Department of Computer Science

08101 Programming I

Week 7 Laboratory 2013/2014

C# and Games

In this sequence of practical sessions we are going to make the first components of the game of “Hyperpace Cheese Battle”. Each week we will add a gameplay element so that by the end you will have an implementation of the game. Please don’t worry if you find this hard to understand. We will be giving over the lectures and tutorials for the rest of this semester to this development.

In the first lab we are going to write the part of the game program that will manage the position and movement of players.

The Game of “Hyperspace Cheese Battle”

Hyperspace Cheese Battle is a game of intergalactic conflict and racing. And cheese. Between 2 and 4 players can play. Each player controls a rocket which they move through space. Moves are controlled by the use of a multi-faceted random value indicating system, otherwise known as a dice. Players roll the dice to move their rockets onto space quadrants, otherwise known as squares. Certain parts of space are infused with “Cheese Power” which can be used by the advanced technology in the ships to perform special actions.

Rocket Moves

Each player takes it in turn to throw one dice. They start off the board at the bottom left hand corner and move their rocket onto the board with their first throw of the dice. At each turn they must then move their rocket that number of squares over the board in the direction of the arrow on that square. If their throw would take them off the board their rocket is not moved.

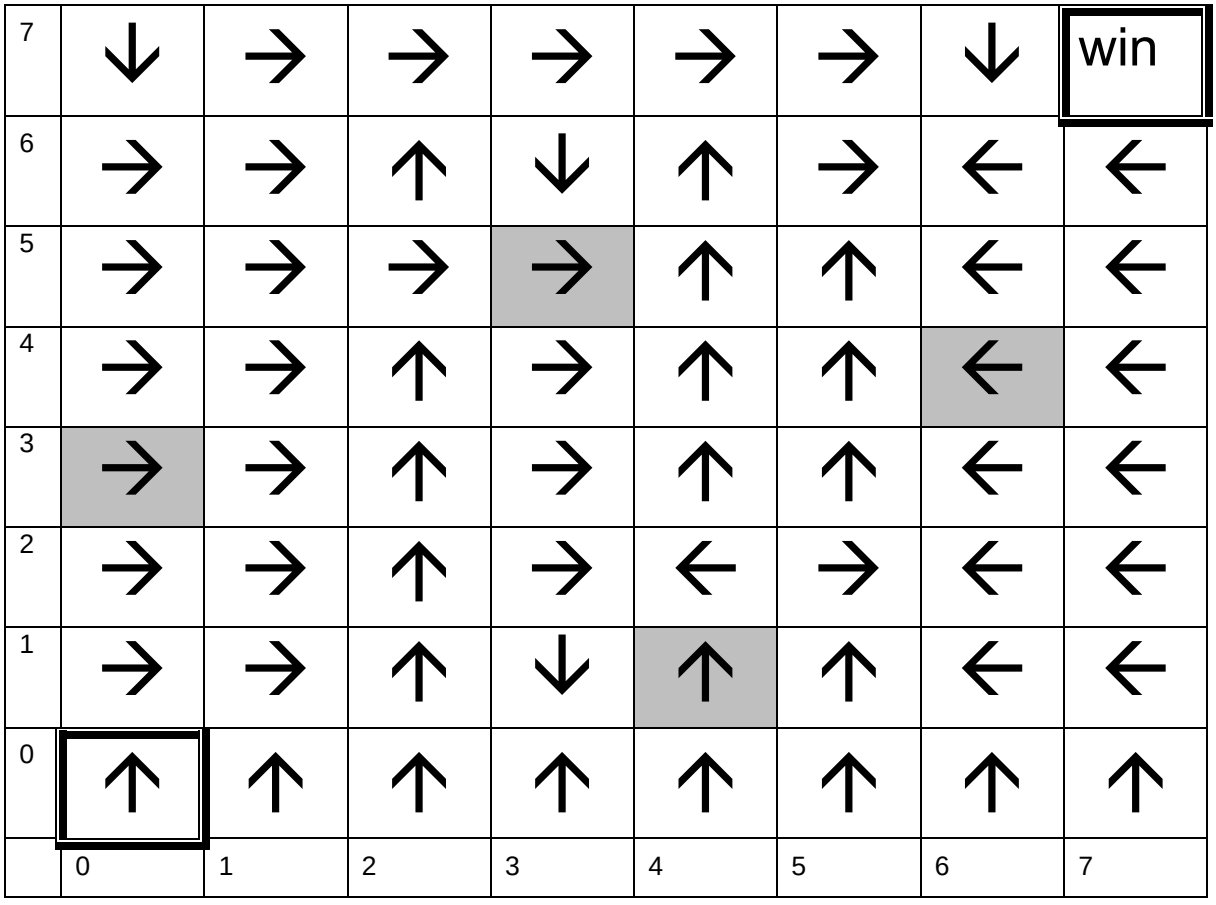
If they land on a cheese square they can perform one of two actions:

- Absorb the cheese power from the square and use it to refuel their engines, allowing them to roll their dice again for an extra move.
- Absorb the “Cheese Power” from the square and create a “Cheese Deathray” which they can fire at any another player. This causes the engines of that rocket to explode, sending it back to the bottom row of the board. The player being sent to the bottom can choose any unoccupied square on the bottom row.

Play continues until a player manages to travel to the square at the top right hand corner. The first player to do this is the winner of the game.

The laws of space and time do not allow a rocket to occupy the same hyperspace as another. If a move would cause a rocket to land on an occupied square the rocket is moved to the next free square in the direction of the arrow on the occupied square. If that square is also occupied the rocket is moved in the direction of the arrow of the next occupied square until an empty square is found.

Sample Gameplay



We can regard the board as a two dimensional array using the coordinates as shown above. The starting square at the bottom left of the board will have the coordinates (0,0) and the winning square at the top right hand side of the board will have the coordinates (7,7)

Arnold (A), Betty (B), Cedric(C) and Damian(D) all sit down for a game of “Space Cheese Battle”.

1. A rolls a 2, follows the arrow on (0,0) and is placed on square (0,2).
2. B rolls a 2, follows the arrow on (0,0) and discovers that square (0,2) is occupied. She follows the arrow on (0,2) and moves onto square (1,2)

3. C rolls a 3, follows the arrow on (0,0) and lands on square (0,3). This is a Cheese Power square. C decides to roll the dice again and rolls a 4, landing on square (4,3).
1. D rolls a 3, follows the arrow on (0,0) and lands on square (0,3). This is a Cheese Power square. He decides to explode the engines of C's rocket, and C decides to put their rocket on square (6,0).

This shows how the game works. Gameplay continues until one of the players lands on the winning square.

Storing the Board

1. Start Visual Studio.
2. Use `File>Project` to make a brand new Windows Console application. Call the application *HyperspaceCheeseBattle.*

Each square of the board contains an arrow which gives the direction of movement from that square. The game needs to be able to use the position of the player to determine which way the player should move.

One way to store the board is to create a 2D array which holds a value for each square in the board. The particular value in a given square will give the direction of movement for that square. A program can create a pre-set array of values by using the following construction:

```
static int[,] board = new int[,] {
    {0,0,0,0,0,0,0,0}, // row 0
    {0,0,0,0,0,0,0,0}, // row 1
    {0,0,0,0,0,0,0,0}, // row 2
    {0,0,0,0,0,0,0,0}, // row 3
    {0,0,0,0,0,0,0,0}, // row 4
    {0,0,0,0,0,0,0,0}, // row 5
    {0,0,0,0,0,0,0,0}, // row 6
    {0,0,0,0,0,0,0,0}  // row 7
};
```

The C# code above creates a 2D integer array called `board` and then initialises each of the elements to the value 0. You will have to set the initial values so that each element of the array holds a value that represents the moves possible for that square. For example you could decide that the value 1 means "up arrow". In this case the array would look like this:

```
static int[,] board = new int[,] {
    {1,1,1,1,1,1,1,1}, // row 0
    {0,0,1,0,1,1,0,0}, // row 1
    {0,0,1,0,0,0,0,0}, // row 2
    {0,0,1,0,1,1,0,0}, // row 3
    {0,0,1,0,1,1,0,0}, // row 4
    {0,0,0,0,1,1,0,0}, // row 5
    {0,0,1,0,1,0,0,0}, // row 6
    {0,0,0,0,0,0,0,0}  // row 7
};
```

The array now contains values representing all the up arrows. **Note that in the above setup the value at the top left of the array is the element at [0,0] in the array, in other words the array is upside down as compared to our diagram.**

1. Devise a way of representing each direction of movement and create a board array that gives the direction of movement for each of the squares on the board. You might like to look at using an enum to hold the direction.

Storing the Player Positions and Names

The program will need to store details about each player. This information will include the name of the player and their position on the board.

The name of the player can be held as a string and the position can be held as an X value and a Y value. It would seem sensible to store this as a structure:

```
struct Player
{
    public string Name;
    public int X;
    public int Y;
}
```

Up to four players can play the game, so we could create an array that holds the information about four players.

```
class Program
{
    static Player[] players = new Player [4];

    static void Main()
    {
    }
}
```

The player positions are held in a static array variable which is part of the class. This means that all the methods that we make in the game will be able to use these values. The details of player 1 will be held in element 0 of the array, player 2 will be in element 1 and so on.

The player position value holds the x and y coordinates of the square that the player is on. At the start of the game the player is just before the start square, so we can set their position to be (0,0) at the beginning of the game.

3. Add the variables to hold the player position information.

Starting the Game

The first thing the game must do is determine how many players there are and then put each player at their start position.

```
static void ResetGame()
{
    // TODO: get the number of players and set the required elements in
    // players to put each player at square (0,0)
}
```

2. Add the `ResetGame` method to your game as above.
3. You will need to store the number of players in your game. Create an appropriate variable.
4. Put in the code to ask how many players are taking part, store this number in your program and clear the `players` array.
5. Create a test in `Main` that proves your method works.

Creating a Dice

Each time the player makes a move they will throw a dice. We can implement this as a method that returns the throw the dice has made:

```
static int DiceThrow()
{
    return 1;
}
```

This is a very boring dice that always returns a score of 1. We will improve it later.

1. Add the `DiceThrow` method to your program.

Managing a Player Turn

You move a player by adding a dice throw to the player position. We can write a method to move a player onto their next square:

```
private static void PlayerTurn(int playerNo)
{
    // TODO: Makes a move for the given player
}
```

The method should roll the dice (i.e. call `DiceThrow`) and then move the player on to the destination square. The program must get the direction of movement from the square that the player has reached.

- If they are moving up the dice throw should be added to the Y coordinate
- If they are moving down the dice throw should be subtracted from the Y coordinate
- If they are moving right the dice throw should be added to the X coordinate
- If they are moving left the dice throw should be subtracted from the X coordinate

The program should make sure that the move does not take the player off the board. In other words the value of X or Y should never be greater than 7 or less than 0. If a move would take the player off the board then an appropriate message should be displayed and the player should remain on their original square.

1. Add a `PlayerTurn` method to your program.
2. The `PlayerTurn` method should access the appropriate element in the `playerPositions` array and move it down the board.
3. It should also print out the location of the square that the player has landed on.

Testing Your Program So Far

You now have some methods and behaviours. Now we can write some tests that will allow us to prove that they work as we would like. We are not going to create the game with these tests, but we are going to prove that the game will work correctly when we do build it. This is standard development technique which is called “Test Driven Development”.

Add some statements to the `Main` method that test the following.

1. Get the number of players and reset the game.
2. Make a move for each player (use a for loop for this).

Handling Rocket Collisions

The present version of `PlayerTurn` doesn't work properly. After each rocket has moved in the test program above they are left on the same square (0,1). This would cause a rift in the space time continuum and probably bring about the end of the universe. So we need to fix it.

When a rocket lands on a square containing another rocket they must be moved on to the next square. If that square also contains a rocket they must move onto the next one along, and so forth.

The best way to handle this would be to create a method that checks to see if there is a rocket on a particular square.

```
static bool RocketInSquare(int X, int Y)
{
    //TODO: write a method that checks through the
    //rocket positions and returns true if there is a rocket in the given square
}
```

Just before we move our rocket we can call this method and see if any of the rockets are on the square we are heading for. If there is a rocket there we need to move on to the next square. Of course, if there is a rocket on that square we need to move on to the next, and so on. If you are thinking this would be a good place for a while loop then you would be right. The loop should go in the `playerTurn` method, as this is the place where the rocket will be moved.

Before you try to write the code to do this you should think about the sequence of steps that need to be followed to make this work. This might be a good place for a while loop, as you want the rocket to keep bouncing while there is a rocket on the destination square. You can think of a "bounce" as a move with a dice throw of 1.

Completing the Lab

At the end of this practical you should have methods that reset the game and make moves for the players the player name and make a player move. The move should work correctly if every player throws a 1 in their first turn, leaving the rockets as follows

- Player 1: (0,1) – arrives here first
- Player 2: (1,1) – bounces off Player 1
- Player 3: (2,1) – bounces off Player 1 and then Player 2
- Player 4: (2,2) – bounces off Payer 1, 2 and 3 following the arrows each time

Remember that when a rocket "bounces off" another rocket it bounces in the direction of the arrow on that square.

Rob Miles November 2013