Life - Part 1: Assignment Specification

CAB201 Semester 2, 2020

Last Updated: 04/09/2020

Assigment Details

Progress Due Date: 13/09/2020 - 11:59pm
Final Due Date: 20/09/2020 - 11:59pm
Weighting: 35%
Type: Individual Project
Version: 1.3 - 04/09/2020

Change Log

There is no intention to have this assignment specification change over time. However, if anything does change or additional detail is added to the specification, the table below will be updated to reflect the changes.

Version	Date	Description
1.0	11/08/2020	Initial Specification
1.1	13/08/2020	Added warning about external libraries to the Academic Integrity section
1.2	17/08/2020	The minimum valid grid size has been increased to 4
1.3	04/09/2020	Added more test cases

Overview

In memoriam of the legendary mathematician John Conway, we will be be exploring one of his most famous creations as an exercise in programming principles - the Game of Life. Also referred to as Life, the game is a zero-player game, meaning that it's behaviour is determined by it's initial state with no additional input. Conway explains the rules for the Game of Life in a Numberphile interview (explanation starts at 1:00). The rules will be described in detail further into the document.

Your task is to create an command line application that can simulate Life. The application must be designed to be invoked from the command line, where game states and program settings can be provided. Additionally, the application must display a running animation of Life in a graphical display window (this functionality has been partially completed for you in the form of a small and simple API that you must utilize). This assignment is designed to test your ability to:

- □ Build command-line applications
- □ Develop high-quality file I/O functionality
- □ Design a solution to a non-trivial computational problem
- □ Understand and utilize an external API

Game Behavior

The game of life exists in universe comprised of a two-dimensional grid of square cells. The number of rows and columns in the universe may be specified by command line arguments (see *Command Line Interface* below). By default, the dimensions of the universe will be 16×16 (16 rows and 16 columns). Each cell can be referred to by it's row and column, starting in the bottom left with 0 for each dimension. An example of a 5×5 Life universe is illustrated in Figure 1.

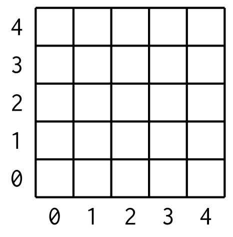

Figure 1: A 5×5 *Life* universe. Row and column indices are labelled. All cells are *dead* (*inactive*)

Each cell in the grid can exist in one of two states - *dead* or *alive* (also referred to as *inactive* and *active* respectively). Each cell is aware of it's eight closest neighbours during any generation of the life universe. These neighbours are the cells positioned horizontally, vertically and diagonally adjacent of any given cell.

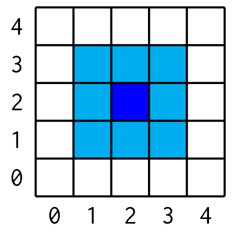

Figure 2: Visual representation of a neighbourhood in *Life*. The neighbours of the cell highlighted in blue are highlighted in light blue.

Life progresses (or evolves) through a series of generations, with each generation depending on the one that proceeded it. The rules that determine the next generation are rather simple and can be summarized by the following:

- Any live cell with exactly 2 or 3 live neighbours survive (stay alive) in the next generation.
- Any dead cell with exactly 3 live neighbours resurrect (become alive) in the next generation.
- All other cells are dead in the next generation.

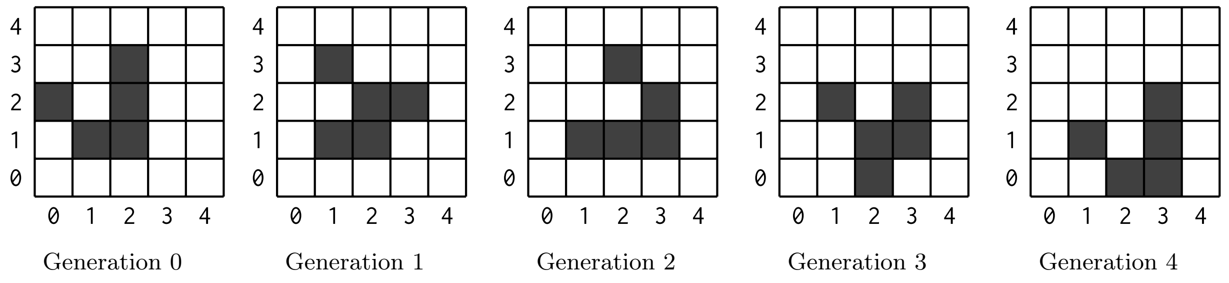

Figure 3: Generational evolution of a *glider*. Note how the pattern repeats after 4 generations of evolution.

An example of the effects of these rules is given in Figure 3, which contains five consecutive generations. To describe how one of the rules work with an example, from generations 0 to 1, cell $(3,1)$ is changed from dead to alive because it has exactly 3 live neighbors in generation 0 (cells $(2,0)$, $(2,2)$, and $(2,3)$). For this assignment, life will continue to evolve for a pre-determined amount of generations (50 by default).

Seeds

The first generation of the game is called the *seed*, which will be determined randomly, or through a formatted file. The long term behaviour of the game is determined entirely by the *seed*, meaning that the future state of any game is deterministic. This is one way you can determine whether your program is running correctly and will be the basis for grading your assignment.

Randomisation

The default method for generating a *seed* is through randomization. For each cell in the *seed*, the probability of the cell being alive is 50% and is otherwise assumed dead. The probability that any given cell is initially alive can be modified through command line arguments.

File

Alternatively, the *seed* can be defined by a file that is supplied through the command line arguments. If a file is supplied in the command line arguments then the randomization process described above will not occur. The seed files will have the extension of `.seed` and will be formatted according to the following rules:

- The first line of the file will always be `#version=1.0`.
- Each line of the file represents a cell that is initially alive (the rest are assumed dead).
- Each line will contain two integers, separated by a space character.
    - The first integer represents the row index of the live cell.
    - The second integer represents the column index of the live cell.

Below is an example of a seed file that would result in the glider pattern shown in Figure 3 (Generation 0):

```
#version=1.0
3 2
2 0
2 2
1 1
1 2
```

Because the file and the universe size are specified independently, it is possible for a file to specify alive cells that exist outside the bounds of the universe. If this occurs, the user should be presented a warning and the recommend universe size that would suit the seed file (the maximum row and column indices). See detailed requirements in *Command Line Interface* below.

Periodic Boundary Conditions

Periodic boundary conditions can be applied to a *Life* universe for some more interesting behaviour. Under periodic boundary conditions, a cell that exists on the boundary of universe will have neighbours that "wrap around" to the other side of the universe.

You will find it particularly useful to draw on your knowledge of integer arithmetic operators (such as **modulus**) when attempting to implement this feature.

This is best described visually, see Figure 4. Consider the last pair of examples in the figure. In the non-periodic case, the the neighbouring cells that would exist if the universe was any larger ($(5,3)$, $(5,4)$, $(5,5)$, $(3,5)$ and $(4,5)$) are non-existent. In the periodic case...

- The neighbouring cells that would normally be located at $(5, 3)$ and $(5, 4)$ are located at $(0, 3)$ and $(0, 4)$.
- The neighbouring cells that would normally be located at $(3, 5)$ and $(4, 5)$ are located at $(3, 0)$ and $(4, 0)$.
- The neighbouring cell that would normally be located at $(5, 5)$ is located at $(0, 0)$.

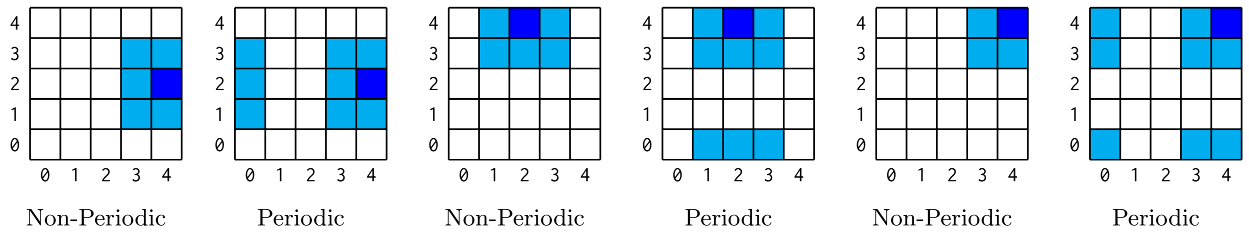

Figure 4: Examples of neighbouring cells in contrasting non-periodic and periodic boundary condition based universes. The neighbours of the cell highlighted in blue are highlighted in light blue.

Update Rates

Even in the most inefficient implementations of *Life*, the computation time for a single generational evolution can be quite fast (for small universes). As such, it is often necessary to limit the number of generational evolutions that can occur every second. This can be accomplished using a number of different different methods, two of which are briefly described below:

- (**Simple**) Add a delay between each generational evolution by sleeping the main thread. Although simple, this method can be inaccurate, particularly if your algorithm for evolving your universe is slow.
- (**Accurate**) Use a stopwatch or timer to delay the next evolution until the correct time has elapsed.

Keep in mind that the time between updates can be found with the following formula (where T is the time between updates in seconds, and f is the update rate/frequency):

$$T = \frac{1}{f}$$

Display

You will not be required to write your own display component for this program. Instead it has been provided to you in the form of a library called `Display`. It is your task to integrate this library into your program. The project library is already contained within the assignment template provided on Blackboard. To interface with the display, you will need to use the `Grid` class. The classes in the library are well documented using XML style comments, you are encouraged to read these comments to help you understand the code. Ensure that you do not modify the code in this library so that the display works as intended.

Command Line Interface

Many modern software applications are controlled via graphical user interfaces (GUI), however command line interfaces (CLI) are still frequently used by software developers and system administrators. The creation of GUIs are outside the scope of this unit (event driven programming is covered more in CAB302), so your application must be created using a CLI instead. Depending on the application, CLIs have different usages. This section will describe the general structure and usage of a CLI, and the specific usage that your program's CLI **must** follow. **Failure to follow these CLI requirements precisely will mean that your program cannot be tested correctly and will likely result in severe academic penalty**.

CLI Command Structure

A lot of single purpose CLI programs follow the following structure:

```
> executable [<arguments>]
```

The right angle bracket (>) indicates the beginning of the command line prompt, you do not type this yourself. The `executable` is the name of the program. In this assignment, the program will be called `life`. More specifically, because we are creating a .NET Core application, the `.dll` file that is built for our program can be executed on MacOS and Windows like so:

```
> dotnet life.dll [<arguments>]
```

The `[<arguments>]` are a series of **optional** arguments for your program. These arguments will allow the program to run with a variety of different settings that differ from the defaults. Each of the option type arguments are described below.

Rows and Columns

The number of rows and columns of the game board may be specified using the `–dimensions` option. The flag must be followed by two parameters, the first specifying the number of rows and the second specifying the number of columns.

Defaults: The game board will have 16 rows and 16 columns.

Validation: Both values must be provided and must be positive integers between 4 and 48 (inclusive).

Usage: `--dimensions <rows> <columns>`

Periodic Behaviour

Whether the game board acts in a *periodic manner* (see **Behaviour**) may be specified using the `--periodic` option. This flag should not be followed by any parameters.

Defaults: The game board will **not** behave in a periodic manner.

Validation: N/A

Usage: `--periodic`

Random Factor

The probability of any cell to be initially active may be specified using the `--random` option. This flag should be followed by a single parameter.

Defaults: The probability will be 0.5 (50%).

Validation: The value must be a float between 0 and 1 (inclusive).

Usage: `--random <probability>`

Input File

The path of the file representing the initial game state may be specified using the `--seed` options. This flag should be followed by a single parameter.

Defaults: No input file is used.

Validation: The value must be a valid absolute or relative file path. The value must be a valid file path have the `.seed` file extension.

Usage: `--seed <filename>`

Generations

The number of generations in the game may be specified using the `--generations` option. This flag should be followed by a single parameter.

Defaults: The game should run for 50 generations.

Validation: The value must be a positive non-zero integer.

Usage: `--generations <number>`

Maximum Update Rate

The maximum number of generational updates per second (ups) may be specified using the `--max-update` option. This flag should be followed by a single parameter.

Defaults: The maximum update rate is 5 updates per second.

Validation: The value must be a float between 1 and 30 (inclusive).

Usage: `--max-update <ups>`

Step Mode

Whether the program will wait for the user to press the space-bar key to progress to the next generation, may be specified using the `--step` option.

Defaults: The program will **not** run in step mode

Validation: N/A

Usage: `--step`

Validation

Each of the command line arguments that can be provided to the program have the potential to modify the behaviour of the program. The program must check to see if each of the arguments are valid in accordance with the descriptions above. If the arguments for a particular category happen to be invalid, your program should revert to the defaults for that category and continue to run your program. The user should be issued a warning with a brief description of the problem.

Test Cases

To ensure your program is working as intended you should test it thoroughly. Appendix A contains a series of test cases for your program. It is imperative that you test your program using these test cases before submission as your program will be marked using similar test cases. However, that does not mean you should not limit yourself to the test cases presented here.

Each test case has a link to a video that shows the intended behaviour of each test case. Note that for test cases that involve a random seed, you will not be able to exactly replicate the video demonstration, but it will give you an expectation on the type of behaviour.

Program Flow

Your program should follow the following program flow:

1. The program displays the success or failure of interpreting the command line arguments in the console window.
2. The program displays the runtime settings based on a combination of command line arguments and defaults.
3. The program waits until the user presses the space-bar key to begin.
4. The *Life* simulation is run using the runtime settings. If step mode is active, the program must wait until the user presses the space-bar key before showing the next generation.
5. Once the simulation is complete, the program must wait until the user presses the space-bar key to clear the screen and end the program.

Deliverables

For this assignment, You are required to submit a **zip** file – via Blackboard – containing the following items:

- The solution folder (containing your whole solution and it's projects).
- A brief user manual (in the form of a `README.md`).
- A self-filled CRA and statement of completeness.

A submission template has been provided on Blackboard that you should download and use for your submission.

Do not submit just the `.cs` files. Do not submit just the `.sln` file. You must submit the folder which contains the `.sln` file and all folders and files contained within.

If you are unsure how to submit your assignment correctly, please seek advice from your tutor. Be sure test by zipping your folder, moving it to a lab computer or virtual machine, and unzipping. Ensure that you can open your solution via the `.sln` file and that your solution builds and runs as expected.

Below is a description of each of the non-code items required for submission. You will be awarded points for completing each of the following items in full, make sure you complete and submit both to maximize your grade.

1. **User Manual**: Briefly explain how to build and run the *Release* version of your program using Visual Studio. Describe the location of your executable and how it should be run with command line arguments (adhering to the command line argument guidelines detailed in this specification). This needs to be formatted using Markdown formatting which you can read more about here.
2. **CRA and Statement of Completeness**: This document acts as a declaration of your work and progress on the assignment. The process of filling out this document will ensure that you full understand the task and its markable requirements. You must fill this document honestly (don't give yourself full marks unless you really believe you have aced the assignment), it exists purely for your own benefit.

Getting Started

Tackling a large project such as this assignment may initially seem daunting. However, breaking it down into smaller tasks that can be implemented and tested will make the task much more manageable. This is a crucial skill for professional practice.

This assignment specification is being released before you get to any OOP related topics. As such, you may find it difficult to start implementing any classes for a couple of weeks. As you have been exposed to a significant amount of I/O, begin with the design and functionality of the command line arguments, and move on from there as you are introduced to new topics.

Make sure that you keep the CAB201 C# Programming Style Guide document on Blackboard handy while completing this assignment. It is important that your maintain well documented and formatted code from the very beginning.

This specification will be uploaded alongside a video demonstration of the completed program. You should watch this video before getting started to help you understand how your program should function or how *Life* works.

Submission Instructions

Please follow the instruction details detailed below:

1. Test your program on the practical lab computers or on the QUT VMs to ensure that your code works as intended on a standard machine. **Code that does not compile or run will not receive any marks.**

2. Arrange your deliverables in a zip folder labelled `CAB201_2020S2_ProjectPartA_nXXXXXXXX` where the last part is replaced with your student ID. Your archive must have a `.zip` extension.

3. Submit a preliminary version of your assignment at least once before the progress submission due date. This *progress submission* does not need to be a completed version of your assignment, but you must have made some quantifiable progress. **There are grades allocated for the progress submission.**

4. Submit the final version of your assignment before the final submission due date. The code in your final submission will be the only thing marked directly.

Academic Integrity

All submissions for this assignment will be analysed by the MoSS (Measure of Software Similarity) plagiarism detection system (`http://theory.stanford.edu/~aiken/moss/`). Serious violations of the university's policies regarding plagiarism will be forwarded to the SEF's Academic Misconduct Committee for formal prosecution. As per QUT rules, you are not permitted to copy or share solutions to individual assessment items. In serious plagiarism cases SEF's Academic Misconduct Committee prosecutes both the copier and the original author equally.

It is your responsibility to keep your solution secure. In particular, **you must not make your solution visible online via cloud-based code development platforms such as GitHub**. If you wish to use such a resource, do so only if you are certain you have a private repository that cannot be seen by anyone else. However, it is recommended to keep your work well away from both the Internet and your fellow students to avoid being prosecuted for plagiarism.

Please note that using external libraries for this project will result in academic penalty, you must only use code within the standard .NET API (3.1). You may use any class in the `System` namespace (do not use the `Microsoft` namespace).

A Test Cases

Default

Video Demonstration: https://youtu.be/HB-RHtiv4jI

```
>dotnet life.dll
```

Step Mode

Video Demonstration: https://youtu.be/4750fjO2GJI

```
>dotnet life.dll --step
```

Custom Generations

Video Demonstration: https://youtu.be/tHeo5S5EwQE

```
>dotnet life.dll --generations 10
```

Custom Random Factor (10%)

Video Demonstration: https://youtu.be/0zNceppeEDI

```
dotnet life.dll --random 0.1 --step
```

Custom Random Factor (25%)

Video Demonstration: https://youtu.be/aXZOcAsSZzM

```
dotnet life.dll --random 0.25 --step
```

Custom Random Factor (75%)

Video Demonstration: https://youtu.be/-CCE2lLwTc4

```
dotnet life.dll --random 0.75 --step
```

Custom Random Factor (90%)

Video Demonstration: https://youtu.be/SGR8gdl7TZs

```
dotnet life.dll --random 0.9 --step
```

Custom Dimensions

Video Demonstration: https://youtu.be/9m-b42aZ9uw

```
>dotnet life.dll --dimensions 32 48
```

Custom Update Rate

Video Demonstration: https://youtu.be/Ufi_4xzdO1s

```
>dotnet life.dll --max-update 15
```

Periodic Mode

Video Demonstration: https://youtu.be/47WMN8apf6Q

```
>dotnet life.dll --periodic
```

Seed File (Glider)

Video Demonstration: https://youtu.be/tQzwp6ycp28

```
>dotnet life.dll --seed path/to/glider.seed
```

Glider, Small Dimensions

Video Demonstration: https://youtu.be/Oq5LNPoxp4Y

```
>dotnet life.dll --dimensions 8 8 --seed path/to/glider.seed
```

Glider, Small Dimensions, Periodic

Video Demonstration: https://youtu.be/gHBv9BDHiBI

```
>dotnet life.dll --periodic --dimensions 8 8 --seed path/to/glider.seed
```

Glider, Wide

Video Demonstration: https://youtu.be/_dYjjB7v7SA

```
>dotnet life.dll --dimensions 8 32 --seed path\to\glider.seed --periodic --generations 100
```

Glider, Tall

Video Demonstration: https://youtu.be/cekfvqH55k4

```
>dotnet life.dll --dimensions 32 8 --seed path\to\glider.seed --periodic --generations 100
```

CAB201 Seed

Video Demonstration: https://youtu.be/llFu15a_kRM

```
>dotnet life.dll --dimensions 15 32 --seed path\to\cab201.seed
```

Worker Bee Oscillator

Video Demonstration: https://youtu.be/40394Fyyyb4

```
>dotnet life.dll --dimensions 15 20 --seed path\to\workerbee.seed
```

Pulsar Oscillator

Video Demonstration: https://youtu.be/3E1-GDr_w0c

```
>dotnet life.dll --dimensions 19 19 --seed path\to\pulsar.seed
```

Octagon 2 Oscillator

Video Demonstration: https://youtu.be/vDZCRfn0fx8

```
>dotnet life.dll --dimensions 12 12 --seed path\to\octagon2.seed
```

Figure Eight Oscillator

Video Demonstration: https://youtu.be/kOoJo17b6HU

```
>dotnet life.dll --dimensions 14 14 --seed path\to\figureEight.seed
```