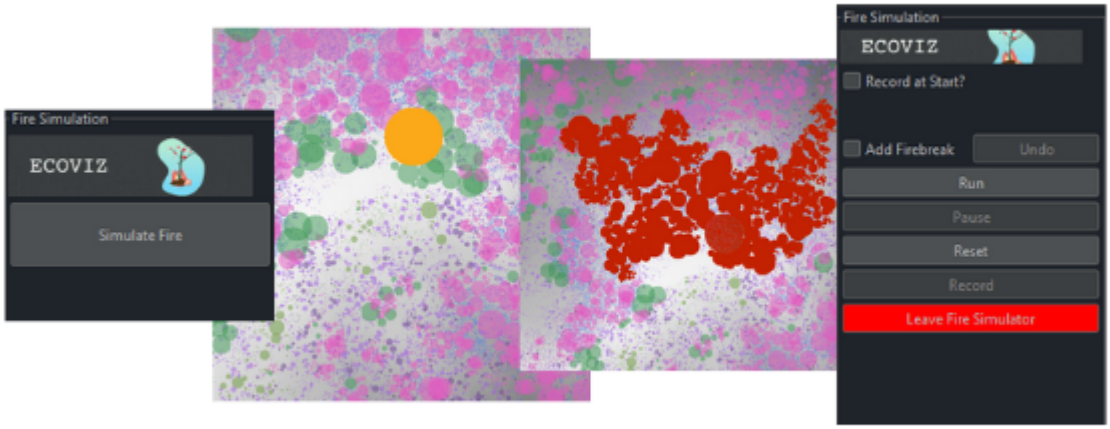

Figure 11: Running a fire simulation

- *Recording a fire simulation:*
  If you wish to record the Fire Simulation. <Enable> "Record at Start?" or <Click> "Record" when you wish to start doing so mid simulation. <Click> "End Record (closes session)" when you wish to. This will bring you to the "Scrubbing UI". Here you can <Scrub> through the rendered simulation and <Click> play or pause.

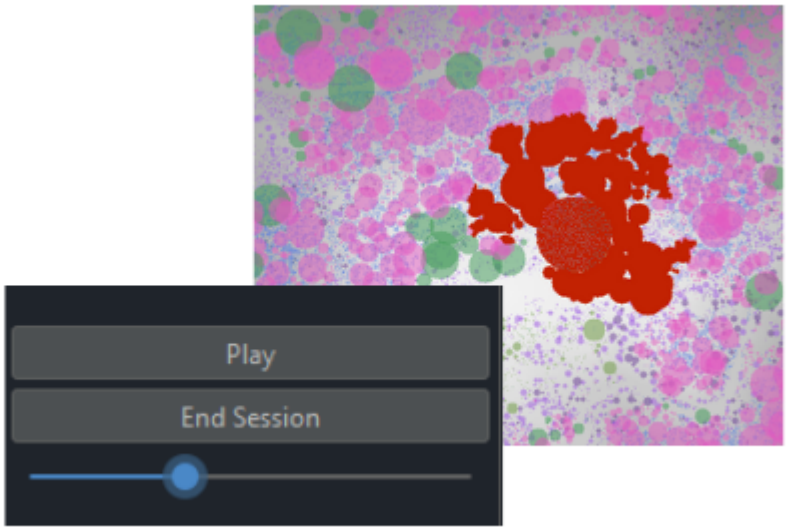

Figure 12: Scrubbing a recorded fire simulation

- *Fire Breaks:*
  Adding a fire break allows you to see the impact on the fire simulation after removing selected trees.

<Enable> the checkbox “Add Firebreak” to enter the ‘Fire Break Mode’. To place a fire break, <Click & Drag> on the map to remove selected trees. If you would like to undo the previous fire break, perform a <Click> on the “Undo” button.

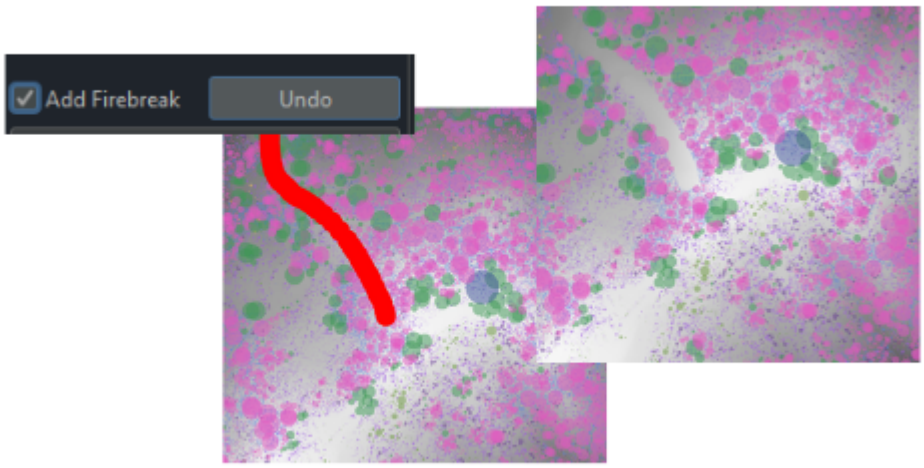

Figure 13: Applying a fire break

B. Use Case Narratives

Load in simulation input files. Upon starting the application, the user may click on the ‘File import’ button and be prompted to select the relevant files needed to generate an accurate ecosystem model (.elv, .pdb x2, .spc), from their computer/devices directories. This requires that the plant, species and terrain (elevation) data of the desired area/biome has already been collected and formatted correctly into files with the correct extensions. Selecting 4 valid files will cause the system to begin processing the data from these files, populating the model and rendering the 2D landscape and GUI. A possible alternate flow of events would be if the user loads in incorrect files, in which case the application would display an error message to the user detailing which file-types should be chosen and will then once again wait for valid files. Another option would be for the user to simply close the window, in which case the application would terminate.

Load in new files while the application is running. Once a correct set of files have been loaded in, the visual model will be generated and the GUI will load. If a user then wishes to load in files for a different biome, they can select the Menu button, select Load Files and will again be prompted to select a valid set of 4 files. From here the possible flows of events are identical to the previous use case, except that if the user exits the file chooser window without choosing any files, the program will simply continue using the files already previously loaded (upon startup).

Zoom/isolate environment. After a given model has been generated from valid input files, the 2D visualization of this data will be displayed. The initial view will be an overview of the entire loaded biome. Should the user wish to zoom in to a certain subsection of this area, they may use the mouse scroll wheel to directly manipulate the simulation, scrolling forwards to zoom in and backwards to zoom out again. The mini-map displayed in the corner will adjust accordingly, displaying the context of the current view in terms of the entire ecosystem. An edge case/alternative event may occur if the user attempts to zoom in or out too far. If the user attempts to zoom-in past the application defined limit, or out past the original overview zoom of 100%, the application will simply ignore this user action, and remain at a valid zoom

level.

Drag/scroll across terrain. Once a user has zoomed in to a certain degree on a particular subsection of the landscape they may wish to transition to examine another area, at the same zoom level (level of granularity). The user may drag their mouse across the model environment to pan across to other areas. An alternate event could be that a user tries to drag their view outside of the original borders of the entire biome. This is prevented by the system. The user may also not drag at 100% (fully zoomed out) zoom.

Filter the simulation. Once the simulation data has been loaded and processed, and the environmental visualisation of all plants has been displayed, the user may opt to hide certain species or plant layers from view through various filtering options in the filtering and species tabs.The specifics of each filtering use case are outlined below.

Filter by height/canopy radius. The user can enter a combination of values such as the minimum and maximum height and/or radii that must be met, and every plant that fails to fall within this range will be removed or hidden from the view. Any invalid entry, such as a negative height or minimum height larger than a maximum height, will be discarded by the system.

Filter by radius around selected individual. If a plant is selected (discussed in the 'Select an individual plant' use case) the user may choose to filter out the outlying plants - that fall outside of a specified radius surrounding a selected plant (use case discussed below). A user may adjust this filter radius by use of a slider, and can at any point remove this filter entirely, by deselecting the 'Filter outliers' checkbox.

Filter by plant layer. The ecosystem is separated into two categories: 'canopy' and 'undergrowth'. The user may choose to hide a layer to gain more insight on the other, or hide both (all plants) to inspect the terrain/elevations. The filtering of these layers are controlled by two separate checkboxes.

Filter by species/group. The plant data required as input to the application requires that plants are categorised into species. The user may choose to hide a single species or combination of species from view, allowing them to gain more insight on which species are of greater abundance, species elevation distribution etc. Each species found in the ecosystem is represented by a dedicated checkbox, allowing the user to de-select species one at a time, with an immediate visual response each time. An alternate use case could be the user wishing to load in a new set of data files, corresponding to another biome. In this case, all filtering options will be reset to default and all species and plant layers found in this new ecosystem will be initially displayed.

Select an individual plant. A user may choose to selectively view details of certain desired plants or species in the model. The user can click on any plant within the view and the relevant plant's species name and dimensions will be displayed in the Details tab. The selected plant will also be highlighted. The use case requires that valid plant data has been input to the application, and processed to generate the visualisation. An alternate flow of events would be if the user selects any point in the environment in which there are no plants. In this case, any previously selected plants' highlight and details will disappear, as the program recognises that the user has tried to change their selection, but no new highlight or plant data will be displayed. This also forms a way for the user to reset any undesired selection.

Show entire selected species. Once an individual plant has been selected by the user (as mentioned in the previous use case), they may wish to view species-wide details of all plants belonging to that same species, found in the given area. Provided that a plant has already been clicked/selected, the user may select the 'Full Details' checkbox. This will cause the system to identify all plants of the species, highlight them and display their aggregated details, such as minimum and maximum dimensions in the Details tab. An alternate flow of events would be if the user wished to change the colour in which the

individual plant/species was highlighted. To do this they may, at any time, select a new colour using the 'Colour Fill' button and a pop-up colour palette. As another alternative, if the user wished to return to only viewing a single individual, they may deselect the checkbox, which will revert the highlighting and details back to a single plant.

Change wind direction/speed. The fire-spread simulation that our application creates is affected by a user-set wind direction and wind speed. The user may change these values using two separate sliders found in the Configurations tab. Any fire simulation which runs will use the current value of the wind speed and direction, as it starts, although an alternative flow would involve a user changing the wind direction or speed during the simulation, in which case, the new wind settings will be reflected dynamically. A compass in this same tab visually displays the current selected wind speed to the user, allowing for intuitive adjustment.

Run fire simulation. A large function of our application is its capability to perform fire simulations. A user may choose to enter the fire simulation mode of the application at any time, by selecting the 'Simulate fire' button. This will then provide the user with the full toolset for creating and customizing a fire-spread simulation. The most common use-case would be for the user to first add a fire seed-point to the landscape, by clicking on any desired plant. Once the user has selected at least one fire seed, the application will wait for the user to select 'Run', when it will begin to simulate the spread of fire from that location according to any environmental variables set by the user. The user may also pause/resume or reset the simulation, by selecting the corresponding buttons, as well as start a recording of the simulation at any time (discussed below). An alternate flow of events would be if the user wishes to terminate a fire simulation session at some point during the running of the simulation. This is possible through the 'Leave simulation' button, which will reset the application and view to default.

Record fire simulation. Before the user chooses to 'run' their fire simulation, they may select an optional checkbox, to 'Record at start'. Selecting this checkbox will cause the system to sample images from the fire simulation once it begins, in order to compile a watchable video of the fire-spread. This video will be available after this firebreak has completed, or when the user chooses to End Recording, by selecting the corresponding button. The video can be played or paused and may also be traversed using a scrubbing bar. An alternate flow of events could be the user resetting and leaving the fire simulation mode, before ending the recording. In this case, all saved images will be discarded by the application. Another case could be the user not selecting the 'Record at start' option but wishing to record the simulation from a mid-way point. This can be achieved by selecting the 'Record' button at any point in the run.

Create a firebreak. Before the user sets fire seed location(s), the user may opt to create a fire break in the ecosystem to better understand the potential impact this may have on the spread of a fire. The user would click and drag across the terrain, dynamically removing any vegetation in a radius around the user's cursor (circular brush), hiding the plants from view. An alternate flow of events in this case could be the user wishing to 'Undo' a firebreak, in which case they may remove any firebreaks, in the order of Last-in-First-Out, by selecting the 'Undo' button.

C. Unit Test Descriptions

Terrain.java.
A basic terrain object is instantiated through the default constructor and then initialised further with passable variables through the class's accessor methods. An assertEquals test is then run combining each accessor and mutator to confirm that the values set are correct and won't cause any issues in running the program. The most notable tested methods include the getDimX and getDimY methods which return the x and y dimension of the terrain respectively.

Species.java.

Each species object is instantiated through a custom constructor and then initialised further with passable variables (such as valid canopy and undergrowth arrays as well as height, ID, name and colour values) through the class's accessor methods. An assertEquals test is then run comparing each accessor and mutator to confirm that the values set are correct and won't cause any issues in running the program. The most notable methods being the accessor and mutator methods for common/latin name, species ID, min/max height, canopy filtering and number of plants.

SimController.java.
The simulation controller's code variable initialisation, and code, are tested through checking if each new main method run creates the same valid objects, namely GUI, terrain, canopy plant layer, undergrowth plant layer, and controller. Testing the validity of this code ensures that the simulation controller will perform consistently across every application run.

PlantLayer.java.
Each plant layer object is instantiated through the default constructor and then passed valid data through the mutator methods which fully initialise the rest of the object. The number of species, location dimensions, and species list are set (through mutators such as setAllSpecies, etc) and then retrieved through the respective accessor methods and compared through an assertEquals statement to ensure the layer instantiation has occurred correctly.

Plant.java.
A plant object is instantiated through a custom constructor and then initialised further with passable variables (including a valid species ID, plant ID, coordinates, height, and flags indicating whether or not the plant is to be filtered out or not) through the class's mutator methods. An assertEquals test is then run comparing each accessor and mutator combination to confirm that the values set are as expected and won't cause any issues in running the program. The most notable methods include the isInFireBreak method which indicates whether or not the plant has been removed through the user implementing a firebreak, as well as the regular accessor and mutator methods for which correctness is now ensured.

MiniMap.java.
The MiniMap's constructor is tested by checking if each new instantiation with the a custom ImagePanel object creates the same valid MiniMap object. This 'test' MiniMap is then used to test the setZone method which is passed the dimensions as well as the top left coordinates. If this is successfully sets MiniMap fields to the correct values, without errors then the test will pass and ensures the minimap position guide can be correctly adjusted.

ImagePanel.java.
This class couldn't be realistically tested in the same way as the other classes as it's methods are all focussed on generating large BufferedImages and displaying/painting them to the screen. The contents of these 1024x1024 images could not be realistically tested with Unit testing.

Gui.java.
The GUI object is instantiated through the default constructor and then initialised further with passable values through the class's mutator methods. An assertEquals test is then run comparing each accessor and mutator combination to confirm that the values set are correct and won't cause any issues in running the program. The most notable tested methods include the setSpeciesDetails, setMousePositisions, setFilterList (a list containing which plants to be filtered), and clearFilters.

Fire.java.
A fire object is instantiated through a custom constructor which is being passed custom created dimensions and ID locations of both the undergrowth and canopy. The getDim and isFire methods are then

called and compared to their expected values in an assertEquals command.

FileController.java.
The FileController object is instantiated through the default constructor and then tested on its methods including readElevation, readSpecies, readLayer, getTotalSpecies, compareAndSetMaxHeight, compareAndSetMaxRadius, as well as the getMaxHeight and getMaxRadius methods. Each mutator method is checked for correctness through comparing its initialised values with the values expected through an assertEquals test to ensure that the class functions and reads the given datasets as expected.

Controller.java.
This class operates almost exclusively through listening to user interaction with a fully displayed Gui and could not be simply Unit tested in the same way as the other classes as it relies almost entirely on calls and data from those classes.

D. Testing Result Tables

D.1. Test 1: Zooming Response Time at worst case maximum zoom

D.1.1. 256x256 Data Set

Improved Method (ms)	Old Method (ms)
95	584
84	721
94	534
87	664
96	608
86	559
97	560
87	611
96	603
87	675
90.9 ms	**611.9 ms**

D.1.2. 512x512 Data Set

Improved Method (ms)	Old Method (ms)
127	1553
110	1893
122	1388
110	1705
120	1512
110	1459
120	1656
111	1584
120	1458
110	1451
90.9 ms	**611.9 ms**

D.1.3. 1024x1024 Data Set

Improved Method (ms)	Old Method (ms)
120	2191
112	2704
125	2360
112	2614
119	2307
113	2478
120	2073
114	2679
119	2195
111	2540
116.5 ms	**2414.1 ms**

D.2. Test 2: Comparison between fillOval and circle drawing algorithm full visual render time

D.2.1. 256x256 Data Set

Improved Method (ms)	Old Method (ms)
242	566
276	630
244	635
219	576
218	561
360	574
386	608
233	616
257	597
380	584
281.5 ms	**594.7 ms**

D.2.2. 512x512 Data Set

Improved Method (ms)	Old Method (ms)
295	1191
309	1301
284	1525
325	1300
282	1256
260	1362
395	1692
291	1202
244	1225
455	1399
314 ms	**1345.3 ms**

D.2.3. 1024x1024 Data Set

Improved Method (ms)	Old Method (ms)
389	1613
343	1787
246	1920
331	1768
335	1609
408	1679
319	1929
295	1778
245	1742
303	1892
321.4 ms	**1771.7 ms**

D.3. Test 3: Sorting the list of plants

The total number of plants in each dataset are as follows:

- 1024 data set: 729 727 total plants
- 512 data set: 386 099 total plants
- 256 data set: 113 292 total plants

1024x1024	512x512	256x256
425	206	89
451	226	86
385	200	96
420	212	83
392	215	109
471	251	85
398	203	77
396	261	83
419	216	85
426	223	91
418.3 ms	**221.3 ms**	**88.4 ms**

References

[Brainvire(2021)] Brainvire.com (2004) *Six benefits of using mvc model for effective web application development* https://www.brainvire.com/six-benefits-of-using-mvc-model-for-effective-web-application-development/ (Accessed 9 October 2021). Brainvire.

[Tutorialspoint(2021)] Tutorialspoint.com (2004) *Design Patterns - MVC Pattern* URL (Accessed 9 October 2021). Tutorialspoint.

[Formdev(2021)] Formdev.com (2021) *FlatLaf - Flat Look and Feel* https://www.formdev.com/flatlaf/ (Accessed 9 October 2021). Formdev.

[Un-spider(2021)] Un-spider.org. (2021) *Forest Fire — UN-SPIDER Knowledge Portal* https://www.un-spider.org/disaster-type/forest-fire (Accessed 9 October 2021). Un-spider.

[EDUCBA(2021)] EDUCBA.com (2021) *Java Swing vs Java FX — Know The 6 Most Awesome Differences* https://www.educba.com/java-swing-vs-java-fx/ (Accessed 9 October 2021). EDUCBA.

[Kelly K. (2021)] researchgate.net (2021) *Kellys 22 colours of maximum contrast* https://www.researchgate.net/figure/Kellys-22-colours-of-maximum-contrast_fig2_237005166 (Accessed 9 October 2021). Kelly K.

[Encyclopedia Britannica (2021)] britannica.com (2021) *Phi phenomenon — visual illusion* https://www.britannica.com/topic/phi-phenomenon (Accessed 9 October 2021). Encyclopedia Britannica.