Game of Life

Devan Patel, Kyle Koceski, Daniel Fuentes

CIS 4930: Concurrent Programming

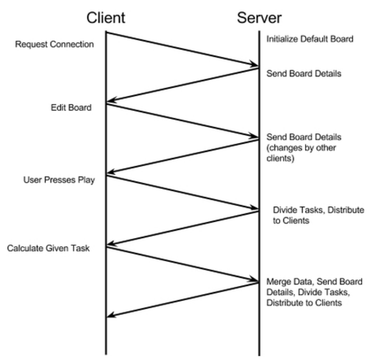
Introduction

Jon Conway devised an algorithm to mimic cell automation in 1970, called the 'Game of Life'. Essentially, the game consists of a finite grid of cells that are either 'alive' or 'dead' and are dependent on the following rules:

- Any live cell with fewer than two live neighbours dies, as if caused by under-population.
- Any live cell with two or three live neighbours lives on to the next generation.
- Any live cell with more than three live neighbours dies, as if by overcrowding.
- Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

Objective

Our Game of Life implementation uses a server-client architecture, controlled and played on multiple clients. Iterations are computed on multiple clients; the server will constantly transmit the grid to clients after each iteration. When played, the server will distribute tasks of calculating iterations amongst the clients, and the calculations are returned to the server; it then merges the calculations, distribute the board and tasks to the clients again. Clients update grid size and initialialize living cells in order to play and then receive commands to calculate the next iteration. After these calculations are complete and sent to the server, client waits for future information.



Member Contributions

- **Devan Patel** developed a concurrent algorithm determining neighboring cells, calculating the next iteration, and synchonously updating the grid.
- **Kyle Koceski** managed the appropriate synchronization of multiple client threads and communication between clients and server.
- **Daniel Fuentes** designed the graphical user interface that allowed clients to interact with the board, as well as displaying the board's current status, and the changes as the game progressed.

Results

NEED TO ADD GRAPHS

Concurrency Involved

Server:

- Synchronization on client list which is necessary to ensure a consistent view of the current clients before distributing tasks
- Utilizing a Lock over HashMaps associating Connections to Tasks and Tasks to Returned Components
- "Barrier" synchronization using Object.wait() and Object.notifyAll() to inform the playThread when all clients have returned back partial nextIteration objects

Client:

- GUI thread and ClientConnection thread
- ClientConnection uses JavaFX marshalling functions to receive information from Server and display it the the user
- Calculates nextIteration from the tasked partial components by distributing amongst threads (relatively equivalent to the number of available processors), sending back to server after threads complete

Conclusion

Nunc tempus venenatis facilisis. **Curabitur suscipit** consequat eros non porttitor. Sed a massa dolor, id ornare enim. Fusce quis massa dictum tortor **tincidunt mattis**. Donec quam est, lobortis quis pretium at, laoreet scelerisque lacus. Nam quis odio enim, in molestie libero. Vivamus cursus mi at *nulla elementum sollicitudin*.

Technologies

The following technologies were required to complete the project:

- Java 1.8
- JUnit 1.4
- JavaFX 2.0.3
- Java Swing

Improving Results

Factors that would improve our results include:

- Support more cores for threads in order to extrpolate our data
- Support larger grids to determine speed for different grid sizes

Acknowledgements

WHAT TO ADD HERE?