CMP-7009A Advanced Programming Concepts and Techniques

Project 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

An abstract is a brief summary (maximum 250 words) of your entire project. It should cover your objectives, your methodology used, how you implemented the methodology for your specific results and what your final results are, your final outcome or deliverable and conclusion. You do not cover literature reviews or background in an abstract nor should you use abbreviations or acronyms. In the remainder of the report the chapter titles are suggestions and can be changed (or you can add more chapters if you wish to do so). This template is designed to help you write a clear report but you are welcome to modify it (at your peril ...). Finally, a guideline in size is approximately 3,500 words (not including abstract, captions and references) but no real limit on figures, tables, etc.

Chapter 1

Introduction

[TODO: Need work] 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.

1.1 Aims and Objective

The aim of this project is to produce a graphical evolutionary simulation with some degree of interactivity. In order to meet this objective, a set of requirements were produced:

- 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 out their objectives
- 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

1.2 MoSCoW

To better understand the scope and priorities of the project, a MoSCoW analysis was produced (Figure 1.1). In general, the "Must Have" objectives are those identified to be necessary for an end-to-end functioning product, while "Should Have" is considered a bare minimum submission. "Could Have" contains a mix of objectives such as spritesheet animation would improve the simulation quality but are not necessarily crucial, or otherwise those such as speciation which would increase the simulation complexity substantially. The implementation of this analysis is discussed in detail within Section 4.1.1.

Figure 1.1: MoSCoW Analysis

Must Have	• Organism life cycle • Genetic crossover algorithm • Unique organism attributes such as health, age, strength, speed and resistances • Organisms state determined by their unique attributes • Unique organism attributes should be editable on the fly • Simple UI Overlay • 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 • Program flow (start screen, simulation setup, end screen etc.)
Won't Have	• 3D graphics • Scale realism

1.3 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. Models and implementation details are outlined in Section 4. Finally, the product's quality will be tested and verified through experiments in Section 5 before being drawn to a conclusion in Sections 6 and 7.

Chapter 2

Background

[TODO: I'm not sure Conway's Game of Life should be the focus anymore]

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 become 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.

Chapter 3

Methodology

3.1 Agile Methodology

The project is managed according to Agile principles by implemented the Scrum framework. At most points in the project, meetings occurred twice per week (one lab session, one outside) and would begin with a short stand-up meeting. Iterative version releases were promoted with the rule that all feature branches should be merged into master before each week's lab meeting. Development as a whole was split into several versions (Section 4.1.1) which were tackled in two-three week sprints. GitHub was used as the centre for project management through a combination of it's Git Project Boards feature, where each board corresponds to one sprint and therefore one product version (Figure 3.1), and it's issue tracking (Figure 3.2).

Figure 3.1: GitHub Project Board

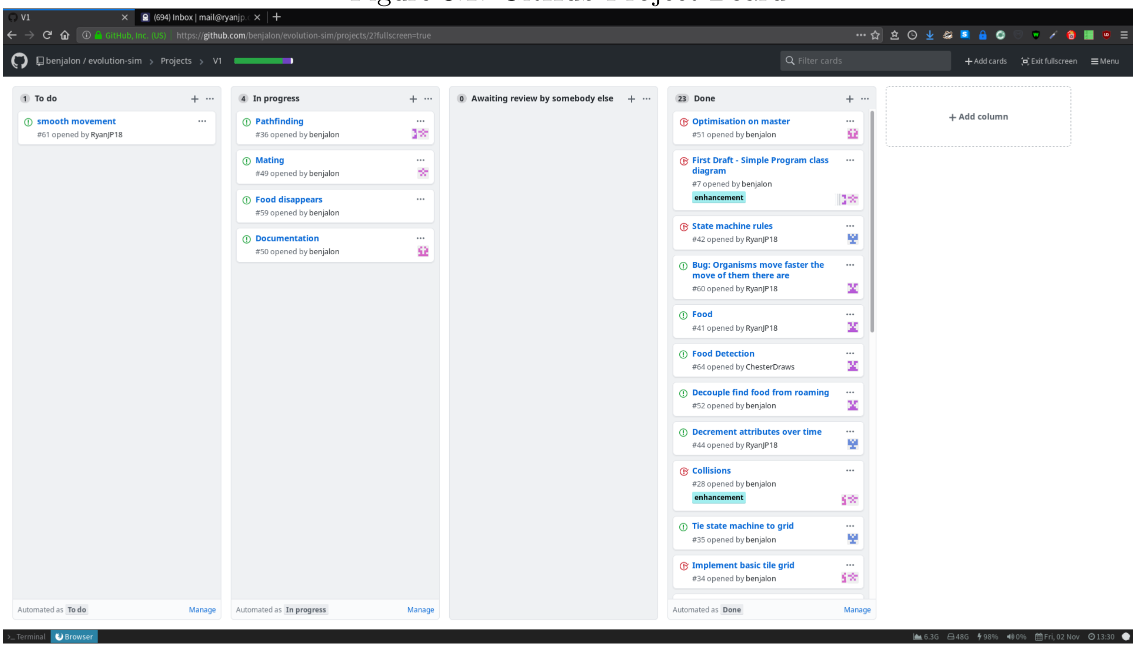
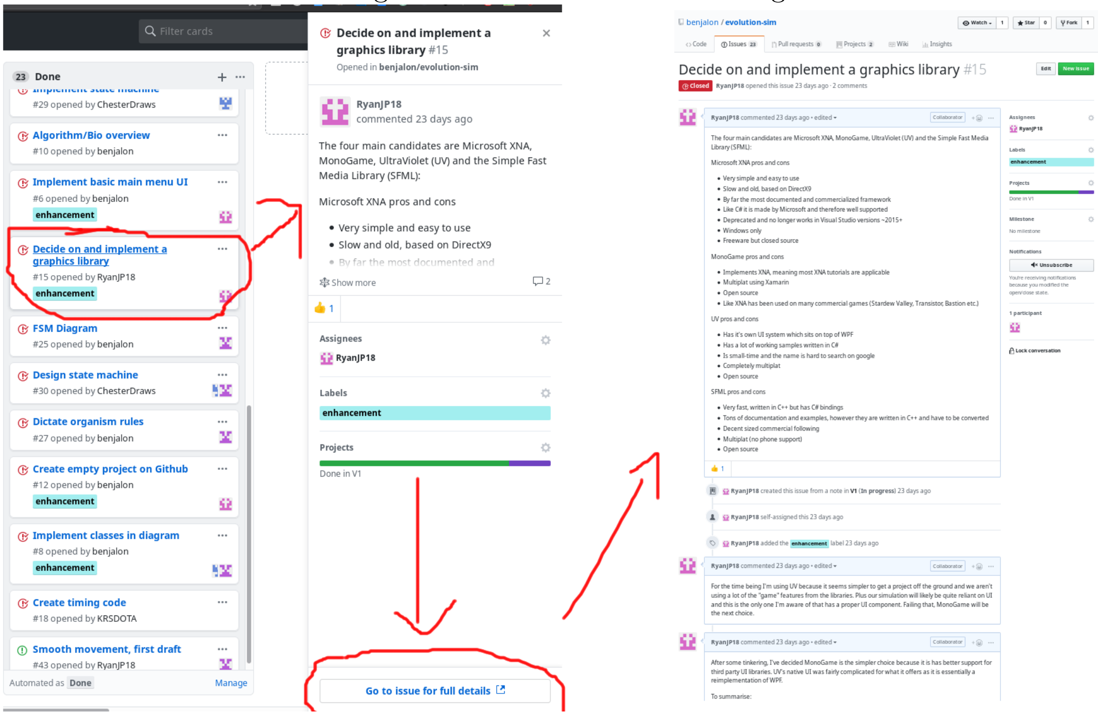


Figure 3.2: GitHub Issue Tracking



Using Git, each ticket is developed on a unique feature branch. When the feature is code-complete, it is sent for pull request (i.e. peer review) before it can be merged into master. This system is enforced through a merge lock on master, which prevents direct commits and only allows branches to be merged if the review was accepted by 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, these pull requests are able to function as a form of white-box testing as discussed in (Section 5).

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

3.2 Code Architecture

The SOLID principles define five guidelines which are used as the guiding principles in the project's architecture to ensure that 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.

An example of how this has influenced design can be seen in the choice to split Grid from StateMachine within their use inside Simulation. Although both handle moving and positioning organisms, Grid is treated 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. This separation of concerns can also be seen in the namespaces, for example EvolutionSim.StateMachine, EvolutionSim.TileGrid, EvolutionSim.Data and Evolution-Sim.UI.

Another example is through having GridItem function as a swappable base for Organism and Food. This is important due to the Tiles of the Grid accepting either as a potential inhabitant. This swappable nature 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 finite 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

TODO

3.5 Tile-Based Movement

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 for path-finding. In terms of efficiency, a grid also reduces the need for collision detection between simulation elements by mediating on the movement between tiles.

Within code, the Grid class holds a collection of Tiles within a jagged array which can each potentially hold a GridItem inhabitant. Within this collection, each index maps the tile to a screen position using the equation:

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

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 naive 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 based on the movement cost to a destination, but is expensive as it compares equally and needlessly expands difficult locations without considering direction. Finally, at least for single-point to single-point searches (as used here), 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. The h(n) is the diagonal distance between the two points and f(n) refers to the sum of g(n) and h(n).

A* is implemented by creating Node object for each Tile, keeping a reference to the previous Node, 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. Nodes to be evaluated are stored in open and closed collections which are manipulated until a path to the goal is found.

3.7 Optimisation

Optimisation is important in the simulation due to its requirement in handling a large number of organisms without noticeable lag. By default, MonoGame's render loop updates and draws at sixty frames per second. Should the update 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, it calls into several loops to cycle through the items which make up the simulation. There can be several hundred objects to iterate through during any given Update loop and keeping this entire iteration within the acceptable 16-17ms limit required a great deal of optimisation. Despite this, optimisation was a task left until it became a problem following the argument that "premature optimization is the root of all evil [...]" [6] by introducing bugs and making code less readable.

As the simulation grew in size, loop optimisation became an issue particularly for state management and path finding. To improve this, variables are typically declared outside of loop scope to prevent them being redeclared on each iteration. Other standard optimisations such as caching the collection length ahead of time and avoiding high precision calculations are also avoided. The grid was also changed from using a multi-dimensional array for its tiles to a jagged array which is more efficient to loop over.

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.

From an more architectural 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, which would otherwise have been a bottleneck for the system. This would have required exponentially more computation as organisms are added because each organism would need to check the itself against all others.

3.8 Multi-threading

TODO.

Chapter 4

Implementation

4.0.1 Modelling

TODO OOA and UML

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.

4.1 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 with C++ libraries 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#.

A comparison was made between several popular 2D graphics libraries available in C#, shown in Figure 4.1.

Figure 4.1: Graphics Library Comparison

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, the lack of documentation and convoluted XML system for managing assets meant that MonoGame was used instead. The third party UI library GeonBit.UI was chosen because one of the project requirements [1.1] was that the UI have a professional look and it also provided out of box support for radio buttons and lists.

4.1.1 Product Versions

In keeping with the spirit of iterative development, the projects aim to produce a new product at the end of each sprint. This ensures that at any given time, the active project board matches the current release version. Each version aimed to iterate upon the previous release with features taken from further down the MoSCoW analysis (Figure 4.1.1).

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 and “should have“ objectives: A* path-finding, improved crossover algorithm, carnivores and a better time system.	Friday week 8
3	Path finding performance enhancements and bug fixes	Friday week 10
3	More “Should Have“ objectives and missing functionality. Primarily eating, dying, mating, terrain and a time system.	Friday week 12
4	Final missing features: Weather system, carnivores and prey, particle effects.	Christmas
5	Loose ends, bug fixes and report updates.	Project deadline

4.2 Proof of Concept

A proof of concept was made with the chosen technologies to avoid a situation where the UML modelling and code architecture was planned according to a language or tool which was a poor fit for the project. It aimed 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 4.2: Proof of Concept

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