CMP-7009A Advanced Programming Concepts and Techniques

Project Interim Report - 14 November 2018

Evolution Sandbox

Group members:
Benjamin Longhurst, Rupert Hammond, Ryan Phelan, Travis Payne

School of Computing Sciences, University of East Anglia

Version 1.0

Abstract

Chapter 1

Introduction

An evolution simulation attempts to represent the way a set of organisms evolve within a limited ecosystem, typically a number of biological algorithms such as fitness functions and crossover are employed to dictate how this plays out. One such example of an evolution simulation is Conway's Game of Life. Created in 1970 by John Conway, the simulation takes place on an infinitely sized grid where each cell is either live or dead. It progresses according to a set of simple rules [1]:

- A live cell with less than two live neighbours becomes dead
- A live cell with more than four live neighbours becomes dead
- A dead cell with three live neighbours becomes alive

Conway's Game of Life is often praised for its ability to show how simple rules can spawn complex evolutionary patterns [2]. This project will tackle evolution simulating by taking inspiration from Conway's Game of Life to produce a piece of software it terms an "evolution sandbox"; a simulation with emphasis on real-time manipulation and customization which will allow the user to observe the outcome of their actions on the ecosystem.

1.1 MoSCoW

In order to better understand the scope and priorities of the project, a set of analysis' were carried out with the goal of producing a MoSCoW analysis. Firstly, the basic requirement analysis was defined as follows:

- Organisms should act based on personal attributes, similar to the rule system in Conway's Game of Life
- Organism attributes should be customisable on the fly
- Organism attributes should mutate over generations using a crossover algorithm
- Organisms should utilize logical path finding when seeking
- The ecosystem should reach equilibrium when left to its own devices
- The simulation should be able to handle a large number of organisms without noticeable lag
- The simulation should employ realistic biological algorithms where possible
- The UI should be clean, simple and professional
- The graphics should faithfully represent the underlying simulation

Next, an Object Oriented Analysis was carried out to identify the objects of the system to later be the focus of the priorities in the MoSCoW analysis:

- End Goal: Biological Evolution Sandbox
  - What is required?
    - End game, stable ecology
      - Net number of organisms doesn't change
    - Food, vegetation, other organisms
    - Water
      - Some organisms can go in water
    - Weather
      - Hot and cold
      - Different types of weather
    - Statistics
    - A log
    - Disease
    - Natural Disasters
    - Terrain
    - Live edit of organisms
    - Tile-based graphics

Finally, taking these objects and systems a MoSCoW analysis was produced:

Must Have	• Organism life cycle • Genetic crossover algorithm • Organism attributes • Live edit of organisms • Simple 2D graphics • Herbivores and natural food sources
Should Have	• Weather/disease system • Advanced path-finding algorithm • Carnivores and predator/prey organisms • Terrain variation, e.g. grass, mountainous, water • Ability to pause, speed up and slow down simulation
Could Have	• Natural disasters • Speciation (new species forming from heavily mutated organisms over time) • A game log with charts and text output • Spritesheet animation • Particle effects, e.g. weather effects, running water, blood
Won't Have	• 3D graphics • Scale realism

In general, the "Must Have" objectives are those identified to be necessary to a bare minimum working product, while "Should Have" is considered a bare minimum submission. The logic being that these "Should Have" objectives could also potentially be developed further to have enough depth to utilize advanced programming techniques or have the simulation revolve around them. "Could Have" objectives are those which are considered incredibly difficult or potentially out of scope. For example, particle effects and spritesheet animation will not improve the depth of the simulation and would require effort in areas which are not programming-related. Natural disasters and speciation on the other hand would be difficult to implement while maintaining equilibrium within the ecosystem. Finally, the "Won't Have" objectives are identified to disproportionately increase the simulation's complexity when compared with the pay-off for implementing them. The implementation of this analysis is discussed in detail within Section 3.1.1.

1.2 Report structure

This report will cover a brief background of evolution and the algorithms which attempt to simulate it in Section 2. Section 3 details the various advanced programming and project management techniques utilized in the project. Tools used and implementation details are outlined in Section 4. Finally, the product's quality will be tested and verified through testing in Section 5 and drawn to a conclusion in Sections 6 and 7.

Chapter 2

Background

Chapter 3

Methodology

3.1 Agile Methodology

The project is managed according to the Agile methodology, specifically Scrum. The team meet twice per week and start with a stand-up meeting where the team give updates on the tasks they are working on and discuss solutions to problems that may have been encountered. In keeping with the Agile methodology, iterative version releases are promoted with the rule that each lab meeting must have a merged and bug-free master branch. Development is split into two to three week sprints, with the objective of producing a new product version [Section 3.1.1]. GitHub is used as the centre for project management using a combination if it's project boards, where each board corresponds to one sprint and therefore one product version 3.1, and it's issue tracking 3.2.

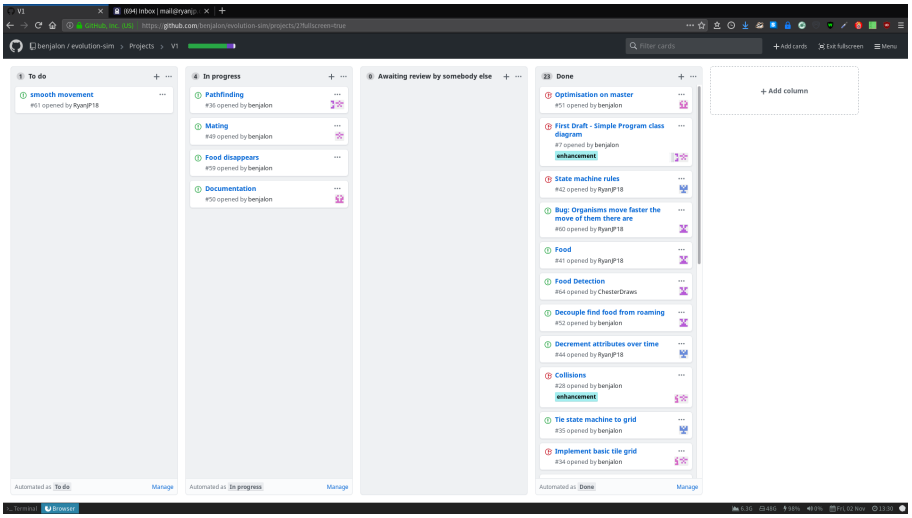

Figure 3.1: GitHub Project Board - Each of these boards represents one full sprint and one version of the software. When all of the issues are completed, the board is closed and a sprint planning meeting occurs.

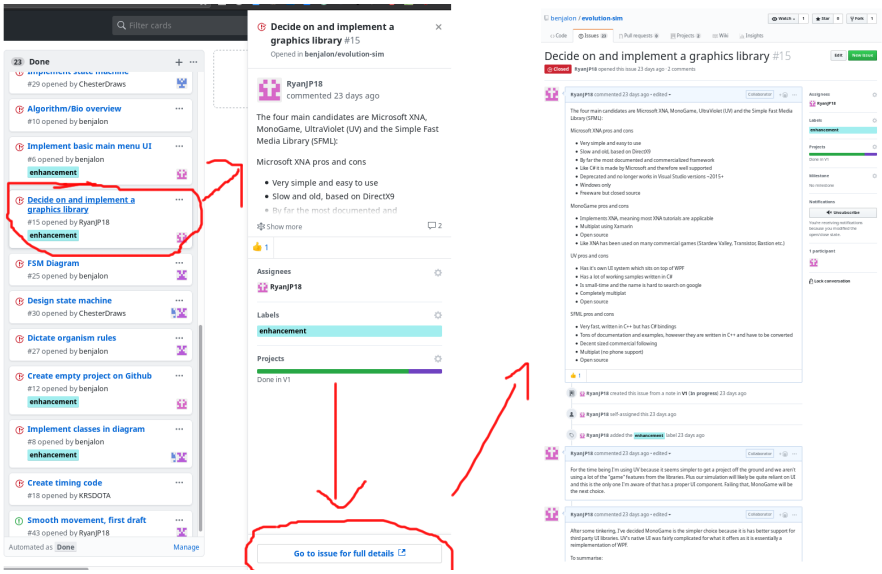

Figure 3.2: GitHub Issue Tracking - Issues are created during sprint planning meetings, where necessary the person working on the task is expected to update the ticket with information that will need to be communicated to the team or included within any reports.

Within Git, each ticket is undertaken on its own branch and merged to master when completed. There is a lock on the master branch to prevent direct commits to enforce this rule, furthermore before a branch can be merged to master there is a further lock to ensure that a pull request is submitted and checked by at least one other person. This system helps ensure the quality of master for branching at all times, avoiding situations where development is halted due to a buggy code-base. Additionally, pull requests can function as a form of white-box testing as discussed in 5.

Other project management tools include Microsoft OneNote for collaborative documentation and WhatsApp for quick discussion and scheduling.

3.1.1 Versioning System

In keeping with the spirit of iterative development, the projects aim to produce a new product at the end of each sprint in order to keep systems well-rounded and stay on schedule. This also means that at any given time, the active project board matches the current release version. During sprint planning meetings, issues for the upcoming version are created against the MosCoW analysis 1.1:

Version	Goal	Deadline
0	A proof of concept to test the chosen technologies	Tuesday week 2
1	A fully functioning basic simulation with all of the “Must Have“ components from the MoSCoW analysis	Friday week 6
2	Improvements on V1 simulation with tasks taken from the “should have“ objectives. In particular, an advanced path finding algorithm such as A*, improved crossover algorithm, carnivores and a better time system.	Undecided
3+	Undecided	Undecided

3.2 Code Architecture

The SOLID principles define five guidelines to ensure code is maintainable and easy to understand [3]:

- Single Responsibility: Every class should have only one responsibility to prevent the class being susceptible to requirement changes.

- Open-Closed: Classes should be open to extension but abstraction should be used to avoid the need for rewrites on the extended code.
- Liskov Substitution: Derived and base classes should be substitutable.
- Interface Segregation: Keep interfaces simple to avoid bulk from implementing unnecessary properties.
- Dependency Inversion: Keep abstract code abstract by avoiding dependencies on low level code.

While these principles each affect the architecture in some way, in particular a great deal of effort is made to ensure that each class has a single responsibility. An example of this is in separating the Grid from the StateMachine despite them operating in similar areas of the simulation. Grid is instead used as the "graphical brain", positioning organisms and drawing them at their positions, whereas the StateMachine is used as the "logical brain", dictating the behaviour which causes the organisms to be repositioned in the first place. Furthermore, the use of MapItem as a swappable base for Organism and Food would not work if the Liskov Substitution principle was not applied.

To ensure a clean and consistent code-base, the official Microsoft C# conventions are applied where possible. Notable examples include placing braces on a new line, using camelCase for variable names and avoiding one line if statements [4].

3.3 State Management

Within the simulation all organism behaviour is dictated by a state machine which contains a set of states and rules for moving between them. The StateMachine class is passed State objects which are essentially a lookup table for each state transition. Transitions use a pair of enum values, OrganismState and Action, where OrganismState refers to the current state of the Organism and Action is the event responsible for causing the transition. Based on this supplied information, the StateMachine determines which state the Organism should transition to and throws an exception if it determines the move to be illegal, for example it is impossible for an Organism to move directly from "Mating" to "Eating" through the Action "FoundFood" as the Organism must be in the "Roaming" state in order to reach this clause.

In addition to these state management tasks, the StateMachine also contains two methods to drive organism behaviour: CheckState and DetermineBehaviour, which are both called every cycle of the Update loop. CheckState is responsible for checking that each Organism is in the correct state based on the rules defined for the simulation, for example if an organism is currently in the "Roaming" state, but it's hunger levels have dropped below 0.4, this would mean that the organism must now move into "SeekingFood". After the state is checked, DetermineBehaviour defines how the Organism should behave within this state.

3.4 Crossover Algorithm

The crossover algorithm is currently a work in progress.

3.5 Grid System

A common method of implementing movement in a simulation is by constraining it to a grid, this is useful because the number of movement states becomes finite and better lends itself to algorithms in areas such as pathfinding. In terms of efficiency a grid also reduces the need for collision detection between simulation elements because when an organism tries to move from one tile to another, the grid can simply accept or reject the

move based on whether the destination tile is occupied. On the other hand, this limited movement is less realistic and does not necessarily reduce code complexity where the grid must constantly observe the positions of its elements which requires a fair amount of manipulation to its collections.

In terms of code, the Grid class tracks the positions of all Tiles within the simulation and supplies some simple methods for interacting with them, for example to check whether a given Tile is occupied. In order to achieve this, the Grid contains a jagged two dimensional array of type Tile, where each index within it maps to a Tile's actual screen position with the equation:

$$tilePos = offset + (tileIndex * tileSize)$$

When an organism wishes to move, a method on the Grid called MoveMapItem is passed as parameters the MapItem requesting the move and a destination it wishes to move to. Since Tiles are always in fixed positions from the simulation start, the Grid simply has to look up the destination from its fixed position array and check whether it is occupied and then ensure that the MapItem position is adjacent.

3.6 Pathfinding Algorithm

Pathfinding is a key component in the simulation and three main methods were considered for it: depth first search (DFS), Dijkstra's algorithm and the A* algorithm. In terms of simplicity, DFS produces fast results but is a naive algorithm and requires a large amount of time and memory to find the shortest route. Dijkstra's algorithm is an improvement because it applies a difficulty value based on the movement cost to a destination but is expensive as it compares equally and needlessly expands difficult locations without considering direction because it does not use a heuristic. Finally, at least for single-point to single-point searches like those used in this simulation, A* is considered the fastest algorithm [5] due to improving upon Dijkstra's with a heuristic.

$$A * Algorithm : f(n) = g(n) + h(n)$$

In the above formula g(n) refers to the difficulty of a tile which is an important consideration in handling different terrain types such as hills or water. h(n) is the diagonal distance between the two points and f(n) refers to the sum of g(n) and h(n).

In terms of implementation, A* is incorporated by converting each Tile into a Node object. These Nodes keep a reference to the previous Node on the path, its Grid position, the destination's distance and the difficulty in reaching it. To calculate the g(n) difficulty value the terrain type of the Tile is taken into account and for the h(n) distance value the diagonal distance between the two points is calculated taking the maximum value of either the goal Tile's x coordinate minus the current Tile's x coordinate or the same for y coordinates.

To build up a path, two lists track Nodes to be evaluated: open and closed. When the algorithm starts, the Node that an Organism currently occupies is placed at the top of open. Open is iterated to evaluate which Node within the list has the lowest f(n) and expands it. Expanding consists of storing all of the adjacent Tiles on the Grid as Nodes and calculating their f(n) and adding them to open. If any of the Nodes evaluated are the goal Node then the loop ends and returns it. A list containing the goal Node and the path taken to get there is then created by repeatedly following the previous Node reference backwards from the goal to the start.

3.7 Optimisation

Optimisation is considered an area of importance due to the simulation's requirement in handling a large number of organisms without noticeable lag. By default, MonoGame's

render loop calls two methods at a speed of sixty times per second: update and draw. Should the logic fail to complete within this 16-17ms time-frame then it delays the draw call, which lowers the simulation's frame-rate. This has further knock-on effects where the draw calls become progressively more delayed and eventually cause input lag on the UI.

The update method is essentially the primary path through the code and handles all of the computation and organism logic. This method calls into several for loops to cycle through the various organisms and map items which make up the simulation. There can be several hundred objects to iterate through during any given Update loop, which as previously mentioned occurs sixty times per second. Keeping this entire iteration within the acceptable 16-17ms time limit has requires consideration from an optimisation perspective.

Optimisation is a task better left until necessary following the argument that "premature optimization is the root of all evil [...]" [6], as doing so before it is necessary wastes time, introduces bugs and makes code less readable. However, at several points during the project, particularly during the first round of pathfinding implementation, performance was considered to be a problem. The A* pathfinding algorithm [3.6] requires that the program keep track of open, closed and expanded nodes within various collections and though there are many ways to implement the algorithm, they often involve some degree of manipulation. As previously mentioned, the update method is already within the sixty per second game loop and then within another nested loop for each of the tiles [3.5]. Before delving into loop micro-optimisation, the amount of loops and collection manipulation was cut down as much as possible.

To improve the performance of these loops, the standard loop micro-optimisations are applied. Variables are declared outside of the loop scope, collection length is cached locally ahead of time and high precision calculations are avoided. For example, the DateTime object was initially used to time organism movements but because it uses double precision values, calculations were causing a large slowdown so it was swapped for the better optimised GameTime object. Finally, the grid stores its tiles within a two dimensional array which could potentially be implemented in C# in two different ways: multi-dimensional arrays and jagged arrays. A jagged array is an array of arrays and allows these inner arrays to vary in length, whereas a multi-dimensional is a natural 2D array where the column count is uniform. The grid stores its tiles in a jagged array because within a jagged array, even if the arrays are the same length, it is able to iterate faster than a multi-dimensional array.

Finally, though it caused issue in this instance, from an architecture perspective the grid system is also a form of optimisation. By constricting organism movements to a grid the simulation is able to cut down on the need for collision detection by having organisms request their moves to the grid, which then makes the decision of whether the move is legal. Collision detection would otherwise have been a bottleneck for the system which would have been exponentially slower as more organisms are added because each organism would need to check the position all of the others. With the tile system in place however, there are always a set number of tiles to iterate over rather than a growing list of organisms with references to each of the others.

3.8 Multi-threading

To better improve the performance of pathfinding, it is being moved to different threads. This is a work in progress.

Chapter 4

Implementation

4.1 Tools

Three programming languages were considered for the project: C++, C# and Java. The simulation was identified to have a large dependency on computation due to the fact that there could be upwards of one hundred organisms on screen at any one time and each of them would require state management, path finding and collision detection. For this reason, C++ seemed to be the natural choice due to the speed benefit of dynamic memory management. However, upon further research it was found that C# had a more diverse set of 2D graphics libraries as C++ libraries were typically focused on 3D rendering. Finally, Java was considered because the bulk of the team's experience was with the language, though because C# can be used as a drop-in replacement for Java this was seen as another benefit in using C#.

Since the availability of tools is dependant on the chosen language, the decision of a graphics framework was next. A comparison was made between several popular 2D graphics libraries, the four main candidates being Microsoft XNA, MonoGame, UltraViolet (UV) and the Simple Fast Media Library (SFML):

Library	Pros	Cons
Microsoft XNA	• Simple and easy to use • Very well documented • Well-used commercially • Supported by Microsoft who also made C#	• Slow and old, based on DirectX9 • Deprecated and no longer works in Visual Studio 2015+ • Windows only • Closed source
MonoGame	• Based on XNA with the same syntax, all of the XNA documentation is applicable • Multi-platform but requires Xamarin • Open source • Has seen use on commercial games (Stardew Valley, Transistor, Bastion etc.)	• Convoluted asset management system
UV	• Has a built in UI framework based on Windows Presentation Foundation (WPF) • Truly multi-platform • Open source	• Convoluted asset management system • Limited documentation, small time • Little to no commercial use
SFML	• Very fast, written in C++ but has C# bindings • Well documented • Some commercial use • Multi-platform but no phone support • Open source	• C# bindings are slightly behind on updates • Examples and documentation are written in C++ and require converting • Syntax is not as simple as other frameworks

Though UV was initially chosen due to its built-in UI support, where UI was deemed a core component of the simulation, the lack of documentation and convoluted XML system for managing assets meant that MonoGame was chosen instead. A third party UI library, GeonBit.UI, was chosen because although any 2D graphics framework is capable of rendering UI using sprite assets, one of the project requirements [1.1] was that the UI have a professional look. Additionally, UI elements such as lists and radio buttons were considered a likely requirement and would be difficult to implement with a graphics approach.

4.2 Proof of Concept

A proof of concept was made with the chosen technologies before any further planning took place because the team wanted to avoid a situation where the UML modelling and code architecture was planned according to a language or tool that was then realised to be a poor fit for the project. The aim of the proof of concept was to simply draw a number of organisms to the screen using the chosen graphics framework which could be manipulated using a simple place-holder UI.

Figure 4.1: Proof of Concept

The proof of concept was deemed a success as it was able to create organisms using the UI button, move them around and render them without lag.

4.3 Modelling

The class diagram Figure 7.1 was continually updated throughout the project and although it saw many changes, certain key themes remained constant throughout.

An important architectural decision made from the outset was to ensure the separation of concerns between the graphics, simulation and UI. It was decided that the graphical component should provide a representation of the simulation's output, but that they should be kept unaware of the inner workings of the other to better decouple their behaviour. Likewise, while the UI would be able to interact with the simulation, there was no need to have this behaviour tied in any way to the intricacies of it. As such, the relationship between the three key areas can be observed as limited on the class diagram.

Another theme present in the class diagram is abstraction. Through inheritance, MapItems are kept as generic inhabitants by the Tiles of the Grid. This means that the Grid can manage its Tiles and their inhabitant MapItems without particular knowledge of whether they are Organisms, Food or Obstacles, simplifying the implementation.

4.4 Version 1

4.5 Version 2

Version 2 is currently a work in progress.

Chapter 5

Testing

This section will be about the following:

- Pull requests as a form of white-box testing during development
- Unit testing with the built in C# tools
- Experimenting with several untouched simulations to ensure equilibrium is reached
- Experimenting with editing attributes and reaching equilibrium, likewise with natural disasters and disease
- A section on the bug fixing ticket system in place

Chapter 6

Discussion

This section will be about the following:

- Discussion of testing and experiment results
- Issues encountered: frequent rewrites, tangled state management early on, slowdown and lag, inconsistent coding standards, pathfinding necessitating optimisation and multiple threads, pull request system
- What went well/badly
- What could be improved

Chapter 7

Conclusion and Future Work

This section will conclude the MoSCoW analysis, discuss shortcomings and future developments with more time. It will avoid subjective opinions, rants and excuses.

Bibliography

[1] The Game of Life: a beginner's guide, 2014.
https://www.theguardian.com/science/alexs-adventures-in-numberland/2014/dec/15/the-game-of-life-a-beginners-guide
Website accessed 01st November, 2018.

[2] What is the Game of Life?, 2018.
http://www.math.com/students/wonders/life/life.html
Website accessed 01st November, 2018.

[3] SOLID Principles made easy, 2017.
https://hackernoon.com/solid-principles-made-easy-67b1246bcdf
Website accessed 03rd November, 2018.

[4] C# Coding Conventions (C# Programming Guide), 2015.
https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/inside-a-program/coding-conventions
Website accessed 03rd November, 2018.

[5] A* Search Algorithm, 2018.
https://www.geeksforgeeks.org/a-search-algorithm/
Website accessed 14th November, 2018.

[6] Donald Knuth. Computer programming as an art. *ACM*, page 671, 1974.

Contributions

The team has agreed on a 25% contibution each as of the time of writing. Note that these contributions are to date and not representative of planned features.

Member	Ownership of/Major Contributions	Assisted on/Minor Contributions
Benjamin Longhurst	• Project management [3.1] • A* Pathfinding [3.6]	• Grid/tile system [3.5] • Bio. algorithms research [3.4]
Rupert Hammond	• State management [3.3] • State machine rules • Bio. algorithms research [3.4] • Organism attributes	• Bug fixing [5] • A* Pathfinding [3.6]
Ryan Phelan	• Graphics/UI research and implementation [4.1] • Grid/tile system [3.5] • Optimisation [3.7] • Report writing	• Project management [3.1] • Code architecture [3.2]
Travis Payne	• Food system • Movement logic • Simulation flow [3.3]	• Code architecture [3.2] • Bug fixing [5] • A* Pathfinding [3.6] • Organism attributes

Other work such as the Proof of Concept, UML and analysis' were completed as a team with all members present.

Appendix A

Figure 7.1: UML Class Diagram