

# Week 2 Worksheet: Rule Based Systems

## Overview

The lab Objectives are:

1. Play and understand a RBS Example
2. Finish and extend the ruleset for the Game of Life
3. Explore and implement rule-based AI for a racing game
4. Bonus: Explore the BOIDS swarming algorithm

## Software

This tutorial work will require the use of a C++ IDE. Visual Studio is the recommended software to use, although any C++ development environment should be sufficient.

One of the tasks will require Unity. This may require you to create a Unity Account.

Visual Studio and Unity should be available on the lab machines.

There is also AppsAnywhere, that allows you access to some of the software we use. You can use this to install some of the software in the labs - <https://myapps.abertay.ac.uk/> .

# Game of Life

## Getting Started

Go to <https://bitstorm.org/gameoflife/> and experiment with the ‘Game of Life’.

The site gives you a window to play the game and an explanation of the rules. To summarise:

**For a space that is 'populated':**

Each cell with one or no neighbours dies, as if by solitude.

Each cell with four or more neighbours dies, as if by overpopulation.

Each cell with two or three neighbours survives.

**For a space that is 'empty' or 'unpopulated'**

Each cell with three neighbours becomes populated.

From the Week 2 content on MLS, download ‘**Rule Based System - C++**’ from the “Week 2 Lab Resources” section. Extract the contents of the zip file. Open the Visual Studio Project.

## Finish the Ruleset

The project contains a version of the ‘*Game of Life’* built using SFML, but the currently implemented rules are different.

**For a space that is 'populated':**

Each cell has a chance to move to an unoccupied adjacent plot (cannot move off map)

Each cell with more than one neighbour dies

**For a space that is 'empty' or 'unpopulated'**

Each cell with more than one neighbour becomes populated

Unlike the regular version of the *Game of Life*, there is no death by loneliness rule.

1. Open the project in Visual Studio and compile the program
2. Play the game and experiment with shapes
3. Edit the function on line 77:

Currently it is very crudely done, if a grid space is occupied and it has more than one neighbour, then it becomes unoccupied

If it is unoccupied and has more than one neighbour, then it becomes populated

Edit this code so it matches with the regular version of *Game of Life.*

1. Add the death by loneliness rule

Occupied nodes with no neighbours becomes unoccupied

## Extend the Ruleset

Extend the ruleset to accommodate the following rules:

1. Random spawning once every five cycles:

Find an unoccupied node in the grid and populate it

1. Random death every six cycles

Every six cycles of the game, find a populate node and kill it

1. When the number of occupied nodes is over 75% the size of the grid, kill half of all occupied nodes

You can add your own rules or edit the ones you currently have as you see fit to see the impact on behaviour.

# Racing Game AI

*Heat: Pedal to the Metal* is a Board Game for 1-6 players published in 2022 by [Days of Wonder](https://www.daysofwonder.com/heat/).

*“Based on simple and intuitive hand management, Heat: Pedal to the Metal puts players in the driver's seat of intense car races, jockeying for position to cross the finish line first”*

Automated Drivers, called Legends, can be added to the game. The game instructions outline the rules to create the desired behaviour from these automated drivers.

We will implement a version of this AI behaviour in a simplified game environment built in Unity.

## Getting Started

From the Week 2 content on MLS, download ‘**Rule Based System - Unity**’ from the “Week 2 Lab Resources” section. Extract the contents of the zip file. Open in Unity.

If you received any versioning messages, update the project to the latest installed version.

* Press Play to run the scene.

You will see a racetrack, which comprises a series of straights and corners, split into sections (Spaces) which the Legend will move along.

Around the racetrack are several symbols. Each corner has a recommended speed, for example Corner with speed ‘6’.

A video game with a number

Description automatically generated

**Corder with Speed 6**

Before each corner is a Legends Line. These are used by the Legends, our automated driver (more details below).

A screen shot of a video game

Description automatically generated

**The Legend Line**

Our Legend will start at the Starting Line.

A red and white striped awning

Description automatically generated

**The Starting Line**

Each round a “*Legends Card*” is revealed which will also determine the behaviour of the Legend. This contains two relevant values – Diamond and Speed. These are shown in the top right panel, along with some other information and some control buttons.

A screenshot of a computer

Description automatically generated

**The Database & Controls Display**

We will use the ‘Red’ colour for our Legend.

* Press Race and observe the car movement.

## Desired Behaviour

The current behaviour is temporary, and we need to update it.

We will use a Rule Based approach to create the desired behaviour.

### Rule 1: Clearing Corners

The first rule we need to implement is:

1. ***“If a Legend has crossed the Legends Line (its car is between the Legends Line and the next Corner Line), move the car forward as many Spaces as the speed limit of the upcoming Corner plus the Diamond number for the Legend Colour.”***

* From within the Unity project, open the C# script *RuleBasedSystemAI* from within the Scripts folder.
  + All the elements we need to configure are in this class
  + We will define our rules in the *initRules*() function

We will implement our rules using two parts - the **Condition** and the **Action**. See this week’s lecture slides if you need a refresher.

The initRules() function already contains code for creating the rules. Rule 1’s code begins on Line 48. The code uses generic functions, similar to C++ template functions, and we will store them in a similar way to the example from this week’s lecture.

First, we define the Condition. This checks to see if certain criteria have been met, and if so, the Rule is triggered. Our Rule 1 should be triggered if a Legend has crossed the Legends Line before the next corner.

The below condition check will return true if this is the case, otherwise false.

A computer screen with text

Description automatically generated

Secondly, we define the Action. We need to return the distance the car should move.

A screen shot of a computer

Description automatically generated

Currently, this always returns ‘1’.

* Update the code to match the desired behaviour from Rule 1 above
  + The *Database* object will contain the information you will need to calculate the appropriate return value for the Action
  + The *selectedLegendColour* variable contains information about the colour of the Legend
  + The function *getDiamondForColour*() within the database’s *currentLegendCard* will return the Diamond value for a given Legend Colour

The created Rule1 is already added to the *allRules* object through the below code. The condition and action parts are added together, associating the condition of the Rule with the action if the rule is fired.



The *CheckRules*() function in *RuleBasedSystemAI* is already setup to handle checking and firing the rules from within *allRules*.

* Run the program and check the desired movement behaviour is occurring when the Legend is between a Legend Line and a Corner

### Rules 2&3: Approaching Corners

The next behaviour we need to implement is when the Legend is approaching a corner. Here, the desired behaviour is:

*“If a Legend has not crossed the Legends Line, it will try to get as close to the next Corner Line as possible, without going past it. There are two possible outcomes:*

1. ***If the Legend can move at the Speed number for the Legend Colour without crossing the next Corner Line, it will do so***
2. ***If moving at that Speed would cause it to cross the next Corner Line, instead move the Legend forward the Diamond number before the Corner”***

The current code for Rules 2 and 3 has two issues.

First, the Conditions are incomplete.

Second, the Action simply returns ‘3’ in both cases.

* Update both the Condition and Action components so the behaviour matches Rules 2 & 3 above.
* Run the program and check to see if the desired behaviour is occurring for both approaching and clearing corners.

### Additional Rules

There is one more Rule we need to implement.

1. **Legends may not cross 2 Corner Lines in the same move. If this is about to happen, place the Legend car on the first free Space before the second Corner Line**

* Add additional Rules to implement this behaviour
  + It is important to remember that several Rules may be eligible to fire, however only one should be selected
  + The current code for *CheckRules*() is limited in that it will fire the first rule it finds whose conditions are met
    - You will want to modify this to check for all possible Rules that can be triggered, before selecting a single one to fire using some method, such as a priority system
      * See the Arbiter content from this week’s lecture
* You can copy the format of the current rules for the new rules
  + Remember to ensure each Rule is added to the ‘*allRules’* object in the *RuleBasedSystemAI* class
* You can modify the *ruleData* stuct with additional information, such as priority
* Once more, run the program and see check that the desired behaviour is occurring for both approaching and clearing corners, and that all other rules are satisfied.

# Open Ended Bonus Work

## Boids

A flock of birds flying in the sky

Description automatically generated

[Boids](https://en.wikipedia.org/wiki/Boids) is an RBS that controls the movement of a swarm/crowd/group. Classically the flock does the following:

* Alignment – Unit in the group will align itself with the average of the rest of the flock
* Cohesion – Unit will steer towards the average position of the rest of the flock (centre of mass)
* Separation – Units will maintain a distance between other flock members

Design what the database will contain and the rules the system would contain. Consider how each member of a flock would interact with another, what data they will require, what necessary data needs saved etc.

Write a section of pseudocode to explain how the program would run (you don’t have to worry about specifics of drawing or rendering). Explain how the data is written to memory, how the flock access the data, how the RBS system reads and executes the data and its rules.

Image Credit: [Pixabay](https://pixabay.com/photos/emotions-nature-sunrise-bird-flight-3406667/)

## Racing Game AI Difficulty

Legends can be made tougher opponents by increasing the Speed values on cards by 1 or 2. Implement a difficulty system that incorporates these rules.

An efficient method for representing and checking our rules is important as more rules are added, or additional complexity is required. Review the content in Chapter 5.8.7 of ‘*Games for AI’ (Millington)* (available online via Abertay Library) for approaches for this.