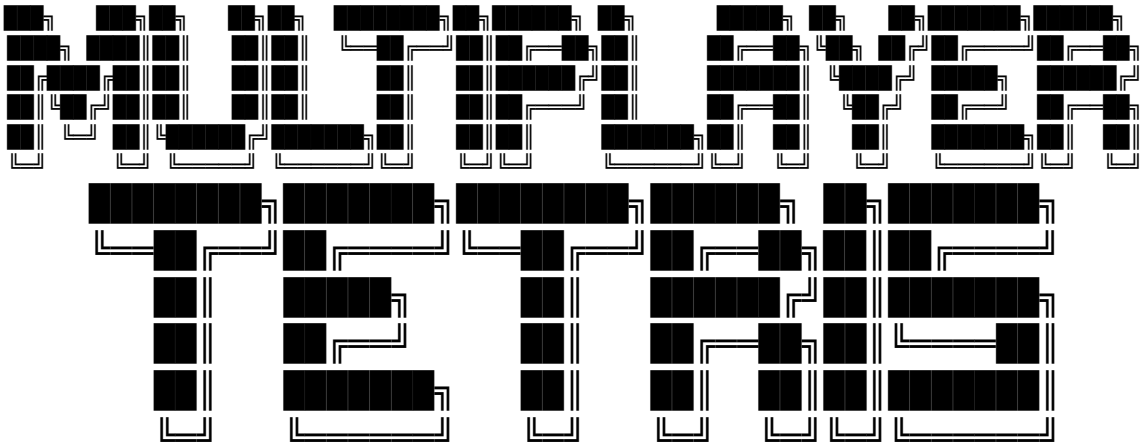

Amelia Cook, Elliot Bonner, Alex Williams Ferreira, Trevor Sullivan

Final Refinement of Design

Below are two diagrams of our code modules. The first represents the code modules and the function calls that are made between them. The second diagram represents the processes that exist within the single-player version of the game and the messages that are passed between each. We believe that both diagrams together give the best picture of the code.

The module diagram looks very different from the initially proposed design, including many additional components and many additional connections between them. As we continued to build out the project, modules were added to improve readability, modularity, and extendability.

Improvements towards modularity centered around organization and abstraction. The tetromino module was created to contain its representation and operations. Functions like get_all_coords, generate, and rotate are specific to the “tetromino piece” and will never be used outside of actions related to it. This idea inspired the addition of board.erl and others. Breaking up what we originally had as a very general “client” module.

When it comes to the IO Module, our new diagram remains decently loyal to our original one. It maintains that each client process will have its own IO processes, with almost all cecho calls being in tetris_io.

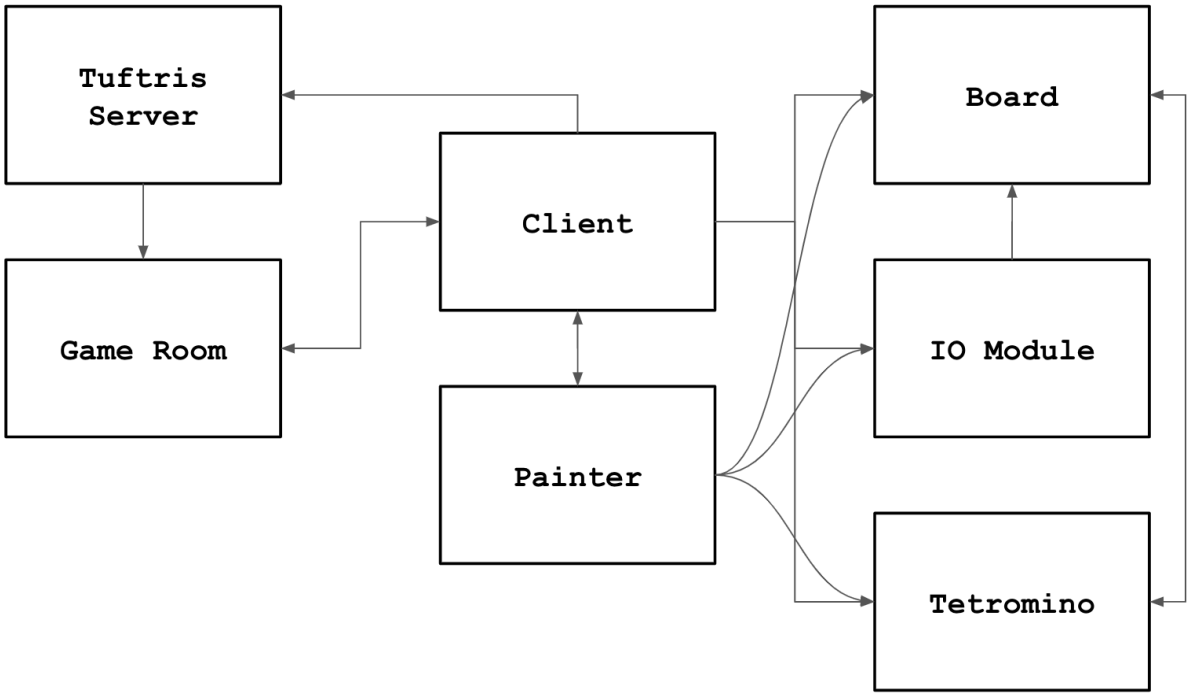

Figure 1: Module Diagram

We didn't have a process diagram at the beginning of the project, but we feel that it represents the functionality of our program well. The process diagram is quite different from what we had imagined at the beginning of the project (a process for the server, client, and I/O). As we implemented more features, we realized that adding more processes helped split up the work (and in some cases, like the timer and keyboard input, was necessary for functionality). In this diagram, each box represents a process, and each arrow represents communication (one-way or two-way) between processes. The processes are:

- Tuftris Server: server that manages all of the game rooms–the client connects with it initially before connecting with the game room
- Game Room: process that connects all of the players into one game
- Client: client process that handles most of the game logic
- Painter: updates the boards/screen during the game
- Keyboard: waits for keyboard input from the player
- IO Module: CEcho process that handles NCurses functions
- Refresh: process that checks if the user resizes the window

These are all of the processes that are active *in a singleplayer game.* At any point in time, there is only one Tuftris Server process, but there is not limit to the amount of games that can be played at once (except for the number of Erlang processes that can be spawned). The rest of the six processes are necessary for each player.

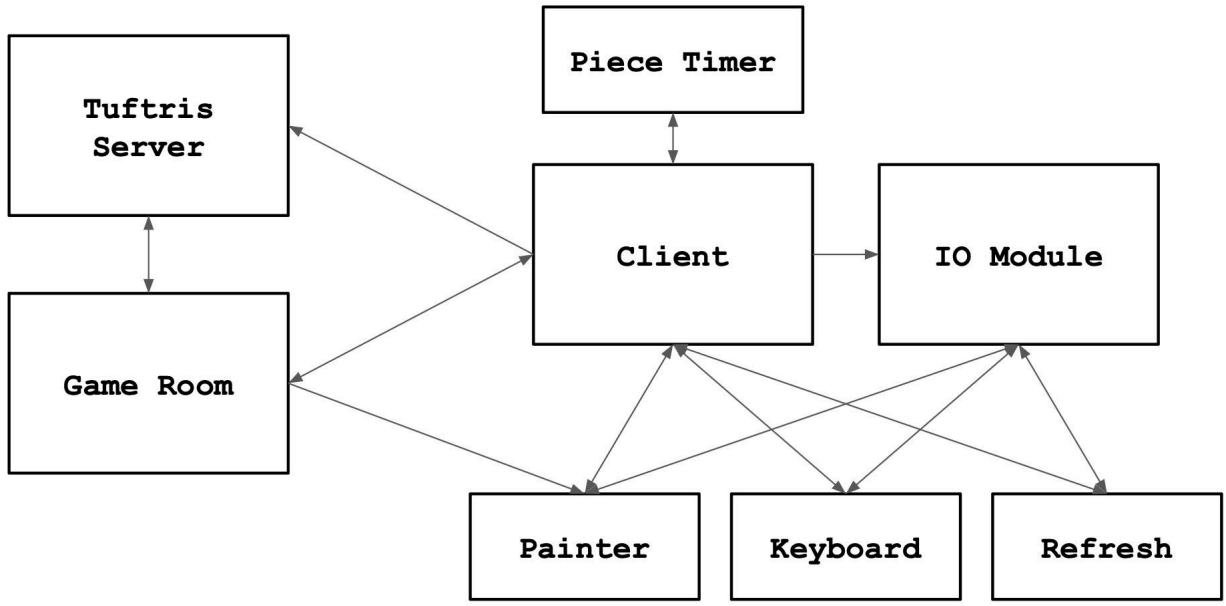

Figure 2: Process Diagram

Outcome Analysis

Overall, we confidently believe that we have achieved our initially proposed MVP. We have implemented a fully functioning concurrent multiplayer Tetris game using Erlang and an NCurses library called CEcho. The proposed maximum deliverable included three game modes and turned out to be ambitious for our time frame, but we were able to implement both singleplayer and multiplayer with many additional

features. Some of these features include the ghost piece that tracks the player's block, increasing the speed of the falling block as more rows get cleared, animations for clearing the rows, window resizing, blocking out and the resulting background on other players' screens during multiplayer games, post-game podiums and rankings, and multiple games running at once. These features make our game more polished overall.

Given the time in this project, or perhaps in our time after this project, there are some features that would be cool to implement. Some of the basic features we want to add would be additional safety checks for edge case situations (i.e. when a player quits after creating a room before any other players have joined). Another safety feature would be registering PIDs as players enter the game, since currently any zombie nodes named after a player's UTLN and VM could prevent them from playing the game. Some additional features to improve the overall functionality and coolness of the game would include a lightswitch to start and stop the server, a point system, a better piece generation algorithm, a persisting high-score and leaderboard, piece swapping, displaying the levels, displaying other players' current pieces, and changing your own board to the blocked-out background as you watch other players continue a multiplayer game you've lost.

Reflection on Design

The best design decision we made was using Erlang to implement this project. While that may seem pretty broad, this choice made the project a lot easier in many ways that we didn't fully realize until almost the end of the project.

For example, we could trust that our messages would reach their destinations. We took a lot of time in the beginning to make sure that our messages wouldn't be too heavy, but after we realized that each of these messages would then be received in a specific order we no longer had to worry about some sort of timing or ordering of events. That then inspired us to use the client-server architecture rather than the peer-to-peer model so we could have a centralized source of truth for all of our players.

We also didn't have to worry about the networking aspects of the distributed game at all. Not only does erlang make it easy, but we also had prior experience from the chat server project. This all enabled us to begin architecting from the very beginning without having to learn a lot of new code or technology. Additionally, as discussed above, we didn't necessarily realize the full scope of the project while we were initially architecting it. There are a lot of features that we decided to add just for fun, but also a lot of details that we felt were necessary for the MVP. As we added functionality we had to keep adding additional processes. Using a language like erlang made the cost of that incredibly cheap, whereas these changes and redesigns would have been crippling in other languages.

While the language decision is certainly a big one, one of our biggest redesigns was to create a painter process to handle most of the IO, and it resulted from the bug that we discuss below. If we had to do the project over again, architecting with that in mind from the beginning would have saved us a lot of time and effort in the long run. We didn't realize that the IO module we were using, CEcho, would cause a race condition across multiple processes. The change we implemented enforces synchronicity between actions with calls to the IO module.

Reflection on Division of Labor

Rather than split the project into tasks and work on it as homework, we worked mostly as a whole group. Within group meetings the tasks of the day were split up among the group members, but the overall work was completed collaboratively. This method allowed us to work independently or in pairs to make progress while giving us the opportunity to debug or discuss large architectural changes as a group. Some features were added independently, and some bugs were resolved independently, but overall the work was collaborative.

Bug Report

There were many interesting bugs throughout the development of this project, relating to concurrency and architecture and simple mistakes. Our project started with the implementation of a single player game using the architecture of the multiplayer product, and one of the more interesting bugs came up when we were beginning to draw additional boards on the screen. The objective was to paint a blue board next to the active game as a proof of concept, before any messages or additional functionality were applied to the secondary board. The second board was painted by a separate process from the main player's board so it could receive messages separately, which ended up being the source of the issue.

Drawing in NCurses involves two separate operations: setting the color and painting the coordinates. Our code draws a square by first setting the color then requesting a specific coordinate to be painted. This pair of operations is not atomic as a whole, and the function calls were not within a critical section, and since there was only one instance of NCurses for the screen the color was global across the processes, so when multiple processes were painting on the screen at the same time the colors became interweaved. We had a concurrency bug!

While this bug was not the most difficult to track down, it required the most work to solve by far. Initially we had thought to have a process listening for messages from other players and painting each of their boards while the main process worked on the main player's board directly. To solve this bug, we made that process draw all updates to the board, which required us to pass around the PID of this listener and create many new message passing schemes for things that were previously handled in function calls. Ultimately this change made our code much cleaner and abstract.

Code Review

See Figure 2 for an overview on each process and the communication between processes.

tetris.erl

Contains the client-side game logic–most of the file is the client process, and the timer process is also included.

tetromino.erl

Contains functionality for creating and manipulating tetrominos.

board.erl
Abstraction for the board–contains functionality for creating and manipulating board data structures.

server.erl
Contains implementation of the server process that manages the game rooms

game.erl
Contains implementation for the game room process

painter.erl
Contains implementation of the painter process.

tetris_io.erl
Contains functions for manipulating the screen for a user and the keyboard input and refresh processes. The IO Module process (CEcho) is not implemented by us.

tetris.hrl
Contains general-prupose macros.