**National University of Singapore**

AY14/15 S2

**CS4344: Assignment 3**

**Interest Management**

**A0105829B, A\_\_\_\_\_\_\_X**

## Observations about Space Battle

When we tried the Space Battle game out with 10 bots, we made a few important observations:

1. The speed of both entities of the game, ships and rockets, are slow-moving.
2. There were only 4 directions that both entities of the game, ships and rockets, which are, namely, left, right, up, and down.

With these observations, we devised our interest management strategy.

## Our Interest Management strategy

Since entities are slow-moving, in general a player would not be interested in trying to shoot down an enemy far away because the rockets would not likely ever hit the player that is aimed at. For a player, it would be more reasonable to be aiming at and chasing down the enemies (and also dodging rockets) in his own vicinity, and so we decided that we should use some sort of distance-based interest management scheme.

Furthermore, since entities can only move in one of four directions at any one time, we decided that it would be reasonable for the area of interest (AOI) of a player to be something like a '+' sign, where the player is in the middle of it.

Assuming that there will always be many players, the size of the '+' AOI we chose for this game has a width of 50 (how thick the '+' is), and a length of 150 (how far the '+' stretches away from the player) because we felt that there is no need to be bigger than this while play-testing the game, since that is about 5 times the size of the player's character, the ship, and it is unlikely that a player would be interested in shooting at other players further than this distance and also rockets are dodge-able at this distance. If there were very few players, the AOI could be scaled dynamically to allow players to find each other more easily.

In order to find all the other players we are interested in quickly, we adopted a grid(cell)-based approach in which the entire game world is broken up into cells of a fixed size and players would be subscribed to cells they are interested in. Then, everytime there is some interesting event from any player, it is sent to players who are subscribed to the cell(s) in which it happened. For example, when a player turns, all players subscribed to the cell which that player is in would receive the turn event. Also, everytime a player crosses a cell boundary the server will update its cell-subscription and send an update to all subscribers of the player's new cell in order to update the players who are subscribed to this new cell but not the cell that the player was in previously. Lastly, the 'fire' message will not be sent to all players all the time, but rather only when the rocket enters the AOI of the player. We ensure that the 'fire' message will be sent at most once to each player as well, so as to not perform even worse than without interest management.

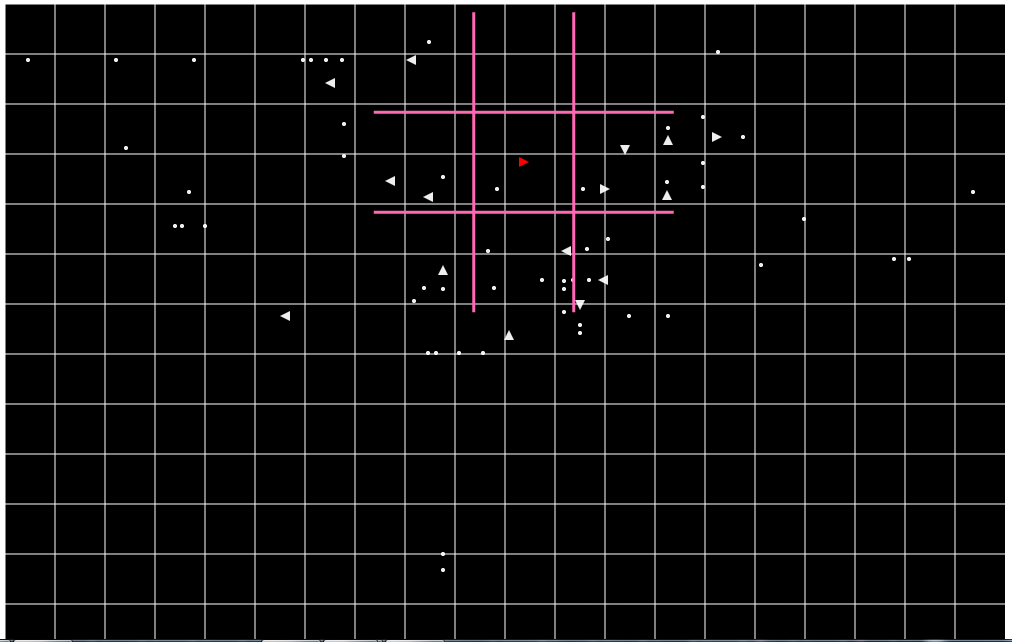


Figure 1: Visual grid-cells and AOI of player with 100 bots in game

Figure 1 above shows how our AOI and grid looks like visually, and it is a real image captured from a play of the game. It may be important to note that we decided to stop rendering the enemies who have left our AOI for more than 5 seconds because if we let the enemies who are not in our AOI continue to render, then the 'ghost' of the enemy might come right next to the player while the actual enemy is somewhere far away.

Finally, to avoid checking every rocket with every ship for collisions, we also decided that a circular AOI for each rocket would be helpful because a rocket is only interested in ships at most (rocket.radius + ship.maxWidthOrLength) radius away from its center. The cells are still used to implement this, and given the small radius (2) of the rockets and the small length (10) of the ships, the rockets will only be interested in ships from at most 4 cells (since our cells have width 50) at any one time and most of the time they would only have 1 (or 2 if travelling along the border of a grid line) cells which contain ships that they may be interested in, and so reduces the number of collision checks made drastically.

## The Implementation

To implement the strategy described in the section above, we created one new javascript file called 'Cell.js'. This module's main purpose is to hold the information about the currently subscribed ships/rockets.

The main changes made are in MMOServer.js. There are many new constants and variables to implement the changes mentioned. There are also many new methods which help us achieve the functionality we require, for which it is probably better to see the source code directly for the comments there about what they do individually. Finally, after we have the new constants and variables, we are able to make the main changes which are shown in Figure 2, 3, 4, and 5 below.

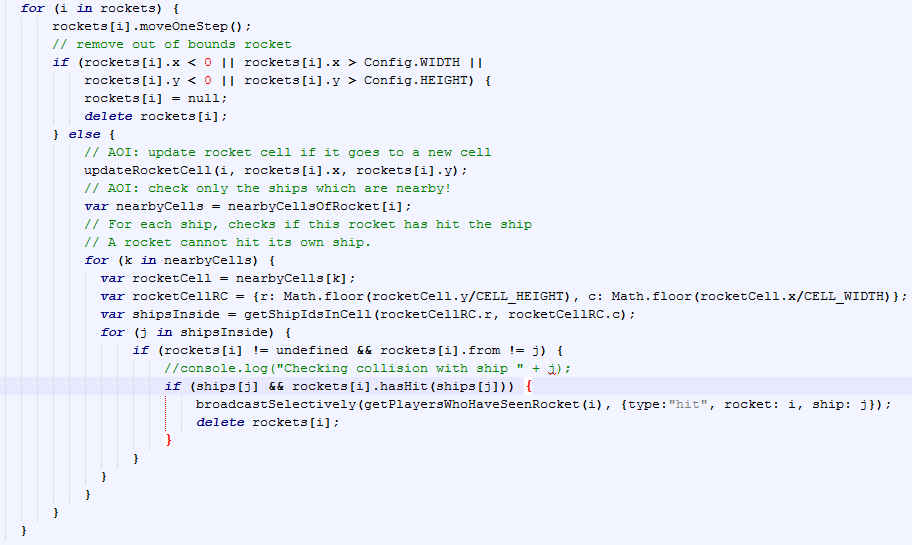
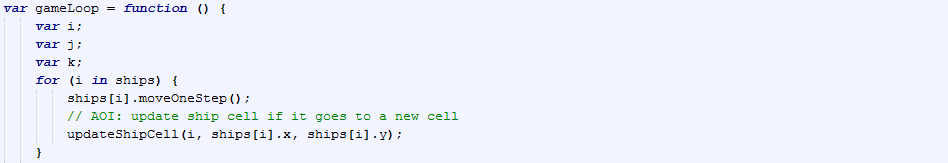


Figure 2: game loop

For figure 2 above about the game loop, the main changes are:

1. new line updateShipCell(i, ships[i].x, ships[i].y) after moving each ship by one step. This method maintains the ship's cell subscription.
2. updateRocketCell(i, rockets[i].x, rockets[i].y) after moving each rocket by one step if they are still within bounds. This method maintains the rocket's cell subscription.
3. Instead of checking all ships for every rocket, we check only the ships inside the nearbyCells of the rocket which is stored and maintained in the associative array called nearbyCellsOfRocket.
4. We do not broadcast the message to every player in game, but instead only to the players who have seen the rocket before.

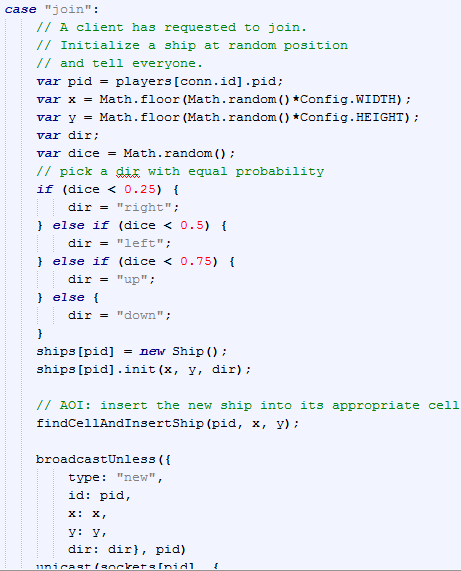


Figure 3: when a player joins

In figure 3 above, inside the switch statement of the types of messages received by the server, there is only one new line that is called findCellAndInsertShip(pid, x, y). This method does exactly what it says, and finds the cell that the ship is in and inserts (and subscribes) the ship.

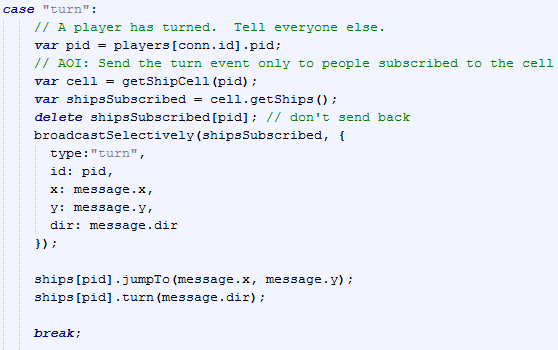


Figure 4: when a player turns

In figure 4 above, inside the switch statement of the types of messages received by the server, there is only one main change, that is the turn event is now sent to only the players subscribed to the cell where the event happened. Note that we omit the code that sends turn events to subscribed players everytime the player goes to a new cell for brevity; it is similar to this code.

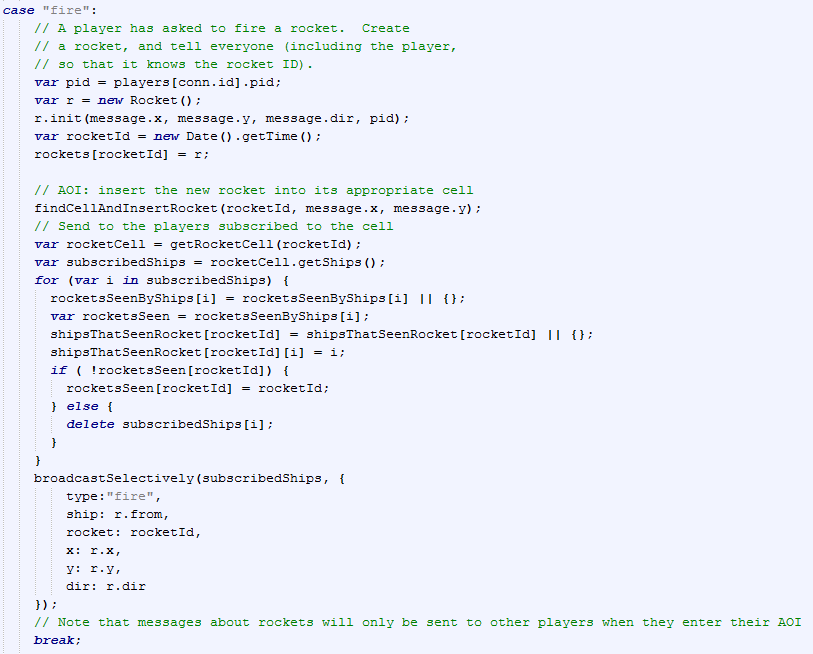


Figure 5: when a player fires

In figure 5 above, inside the switch statement of the types of messages received by the server, there is only one main change, that is the rocket is inserted into its cell by findCellAndInsertRocket, and the message is only sent to players who are subscribed to the cell where the rocket is fired, instead of being sent to every player no matter what.

## Conclusion

We managed to implement the interest management strategy described above fully, and it is working and with a middle-end laptop as the server Space Battle can be played smoothly with slightly more than 100 bots without much visual disruption. However, we would like to note that with 300 bots in the game the laptop as a server is not fast enough to process all the messages coming in on time and the game is hardly playable. Probably, for more than about 200 bots/players there is a need for multiple servers to make it scalable. We also gave thought into implementing the multiple-server architecture based on zones of the grids, where a player would connect to multiple zone servers if its AOI crosses into those zones, but due to time constraints we determined that we would not be able to implement it to a good standard before the deadline.