Atomination

Welcome to **Object-Oriented-Games (OOG)!** As the new programmer here, you will be tasked with creating a demo for the game called **Atomination**. You will write a game called **Atomination**. You will be tasked with writing this game using the Java programming, utilising everything you have learned over the semester.

This game revolves around placing atoms on a grid-based game board. Each grid space has a limit of how many atoms it can contain. Once the limit is reached, the atoms will expand to the adjacent grid spaces. This simple rule exhibits an interesting property where chains can be triggered by a single placement. This acts as a mechanism to capture other grid spaces from your opponents.

The game will be accompanied by a number of utility functions for players to utilise such as saving the game, game statistics and loading games. You will also need to implement a set of commands that will allow the players to interact with the game.

Implementation Details

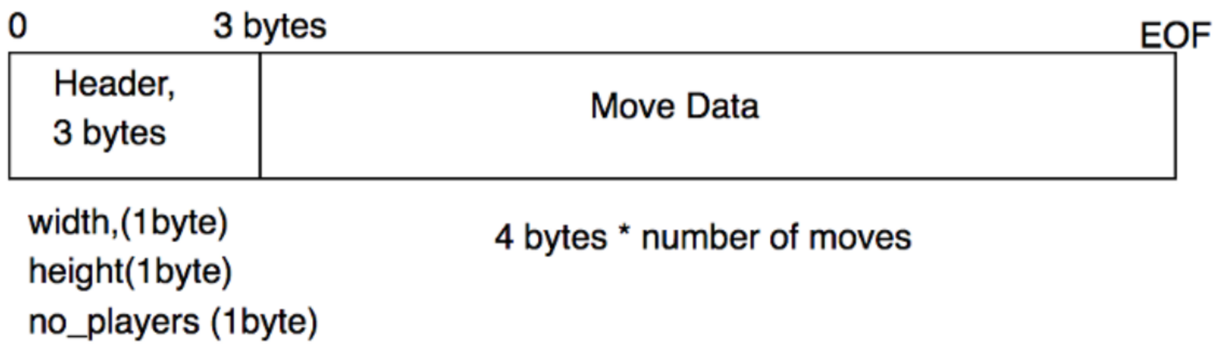
Write a program in Java that implements the game atoms. The sample test cases on ed will only contain valid input command, it is up to you to handle any invalid input. Commands are case sensitive.

Game rules and features:

- There is a game board that is **n x m** grid spaces
- There are k players and each player takes a turn starting from player 1. there is a minimum of 2 players and a maximum of 4 players per a game
- Initially a grid space is unoccupied until a player places an atom in that grid space
- A player can place an atom on a grid space that they already own or is unoccupied
    - If the grid space is a corner, then the limit is 2
    - If the grid space is on a side then the limit is 3
    - If the grid space is neither a corner or a side space then the limit is 4 (The pattern is the adjacent grid spaces)
- Once a grid space has reached it's limit (`number of atoms >= limit`), a single atom will expand out to the adjacent grid spaces, capturing them if they are owned by an opponent
- After the first k moves (k being the number of players in the game), players can be removed from the game if they no longer own any grid spaces
- Players can undo moves that they have performed
- The maximum width of the board is 255, the minimum width is 2
- The maximum height of the board is 255, the minimum height is 2
- You may assume the maximum line length is 255
- There is an option to save the game and load it when the program has been reloaded
- The player colour order is **RGPB (Red, Green, Purple, Blue)**
- The user interacts with a set of commands that are specified later in this document

Save file specification

The file header contains: 1 byte for width 1 byte for height 1 byte for player After the header, the move data will fill the rest of the file until the end of the file.



Each move is encoded into **4 bytes**, the first **2 bytes** are used for coordinates, storing x and y respectively while the remaining **2 bytes** is empty padding, simply containing a value of 0.

You have been provided a scaffold to help you get started with your application.

Player.java

```
public class Player {
    private int gridsOwned;
}
```

Grid.java

```
public class Grid {
    private Player owner;
    private int atomCount;
}
```

Part 1 - Game Logic

Help

Provides a list of commands, elaborating on each functionality.

```
HELP    displays this help message
QUIT    quits the current game
DISPLAY draws the game board in terminal

START   <number of players> <width> <height> starts the game
PLACE   <x> <y> places an atom in a grid space
UNDO    undoes the last move made
STAT    displays game statistics
SAVE    <filename> saves the state of the game
LOAD    <filename> loads a save file
```

START <number_of_players>

The start command will accept 3 arguments, the first two arguments are the dimensions and of the game board, the last argument specifies the number of players in the game.

If any of the input is invalid (width or height or n) are negative digits or character values, the application must respond with:

```
Invalid command arguments
```

If the argument length is less than the required number (3) the program should respond with:

```
Missing Argument
```

The number of arguments is greater than the required number the program should respond with:

```
Too Many Arguments
```

The **START** command cannot be used after the **START** command has been successfully executed. If this case is presented the program should respond with:

```
Invalid Command
```

When **START** command is successfully executed the program should respond with, the colours are assumed to be in this order: RGPB (Red, Green, Purple, Blue).

```
Game Ready
Red’s Turn
```

STAT

Displays the current state of the game

STAT cannot be executed unless a game has been started, if it is executed in this state the program should respond with

```
Game Not In Progress
```

When executed successfully the STAT command should display the current statistics related to the players.

```
Player Red:
Grid Count: 5

Player Green:
Grid Count: 2
```

In the account where a player has given up or has been removed from the game the STAT command should show. (In this example Player P gave up).

```
Player Red
Grid Count: 5

Player Green:
Grid Count: 3

Player Purple:
Lost
```

When printing out the the players, it will be from first index to last. The player colours are in the order: **RGPB** (Red, Green, Purple, Blue).

PLACE

Places an atom at an x, y position on the game board that is associated with the player's turn

If the argument length does not equal two then the program should output Invalid Coordinates

If the x value is less than 0 or greater than or equal to the width and/or If the y value is less than 0 or greater than or equal to the height then the program should output:

```
Invalid Coordinates
```

If the grid space selected is not owned by the player or is not unoccupied:

```
Cannot Place Atom Here
```

In the case that PLACE command has not be executed successfully, the program should be ready for another command:

```
PLACE -1 -1
Invalid Coordinates

PLACE 0 0
Red’s Turn
```

On success, the program should respond with whose move is next:

```
PLACE 2 1
Red’s Turn
```

In relation to expansion (when the grid space has reached its limit), the expansions move clockwise starting with the grid space that is y-1.

After a PLACE command is entered and only one player is remaining after it successfully executes, the program should respond with and quit:

```
PLACE 5 4
Red Wins
<program ends>
```

DISPLAY

Displays the gameboard using + to denote the corner and two – to denote the colour and the number of atoms located.

Example Output

```
+--------------------+
|R1|  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |G1|G3|  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |R1|  |  |  |
+--------------------+
```

Empty spaces denote unknown grid spaces while (R1) denote the owner and the number of atoms in that grid space.

PLACE examples

```
PLACE 0 0
Green’s Turn

DISPLAY
+--------------------+
|R1|  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
+--------------------+

PLACE 2 0
Red’s Turn

DISPLAY
+--------------------+
|R1|  |G1|  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
+--------------------+

PLACE 0 0
Green’s Turn
```

```
DISPLAY
+--------------------+
|  |R1|G1|  |  |  |  |
|R1|  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
+--------------------+

QUIT
Bye!
PLACE 0 0
Green’s Turn

DISPLAY
+--------------------+
|R1|  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
+--------------------+

PLACE 2 0
Red’s Turn

DISPLAY
+--------------------+
|R1|  |G1|  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
+--------------------+

PLACE 0 0
Green’s Turn

DISPLAY
+--------------------+
|  |R1|G1|  |  |  |  |
|R1|  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
+--------------------+

PLACE 2 0
Red’s Turn

DISPLAY
+--------------------+
|  |R1|G2|  |  |  |  |
|R1|  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
+--------------------+

PLACE 1 0
Green’s Turn

DISPLAY
+--------------------+
|  |R2|G2|  |  |  |  |
|R1|  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
+--------------------+

PLACE 2 0
Red’s Turn

DISPLAY
+--------------------+
|G1|  |G1|G1|  |  |  |
|R1|G1|G1|  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
+--------------------+

PLACE 0 1
Red’s Turn

DISPLAY
+--------------------+
|G1|  |G1|G1|  |  |  |
|R2|G1|G1|  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
+--------------------+

PLACE 0 0
Green Wins

QUIT
Bye!
```

UNDO

Once a player has made a move, it is up to the will of the next player to let the other player undo it. We are to pretend that this game involves hotseating and therefore it is agreed upon the players if they are to allow an undo.

```
DISPLAY
+--------------------+
|  |  |  |  |  |  |  |
|  |  |  |P1|  |  |  |
|  |G1|  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |R1|  |  |  |
+--------------------+

PLACE 2 1
Red’s Turn

+--------------------+
|  |  |  |  |  |  |  |
|  |  |B1|P1|  |  |  |
|  |G1|  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |R1|  |  |  |
+--------------------+

UNDO
Blue’s Turn

+--------------------+
|  |  |  |  |  |  |  |
|  |  |  |P1|  |  |  |
|  |G1|  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |R1|  |  |  |
+--------------------+
```

In the case that we perform UNDO at start of the game, the program should output:

```
Cannot Undo
```

QUIT

Quits the game, if the game has started it will quit the game early and not declare a winner.

```
QUIT
Bye!
```

SAVE

Save the current state of the game with a filename.

You will need to adhere to the save file structure previous outliend. The file is saved as a binary file, which contains game information stored in header of the file and player moves afterwards.

After the header has been read, the rest of the file can be assumed to contain only player move data, each move is encoded as a 32bit unsigned integer (4 bytes). Each 32bit integer can be extracted as *x* (1 byte), *y* (1 byte) *coordinates* with the additional 2 bytes acting as 0 padding afterwards.

When creating a save file, your program does not need to include any move that has been undone, only the moves that have result in the current state of the board.

This command creates a file and saves the current game details and move set, if a file already exists with this name the method should not attempt to save the file and output:

```
File Already Exists
```

On success, the program must output

```
Game Saved
```

LOAD

Load command takes a filename as an argument, if that file does not exist the program should remain in its original state and output.

```
Cannot Load Save
```

If a game has already started or load has been executed successfully, subsequently load commands should not attempt to load a new game and instead the program should respond with:

```
Restart Application To Load Save
```

If the command is executed successfully the program should respond with:

```
Game Loaded
```

Part 2 - Test

You will need to write your own test cases to ensure that your program covers as many cases as possible. For this assessment, you will need to create **JUnit** test cases for your application.

Test cases must be written in a **JUnit** test case file named `AtominationTest.java`. You will need to have a series of test cases that check each method and the game state. Try to develop test cases while working on **Part 1**. This will help with speeding up your development by maintaining a test suite of the requirements.

Part 3 - GUI

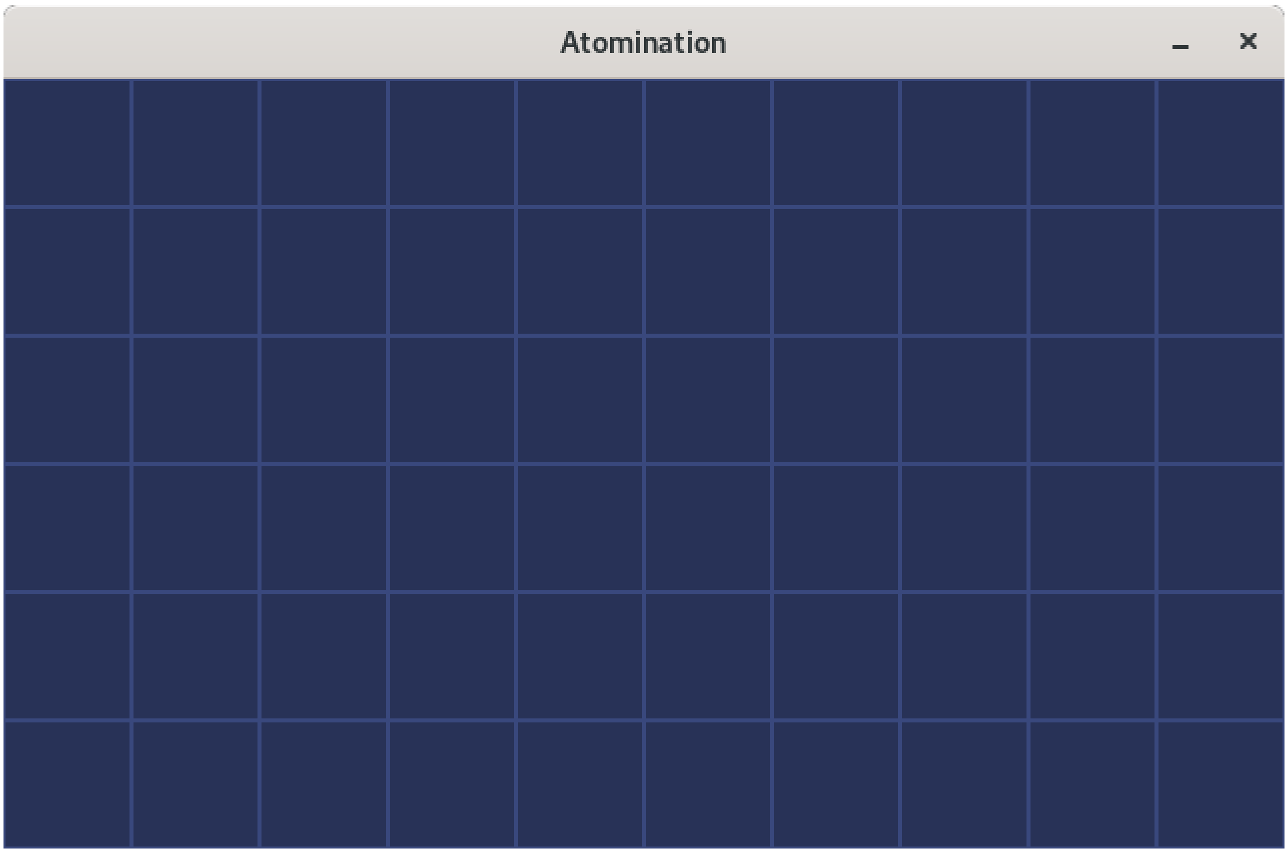
Your task is to prepare a demo for all the backers of the game **Atomination**. This part requires the following:

- Create a window of 640 by 384 (width, height)
- Render 10 x 6 grid with each grid being 64 x 64 (width, height)
- Detect a mouse click on a grid space and call the `place` method.
- Change the sprites of each grid to be based on the number of atoms exist in each grid

You have been provided a scaffold, library, documentation link and assets for constructing the GUI. Update the build.sh file to allow you to quickly build and test your program.

The library **`Processing`** contains documentation that will help with implementing each segment.

Create Window and Render Tiles (0.5 marks)



Use `tile.png` for each tile to make a grid that players can select. The grid created must also reflect the number of grid spaces that exist in the game. The demo is just a singular configuration (**10x6**).

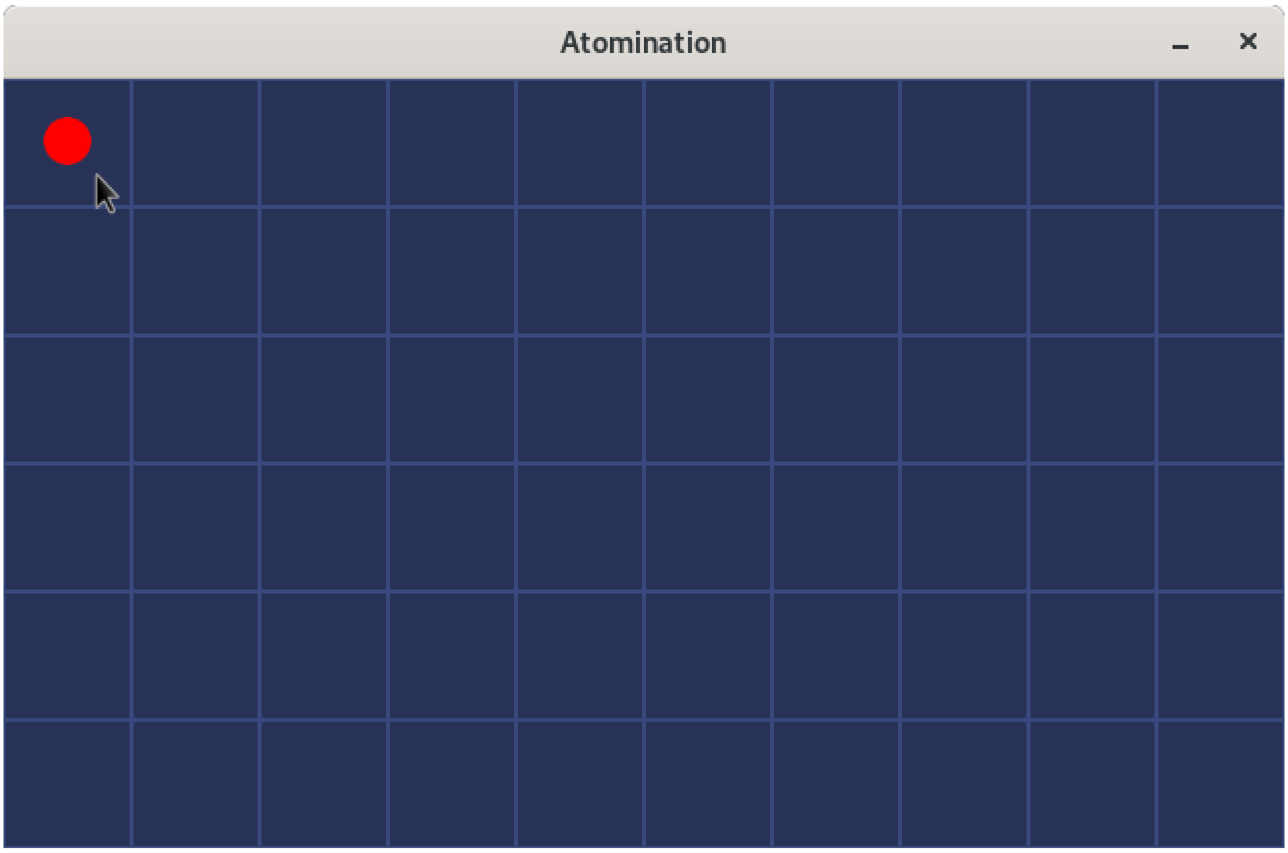
Implement the following in the `setup`, `settings` and `draw` methods. Use the following methods associated with the `AtominationGUI` scaffold class.

- size(*int width*, *int height*), This will need to be used in the `settings` method.
- loadImage(*String path*)
- image(*PImage image*, *int x*, *int y*, *int width*, *int height*)

User Interaction and play (1.5 marks)

Mouse event

You want people to play your game, this will require you implement the `mouseClicked` event in the `AtominationGUI`. Once a user has clicked on a tile, your program must respond by showing an atom placed in that tile. Use the `MouseEvent` object to retrieve the current mouse coordinates on the screen.



Since there are many tiles, your program must deduce what grid space it clicked.

Drawing

The `draw` method within the `AtominationGUI` class is called **60** times a second (as specified by the `framerate` method, use this method to continue to render the sprites on screen. You may want to clear everything before you paint the grid spaces and atoms on screen.

Each grid space maintains information in relation to how many atoms and the owner of the grid, use this information to select which sprite to draw. Use the following methods associated with the `AtominationGUI` scaffold class.

- loadImage(*String path*)
- image(*PImage image*, *int x*, *int y*, *int width*, *int height*)
- background(*int rgba*)

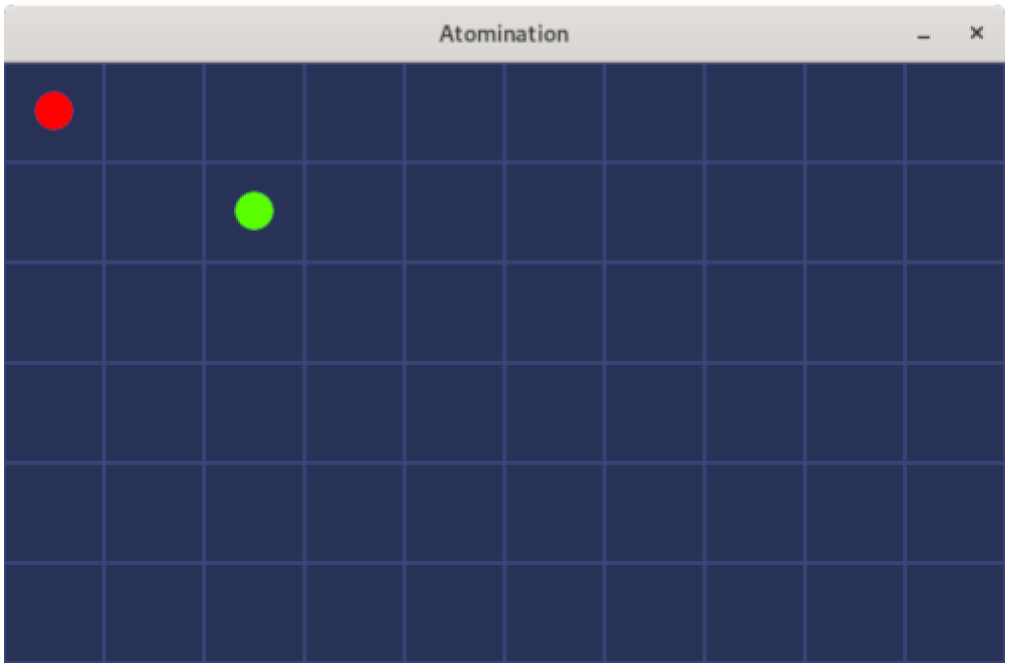
Deviation from CLI Program (1 mark)

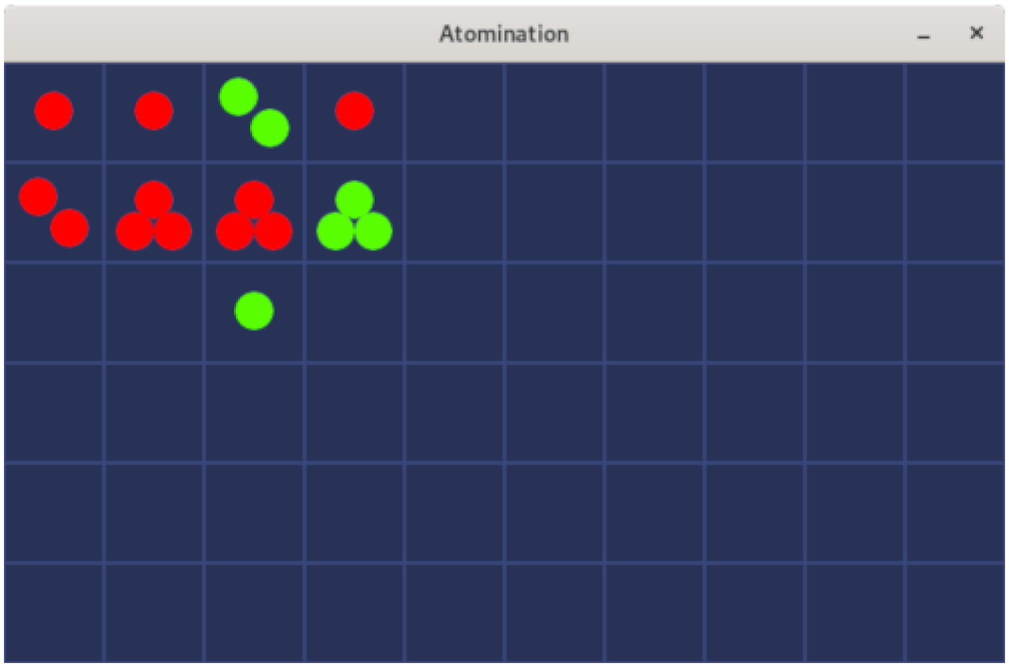
We will be checking to see if program has deviated significantly from your command line version. Your program should attempt to utilise majority of your existing code without modification.

How this game works and examples

The graphic designers have created a mock-up video, showing how the game will be played. You can access this video from the online discussion board (Ed).

Included are a couple more GUI images of how a game will play out.





Algorithms and Functions

Expansion and place

When a grid space has reached capacity, it will expand to the adjacent grid spaces. This may create a chain reaction for other grid spaces that have reached their limit. The expand function is invoked once the number of atoms has reached the limit a grid space can contain from the `place` function.

```
expand(grid, x, y):
    if (y - 1) >= 0 && (y - 1) < height:
        place(grid, x, y-1)

    if (x + 1) >= 0 && (x + 1) < width:
        place(grid, x+1, y)

    if (y + 1) >= 0 && (y + 1) < height:
        place(grid, x, y+1)

    if (x - 1) >= 0 && (x - 1) < width:
        place(grid, x-1, y)
```

2D AABB (Axis-Aligned Bounding Box)

A simple collision/box intersection detection function allows your for program to detect when two rectangular shapes have intersected. This is a general algorithm that checks for an overlap between two shapes, returning `true` if an overlap has occurred. You will need to utilise this function for **Part 3**.

```
aabbintersect(box1, box2):
    return (box1.x < (box2.x + box2.width)) and
    ((box1.x + box1.width) > box2.x) and
    (box1.y < (box2.y + box2.height)) and
    ((box1.y + box1.height) > box2.y)
```

Submission Details

You are required submit your assessment by Sunday Week 13 (11:59pm).

Your code and tests must be submitted using Ed. You can upload your files in the assessment page of the appropriate assessment. You are encouraged to submit multiple time, but only your last submission will be marked.

The submission for **PART 3** must uploaded and submitted with the rest of your code on ed. Update the `build.sh` file to assist with building your GUI program.

Marking

Your program will be marked automatically by Ed, Please ensure that you carefully follow the assignment specification. Your program must match the exact output in the examples and the test cases on Ed. Your assessment is worth a total of **12 marks**.

- 6 marks for automatic marking, these test cases will be available on Ed and will test the correctness of the CLI program.
- 3 marks for testing and manual marking, Your program will need to test the internal structure of your code and replicate some simple user input. Examples include, checking the grid array if certain grid spaces are occupied, checking grid space owners once an expansion has occurred.
- 3 marks for graphical user interface, as part of manual marking, your application will be assessed in regards to how accurately you have represented the game in a visual form and how much you have modified from the CLI variant.

Warning: Any attempts to deceive or disrupt the marking system will result in an immediate zero for the entire assignment. Negative marks can be assigned if you do not properly follow the assignment specification, or your code is unnecessarily or deliberately obfuscated.

Academic Declaration

By submitting this assignment you declare the following:

I declare that I have read and understood the University of Sydney Student Plagiarism: Coursework Policy and Procedure, and except where specifically acknowledged, the work contained in this assignment/project is my own work, and has not been copied from other sources or been previously submitted for award or assessment.

I understand that failure to comply with the Student Plagiarism: Coursework Policy and Procedure can lead to severe penalties as outlined under Chapter 8 of the University of Sydney By-Law 1999 (as amended). These penalties may be imposed in cases where any significant portion of my submitted work has been copied without proper acknowledgement from other sources, including published works, the Internet, existing programs, the work of other students, or work previously submitted for other awards or assessments.

I realise that I may be asked to identify those portions of the work contributed by me and required to demonstrate my knowledge of the relevant material by answering oral questions or by undertaking supplementary work, either written or in the laboratory, in order to arrive at the final assessment mark.

I acknowledge that the School of Computer Science, in assessing this assignment, may reproduce it entirely, may provide a copy to another member of faculty, and/or communicate a copy of this assignment to a plagiarism checking service or in-house computer program, and that a copy of the assignment may be maintained by the service or the School of Computer Science for the purpose of future plagiarism checking.