IS52038A ALGORITHMS AND DATA STRUCTURES

Assignment: Term 2 final coursework; Cheetah and Rabbit program

Designer: Andrea Fiorucci, Seyednima Jamalianardakani

Programmer: Andrea Fiorucci, Seyednima Jamalianardakani

Submission Date: [Andrea Fiorucci, Seyednima Jamalianardakani 01/03/2016]

Rabbit and Cheetah

Documentation of the implementation process.

Introduction:

According to the document brief provided by the lecturer, we had to develop and analyse a game/simulation of a classic problem called "Cheetah and Rabbit". The coursework was split into two parts where the first one involved setting up the game basic structures and movements. The second part instead, was asking to work on top of the first part by adding more advanced features and extending the first part of the project. The way we have structured our program is by dividing the product into three different main states which represent part 1 and part 2 of the coursework together. Part 1 of the project is labeled as GAME_1 in the main menu of the program and part 2 is labeled as GAME_2 and SIMULATION in the main menu of the program. There is also a CREDITS section where all our details are reported.

How to run the program:

The program is entirely made using c++ and open-framework.

In order to run the program you first need to download the open frameworks and c++ compiler using this link:

http://openframeworks.cc/download/

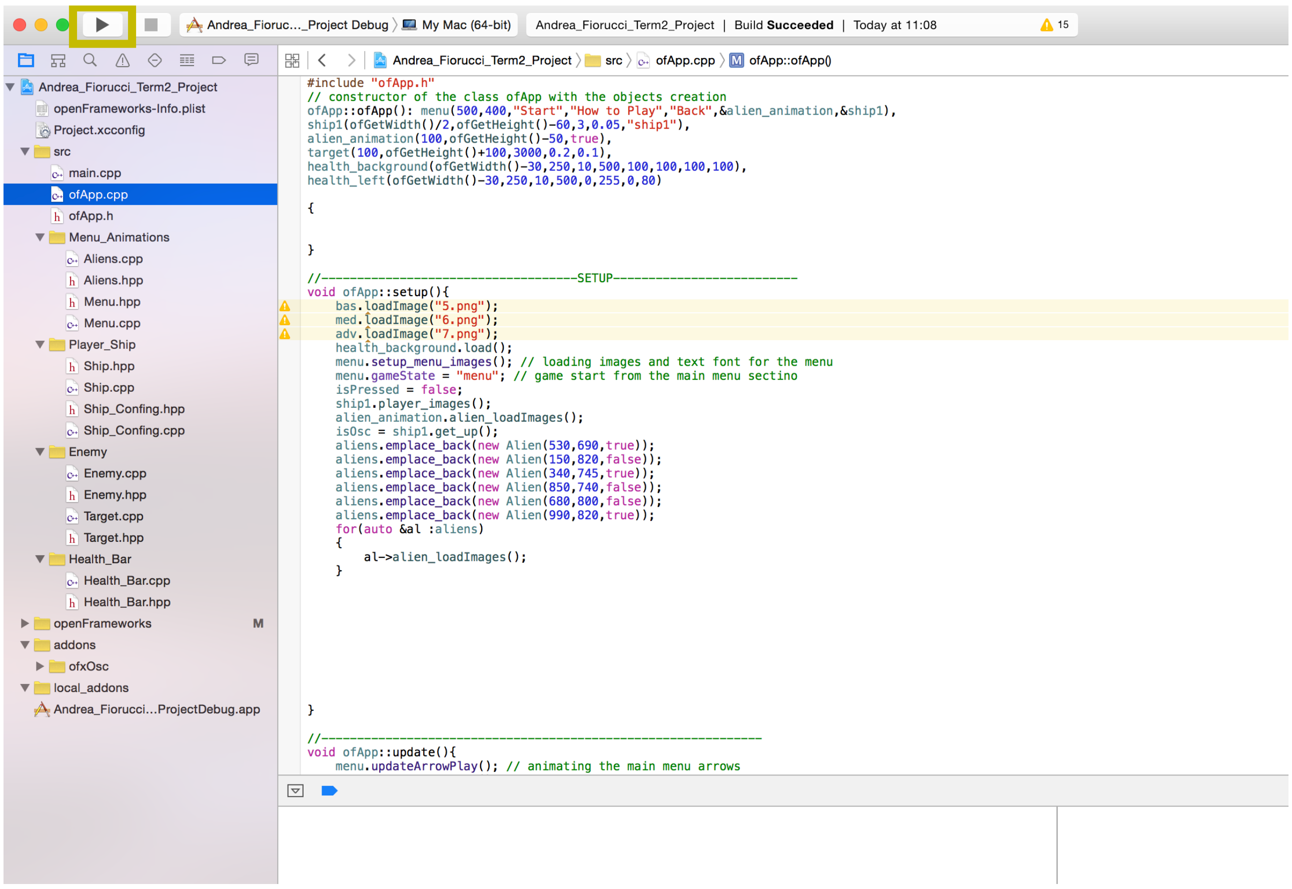
Once you have downloaded the file, you need to unzip the folder and open-frameworks is ready to be used.

Now just drag our submission folder into open-frameworks/apps/myApps.

The program is now in the right location and ready to be open.

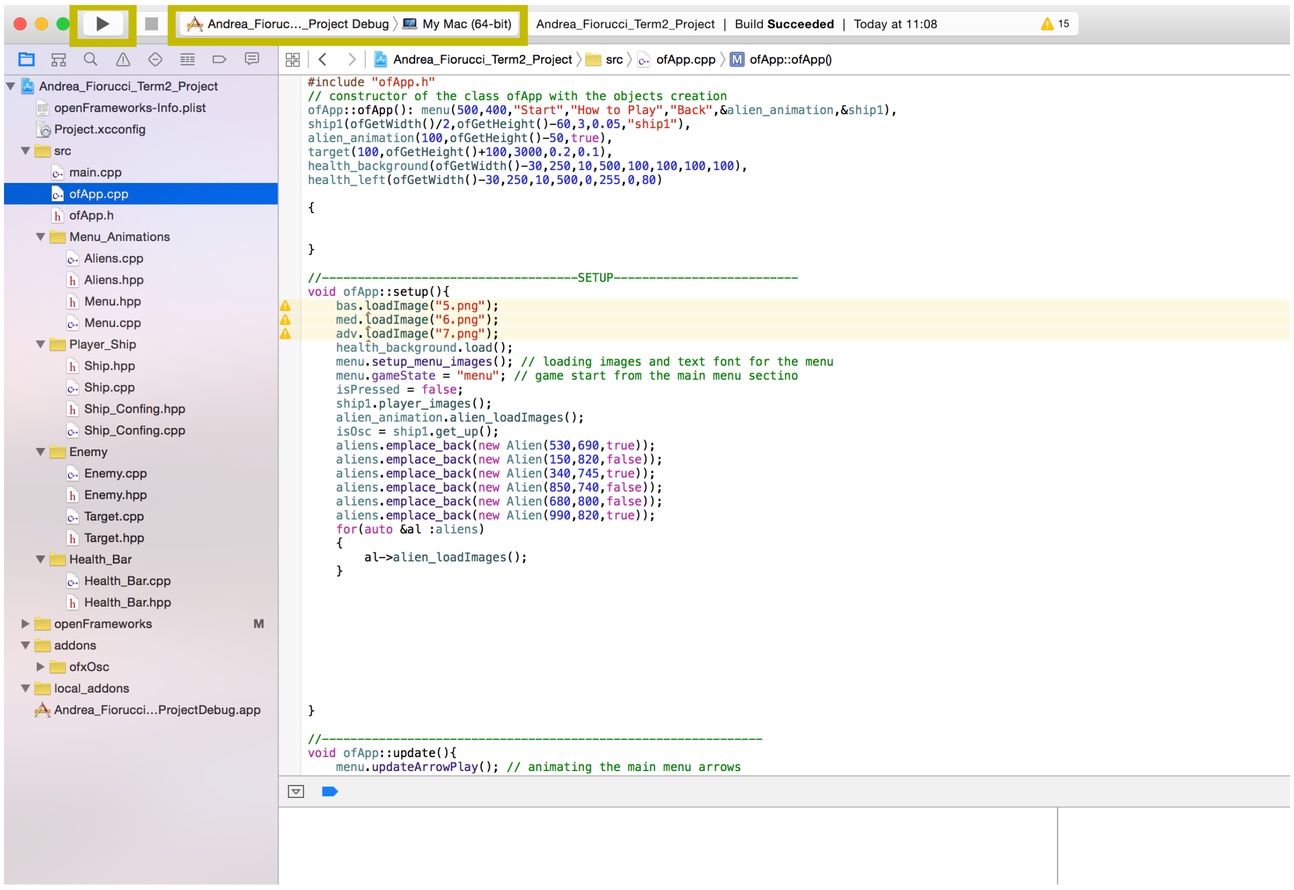
Just open this: Rabbit_Cheetah.xcodeproj

Once you open the above folder you can run the program using :



```cpp
#include "ofApp.h"
// constructor of the class ofApp with the objects creation
ofApp::ofApp(): menu(500,400,"Start","How to Play","Back",&alien_animation,&ship1),
ship1(ofGetWidth()/2,ofGetHeight()-60,3,0.05,"ship1"),
alien_animation(100,ofGetHeight()-50,true),
target(100,ofGetHeight()+100,3000,0.2,0.1),
health_background(ofGetWidth()-30,250,10,500,100,100,100,100),
health_left(ofGetWidth()-30,250,10,500,0,255,0,80)

{


}

//--------------------------------SETUP------------------------
void ofApp::setup(){
    bas.loadImage("5.png");
    med.loadImage("6.png");
    adv.loadImage("7.png");
    health_background.load();
    menu.setup_menu_images(); // loading images and text font for the menu
    menu.gameState = "menu"; // game start from the main menu sectino
    isPressed = false;
    ship1.player_images();
    alien_animation.alien_loadImages();
    isOsc = ship1.get_up();
    aliens.emplace_back(new Alien(530,690,true));
    aliens.emplace_back(new Alien(150,820,false));
    aliens.emplace_back(new Alien(340,745,true));
    aliens.emplace_back(new Alien(850,740,false));
    aliens.emplace_back(new Alien(680,800,false));
    aliens.emplace_back(new Alien(990,820,true));
    for(auto &al :aliens)
    {
        al->alien_loadImages();
    }









}

//----------------------------------------------------------------
void ofApp::update(){
    menu.updateArrowPlay(); // animating the main menu arrows
```

Make also sure that the setting are on debug mode otherwise to would get an empty window when trying to open it:

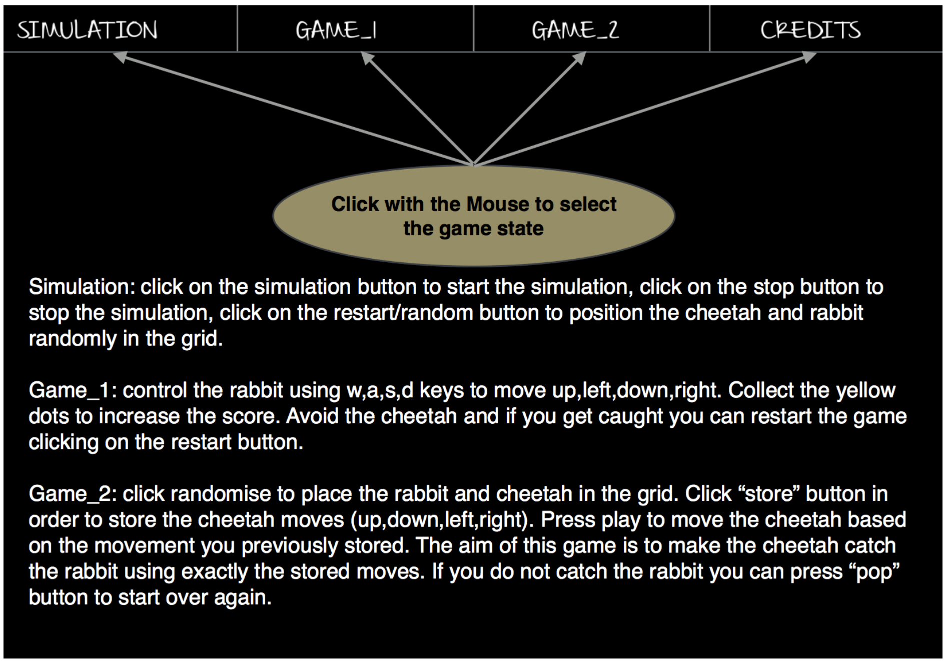


Andrea_Fioruc..._Project Debug 〉 My Mac (64-bit) | Andrea_Fiorucci_Term2_Project | Build **Succeeded** | Today at 11:08

```
#include "ofApp.h"
// constructor of the class ofApp with the objects creation
ofApp::ofApp(): menu(500,400,"Start","How to Play","Back",&alien_animation,&ship1),
ship1(ofGetWidth()/2,ofGetHeight()-60,3,0.05,"ship1"),
alien_animation(100,ofGetHeight()-50,true),
target(100,ofGetHeight()+100,3000,0.2,0.1),
health_background(ofGetWidth()-30,250,10,500,100,100,100,100),
health_left(ofGetWidth()-30,250,10,500,0,255,0,80)

{


}

//------------------------------------SETUP-------------------------
void ofApp::setup(){
    bas.loadImage("5.png");
    med.loadImage("6.png");
    adv.loadImage("7.png");
    health_background.load();
    menu.setup_menu_images(); // loading images and text font for the menu
    menu.gameState = "menu"; // game start from the main menu sectino
    isPressed = false;
    ship1.player_images();
    alien_animation.alien_loadImages();
    isOsc = ship1.get_up();
    aliens.emplace_back(new Alien(530,690,true));
    aliens.emplace_back(new Alien(150,820,false));
    aliens.emplace_back(new Alien(340,745,true));
    aliens.emplace_back(new Alien(850,740,false));
    aliens.emplace_back(new Alien(680,800,false));
    aliens.emplace_back(new Alien(990,820,true));
    for(auto &al :aliens)
    {
        al->alien_loadImages();
    }







}

//--------------------------------------------------------------
void ofApp::update(){
    menu.updateArrowPlay(); // animating the main menu arrows
```

How to navigate in the program:

As soon as you run the program you will be presented a main menu where you can use the 'UP' and 'DOWN' keys to select two different options different options(PLAY and HOW TO PLAY). Press 'ENTER' to select the section desired. Once inside the game, you can go back to the main menu using the 'BACK_SPACE' key and select different sections of the program using the mouse. There is a full explanation on how to navigate in the program in the HOW TO PLAY section as shown below:



Simulation: click on the simulation button to start the simulation, click on the stop button to stop the simulation, click on the restart/random button to position the cheetah and rabbit randomly in the grid.

Game_1: control the rabbit using w,a,s,d keys to move up,left,down,right. Collect the yellow dots to increase the score. Avoid the cheetah and if you get caught you can restart the game clicking on the restart button.

Game_2: click randomise to place the rabbit and cheetah in the grid. Click “store” button in order to store the cheetah moves (up,down,left,right). Press play to move the cheetah based on the movement you previously stored. The aim of this game is to make the cheetah catch the rabbit using exactly the stored moves. If you do not catch the rabbit you can press “pop” button to start over again.

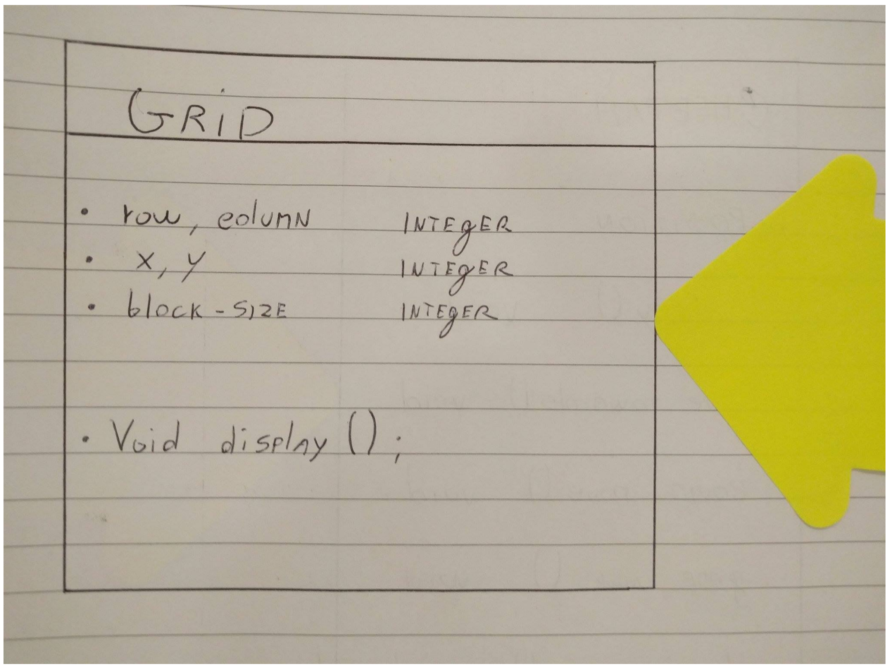
(PART 1) ————————— select GAME_1 in the upper tabs

- Game objects and design process

The main game objects we have implemented and required for this submission are the grid, the rabbit entity and the cheetah entity.

We firstly focused on finding a solution to create and display the grid. In order to achieve this first task, we have chosen a nested for loop, which draws a set of blocks creating a 14x8 matrix grid, as our most suitable option.

The simply reason why we did not use a 2d array for the implementation is simply because the nested loop structure was fitting better for our game purposes .



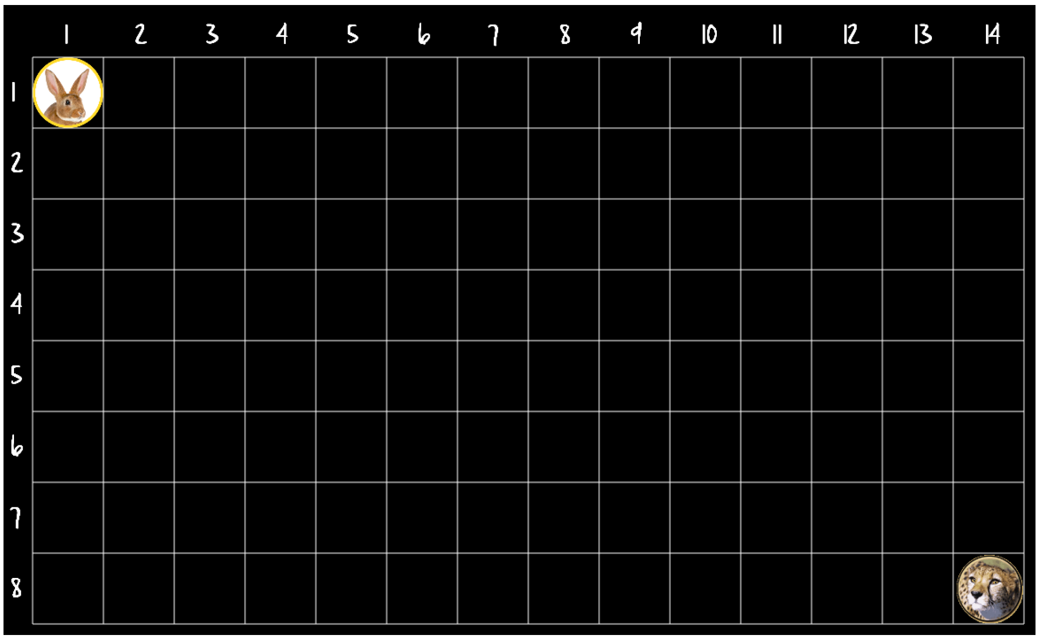
GRID

- row, column INTEGER
- x, y INTEGER
- block-SIZE INTEGER

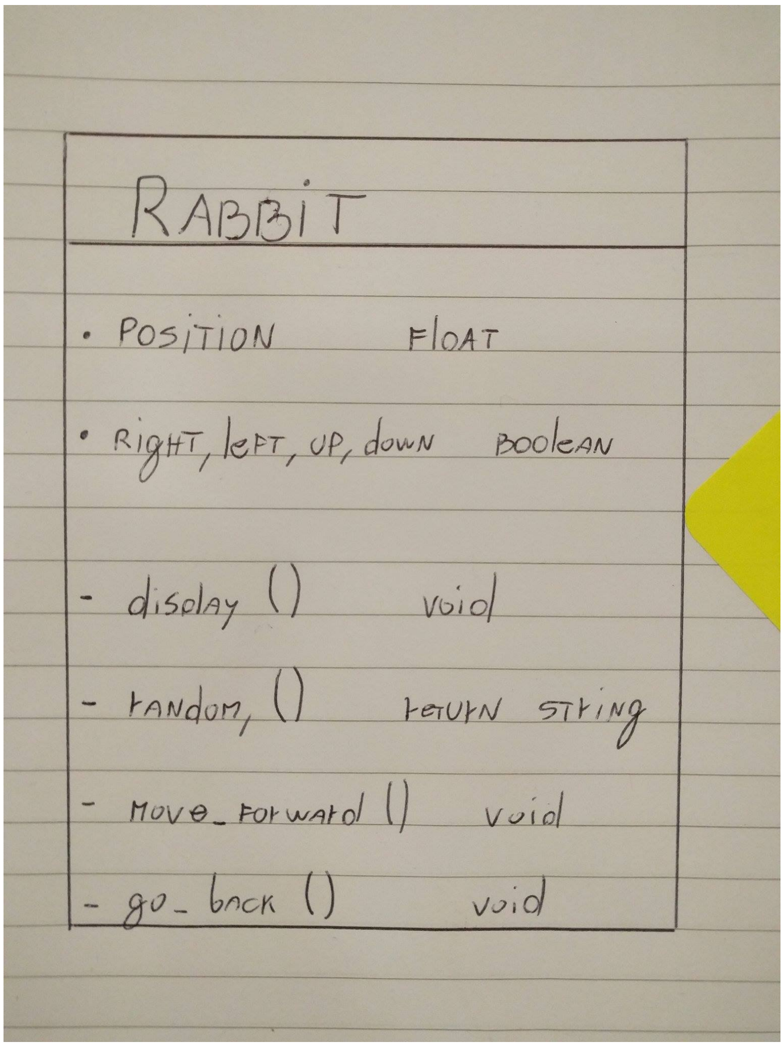
- Void display ();

GRID CONCEPT:

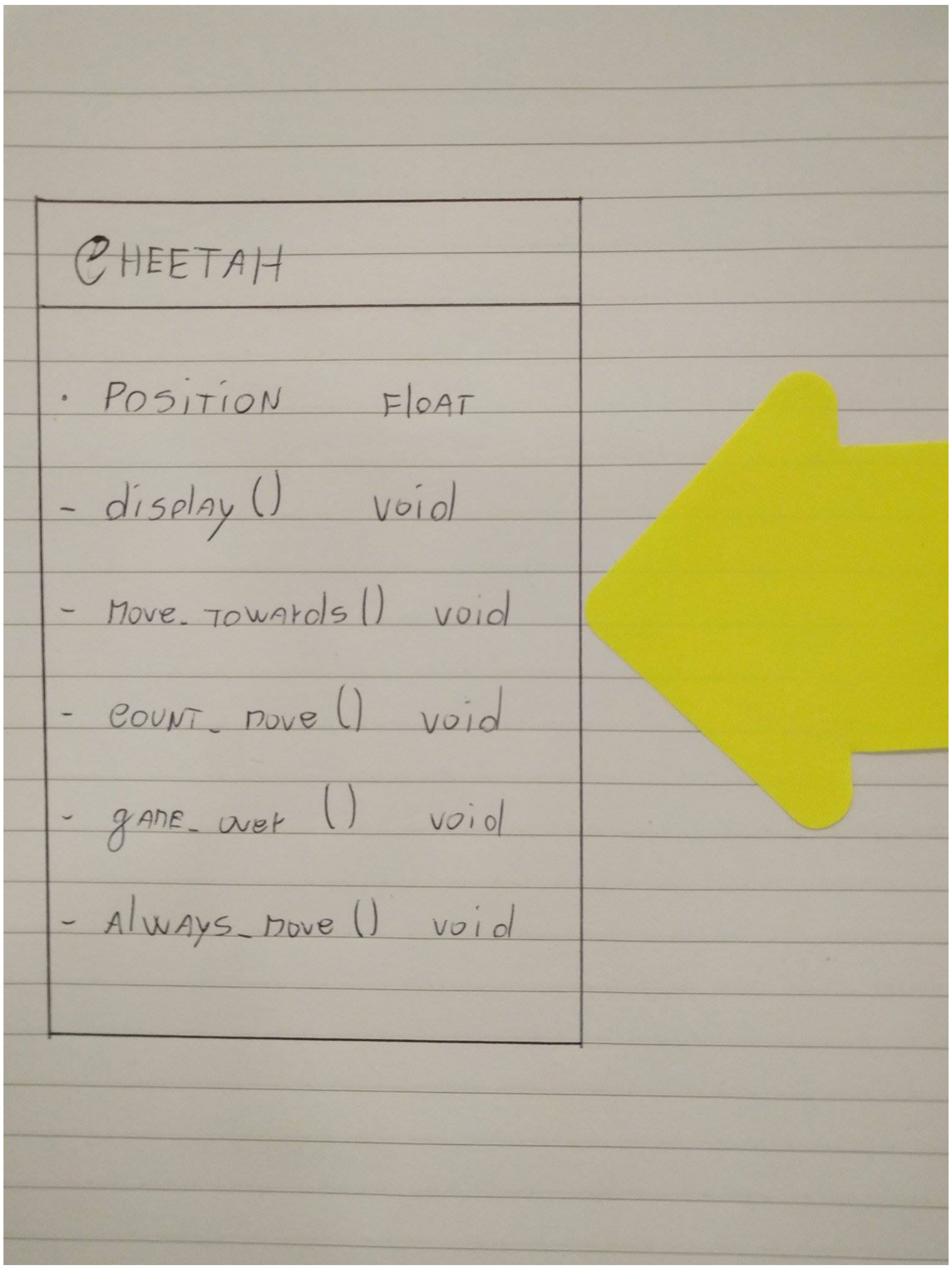
```
void Grid::display()
{
    for (int j = 0; j < row; j++) {//loop through each row
        for (int i = 0; i < col; i++) {//loop through each column
             int x = i*blockSize+offSetx;
             int y = j*blockSize +offSety;
            //strock
            ofSetColor(255);
            ofNoFill();
            font.drawString(to_string(i+1), (i*blockSize+offSetx) + 35, 145);
            font.drawString(to_string(j+1), 15, (j*blockSize+offSety)+50);
            //draw the board
             ofRect(x, y, blockSize, blockSize);
            //setting colour for ellipses
        }
    }
}
```



The first game entity we designed was the rabbit. We implemented this by creating a class which would contain all the informations needed to achieve our goals for this particular character .
The class itself contains the position, the design and the behaviour of the rabbit. The rabbit class is logically inheriting from a base class called 'entity' which only holds the position; x and y coordinates in the grid.
As was asked in the coursework brief, the rabbit moves according to keyboard inputs. We set 'w','a','s','d' (careful not to have the caps-lock key on) as the four basic directions that the rabbit can possibly move, corresponding to 'up','left','down','right'. The main functions that the rabbit class holds are the ones required for displaying it in the grid and updating its position based on the movements.



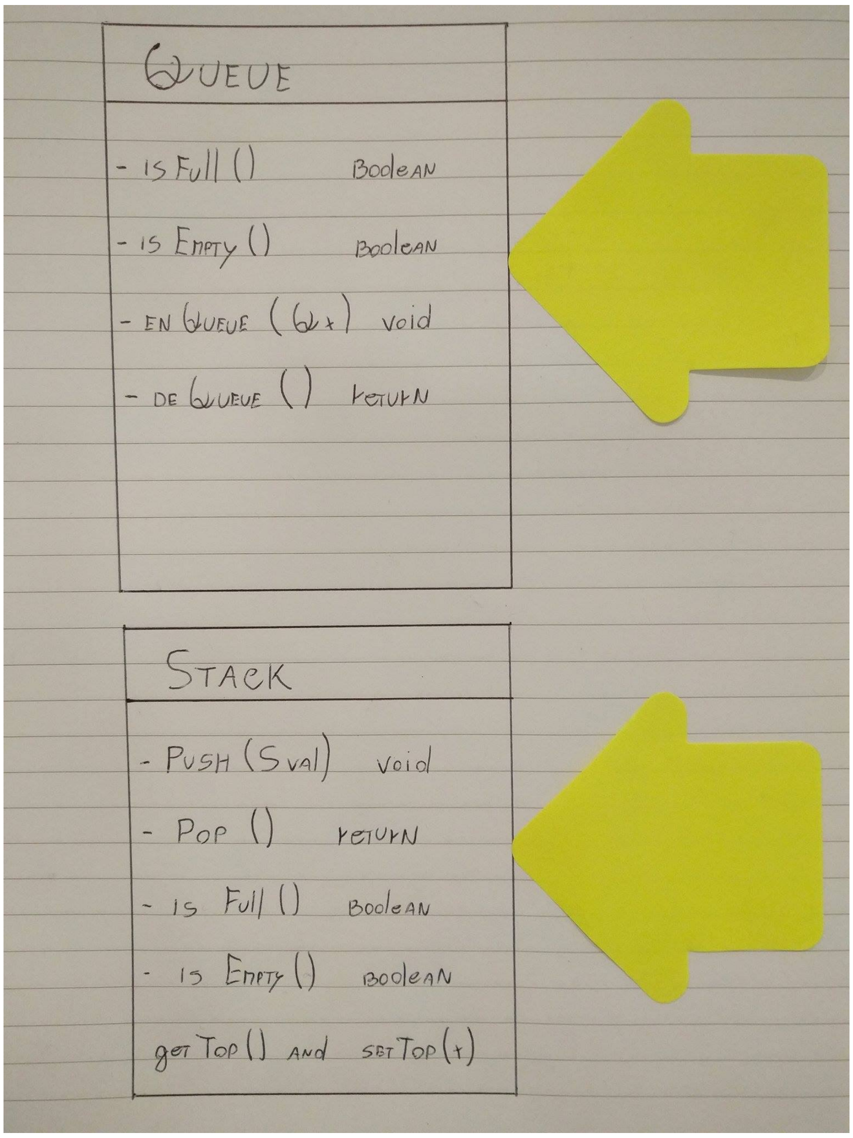
Moving on to the Cheetah character we noticed that to build its structure, it required the use of similar concepts and variables used in the rabbit class and therefore we made the decision of using inheritance to create it. Although the concept was really similar, it was added an extra functionality which would make the cheetah moves towards the rabbit position.



```cpp
void Cheetah::always_move() // functionality added on top of the queue to simulate the cheetah always moves
{

    if(ofGetFrameNum()%30==0) // moves each 30 frames
    {

        if(rabbit.getX()<this->getX()&&rabbit.getY()<this->getY())
        {
            this->setX(this-> getX()-80);
            this->setY( this-> getY()-80);
        }
        else if(rabbit.getX()>this->getX()&&rabbit.getY()>this->getY())
        {
            this->setX( this->getX()+80);
            this->setY(this->getY()+80);
        }
        else if(rabbit.getX()>this->getX()&&rabbit.getY()<this->getY())
        {

            this->setX( this->getX()+80);
            this->setY(this->getY()-80);

        }
        else if(rabbit.getX()<this->getX()&&rabbit.getY()>this->getY())
        {
            this->setX( this->getX()-80);
            this->setY(   this->getY()+80);
        }

        else if(rabbit.getX()>this->getX()&&rabbit.getY()==this->getY())
        {
            this->setX( this->getX()+80);

        }
        else if(rabbit.getX()<this->getX()&&rabbit.getY()==this->getY())
        {

            this->setX( this->getX()-80);

        }
        else if(rabbit.getX()==this->getX()&&rabbit.getY()>this->getY())
        {

            this->setY( this->getY()+80);
        }
        else if(rabbit.getX()==this->getX()&&rabbit.getY()<this->getY())
        {
            this->setY( this->getY()-80);
        }

    }



}
```

After carefully designing and discussing which structure to use, we came to the conclusion that using priority queues in part 1 of our project, would lead us to the concept of reusability when it came to design part 2 of the project. Since a queue is a particular kind of abstract data type in which the values in the collection are always kept in order, and the main operations of the collection are the addition of entities to the back position, known as enqueue, and removal of entities from the front position, known as dequeue and it execute the FIFO procedure, the first element added is the first element to leave the list, we thought it would fit perfectly for our rabbit movements.

The way we implemented the queue data structure with our rabbit is that each time the user inputs a key to move the rabbit; a string corresponding to the key pressed is inserted in a string type queue. This process of storing datas helped us a lot in part two of the coursework.
At the beginning we were not sure what to store into the queue structure and we designed a queue structure capable of holding different types of datas; we achieved this using a template to construct our queue class:



```
string Rabbit::randomDir(vector <string> st) // determine a random direction for the rabbit
{
    if(!stop) // if not sop
    {
    st = directions;
    int temp;
    if(ofGetFrameNum()%10==0)
    {
        temp = ofRandom(0, 4);
        if(this->getY()==160&& temp == 0)
        {
            temp = ofRandom(0,4);
        }
        if(this->getY()==720&& temp == 1)
        {
            temp = ofRandom(0,4);
        }
        if(this->getX()==40&& temp == 2)
        {
            temp = ofRandom(0,4);
        }
        if(this->getX()==1080&& temp == 3)
        {
            temp = ofRandom(0,4);
        }


        moves.enQueue(directions[temp]); // fill the queue with the random direction obtained

    }


    return directions[temp]; // return that direction
    }
```

As part one was only requiring the basics for the cheetah to chase the rabbit we implemented our own algorithms which works as follow:

The way the algorithms works is by checking the rabbit position from the cheetah in the grid and based on this the cheetah makes the correct move towards the rabbit.

This way of implementing the chasing mechanic resulted quite efficient itself but still not as efficient as we thought. For this reason, we designed a new algorithm in order to improve the mechanic. Part 2 of the documentation will explain this process of combining our two algorithms with the aim of developing a more efficient chasing mechanic.
Finally, in order to fully convert the program into a game, we also thought of including a score system where the player could collect yellow dots while controlling the rabbit.
The yellow dots are placed at the beginning of the game and randomised to the next cell once the rabbit collects it.
A score system variable keeps track of how many yellow dots the player has collected .

SMALL DISCUSSION PART 1:
Considering part 1 of our program it was really impressive that we could have a working game/simulation. The algorithms used were quite efficient and resulted fast enough for our goals. Although the fast results we achieved with the cheetah chasing the rabbit, we wanted to add more use of data structure to improve the cheetah artificial intelligence. The idea of implementing a maze/mini-game crossed our mind with the implementation of stacks as a new data structure.

(PART 2) select SIMULATION or GAME_2

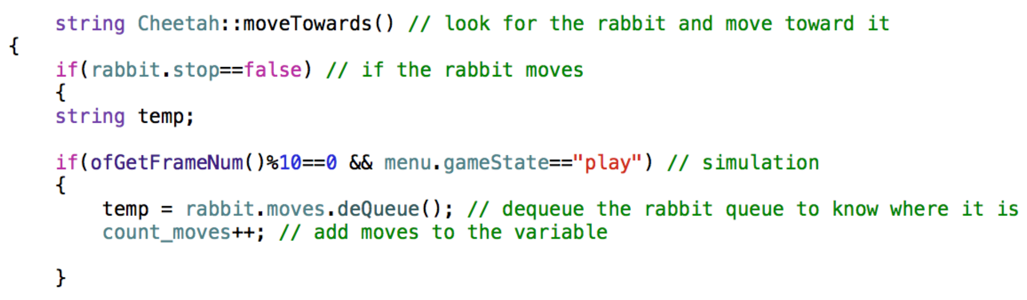
In order to extend our program we thought of upgrading our cheetah and rabbits movements using the data structures discussed in part 1; queues and stacks. There are two main sections of our part 2 for the submission:

SIMULATION:

This part of the program simulates a situation where the rabbit moves randomly and the cheetah knows about the rabbit moves. It is possible to randomise the position of both rabbit and cheetah and start/stop the simulation. In terms of designing the rabbit simulation we started by sketching the rabbit random movements. The way it works is that we have a function which returns every 3 seconds, a string value which correspond to one of the four possible directions "up", 'down', 'left', "right".

As you can see from the picture above, after we randomise a string for the direction, we push this direction into a queue structure. This would allow us later to make a communication between the rabbit and the cheetah.

The way the chasing function works is that each time the rabbit moves, the move is stored into a queue. The cheetah then dequeues the rabbit's queue and based on the position may reverse some directions.



```
    string Cheetah::moveTowards() // look for the rabbit and move toward it
{
    if(rabbit.stop==false) // if the rabbit moves
    {
    string temp;

    if(ofGetFrameNum()%10==0 && menu.gameState=="play") // simulation
    {
        temp = rabbit.moves.deQueue(); // dequeue the rabbit queue to know where it is
        count_moves++; // add moves to the variable

    }
```

Example:

Rabbit -> "right" -> enqueue ("right") -> queue = {right,empty,empty}

Cheetah -> dequeue = "right "-> position check -> move "left" or "right"

We based time complexity measurement and analysis
using a variable which count the steps required for the cheetah to get the rabbit. Each time the cheetah dequeues the rabbit's queue an integer variable counter is increased by one. As first impression the cheetah would require an average of 10 steps to catch the rabbit.

GAME_2

This section of the program combines two datas structure we have studied this year; Stacks and Queues. Since we had a simulation including the basic principles in part 1 and some advanced options for part 2 such as using data structures and allowing the cheetah to move in eight directions, we wanted to implement a maze game where the player inputs some instructions and the cheetah will execute them. The aim of the game is to reach and catch the rabbit in a single input instruction. In case you do not reach the rabbit in a single instruction set, you can go back to the starting position. The way we implemented the "go_back" function is by using a stack. Each time a key is pressed, the direction is pushed both in a queue and stack. When the button "move" is clicked, we dequeue the movement so that the cheetah follows the instruction set. If the "back" button is pressed, we pop the stack and reverse each instruction to let the cheetah go back to the starting position.

Example:

Cheetah -> "right" -> enqueue ("right") -> queue = {right,empty,empty}

Cheetah -> "right" -> push ("right") -> stack= {right,empty,empty}

Cheetah_move -> dequeue "right"-> execute the step

Cheetah_go_back -> pop "right"-> reverse the pop(left)-> step back

In order to make a more complex game we also included trap objects. Traps are game objects which are randomly placed in the grid and must be avoided by the cheetah. If a trap collides with the player there is a game over state.

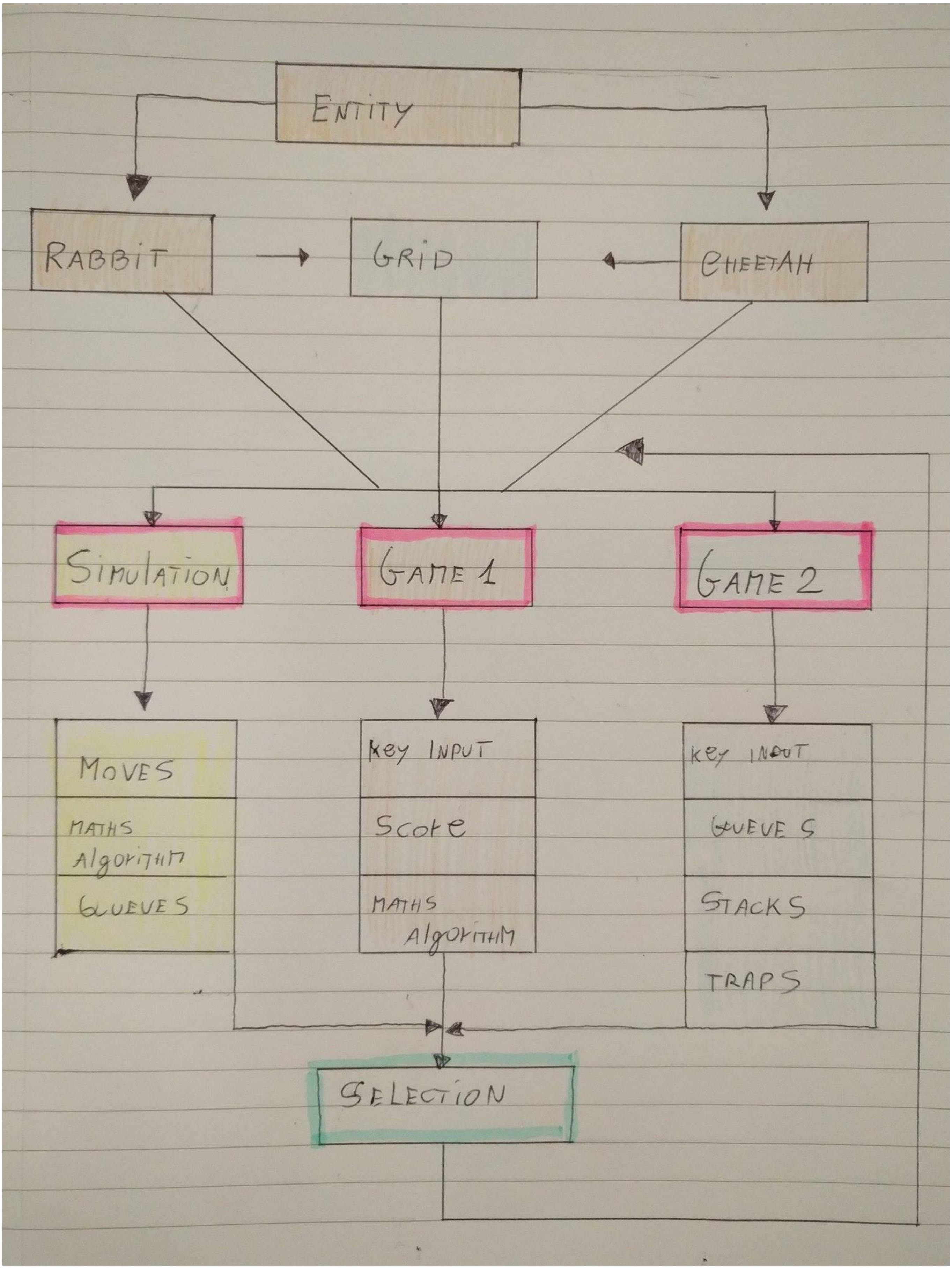
SMALL DISCUSSION PART 2:

We definitely extended our basic part 1 of the coursework by adding more algorithms and game states. We also included more features such as traps and used a second data structure. The efficiency and artificial intelligence of the cheetah has definitely increased from part 1 of the coursework.

GENERAL DISCUSSION AND ANALISYS:

Once the program was terminated we started to evaluate and analyse different aspects of it, including the time complexity and efficiency of the above algorithms described and applied. Generally speaking the time complexity of our program depends and changes in different sections. We are using stacks and queues to insert and remove values, reaching a worst case of O(1) which it is an excellent result.
The grid part for example has an execution complexity of O(n^2) since we are using a nested loop to draw each cell and could be improved.
Talking about the efficiency of the algorithms we used, we did some analysis based on the numbers of steps required for the cheetah to get the rabbit. The algorithm used in part one, was simply calculating the positions of both rabbit and cheetah and using maths to compute the distance and chasing methods. Running the program several times we discovered that it takes an average of 10 steps for the cheetah to get the rabbit considering that the cheetah moves also diagonally. If we add our queue data structure explained above to the maths calculation, it gives us an average of 10 steps,but considering the fact that the cheetah does not move diagonally. Based on this, if the cheetah would move diagonally when using maths and queue structure it would make the process of catching the rabbit more efficient with an assumption of reducing the average steps walked to 6. Also we want to specify that the rabbit and cheetah always started from the same position when we evaluated the efficiency. As a conclusion we have definitely improved the algorithm used in part one and reduced the steps for the chasing mechanic.

Flow Chart Program Concept:



References:

When creating the queue and stacks we followed this tutorial on how to create a queue using templates, since we have not covered this structure in c++: http://www.cprogramming.com/snippets/source-code/templated-queue-class