Abstract:

This project explores the benefits of functional programming to both game and web development through building an original multiplayer game server application using F#, a modern functional language which integrates with the .NET framework. An accompanying JavaScript client supports fast paced gameplay and explores the usefulness of websockets provided through the .NET signalR library. The development process included implementation of substantial server and client code to provide simulation of “real time” interactivity using latency compensation techniques. This project demonstrates the usefulness of functional programming to game development by delivering superior runtime safety, strong support for concurrency, and stateless design patterns which may alleviate performance risks and reduce code complexity. At first glance, game development favors object oriented programming by naturally using hierarchies of classes to model game worlds. However, as these systems grow in complexity, they are notoriously difficult to maintain. Despite data structures that clearly represent game state, programmers may introduce functions which rely upon unsafe referencing and mutation of game elements, threatening the security, performance, and extensibility of the system in unforeseen ways. Functional programming combats these issues by imposing immutability, strong typing and composable program flow, eliminating opportunities for side effects and careless design. While tradeoffs between object oriented programming and functional programming are relevant in all software applications, multiplayer games provide several dimensions in which paradigm choice heavily impacts implementation, and where the advantages of a functional approach are numerous and interesting. This project aims to promote openness toward different paradigms and technologies with an application example that is exciting to a broad audience.

Introduction and Background:

Multiplayer Game Design

Networked multiplayer games are challenging to implement and theoretically interesting for many reasons \(1, 7, 9, 12, 13\). Single player games – and offline multiplayer games – are relatively simple, consisting of input collection, state processing and rendering steps performed in a continuous loop. These games achieve truly “real time” gameplay, as input can be collected from the keyboard or from several player’s controllers almost instantaneously. Conversely, networked multiplayer games must reconcile severe bandwidth and latency limitations, and are vulnerable to the gamut of security risks associated with distributed systems in general; game data may be too large to quickly transmit over the network, and malicious users may attempt to cheat by modifying their outgoing packets or performing man-in-the-middle or denial of service attacks.

Modern games favor client-server protocols to combat networking threats and preserve determinism over time; decentralized techniques are too slow for real time simulation at scale. High level
client-server game protocol can be understood as a loop with steps conceptually similar to an offline game:

1. **Input Collection:** Clients send messages containing input (keystrokes, clicks or game specific requests) to the server; the server is responsible for discarding packets that appear falsified or malicious.
2. **State Processing:** The server updates all game objects as a function of previous state and new input from clients.
3. **Rendering:** The server updates clients with a packet containing the new game state; clients process this information, update local models, and render graphics.

However, to achieve effective simulation of real time and adequate performance for a specific game, programmers must heavily extend and tune code within these steps. My understanding of these issues has been shaped and informed by several resources, including Gabriel Gambetta’s “Fact Paced Multiplayer” blog series, Glen Feildler’s “Gaffer on Games” multiplayer networking blog, discussions on the Valve Developer Community Wiki. For example, one challenge is that some game states may be too large to be passed in single packets. In real time strategy games, individual players may each control hundreds of elements, where each element is likely represented by a complicated data structure. Therefore, clients must maintain concurrent simulations of the game world (local loops similar to the server’s state processing loop) and be capable of performing deterministic updates given only sparse instructions, rather than new, complete descriptions of every element. A second and more problematic challenge is that network latency – often in excess of 100 milliseconds – is too high for the client input/server response cycle to seem instantaneous. This issue – termed “lag” – is responsible for two major design obstacles faced during networked game development.

1. Players can only react to game events as quickly as their connection allows them to receive updates; games are unfair for players with slow or lossy connections.

Given that packet loss may occur randomly and take unpredictable amounts of time to resolve, it is perhaps impossible to fully eliminate this issue. However, with strategies such as fixing lower bounds for client latencies and computing of game state in lockstep (the game proceeds for all players at the rate of the slowest client’s latency), a basic level of fairness can be enforced. These solutions unfortunately correspond to sacrifices of accessibility and performance respectively.

2. Without having instantaneous access to server updates, clients cannot render graphics in real time.

This obstacle is particularly relevant in fast paced action games; clients must be able to render both the local player and the game world simultaneously. To achieve this simulation, clients must be capable of simulating their own player’s movement (applying inputs and updating the player’s local model in true real time) while portraying the game world as it occurred in the last received update. However, when server side events occur which disrupt the accuracy of client side simulation, clients must quickly perform a “reconciliation” process, accepting the server’s update as authoritative and using memory of recent client inputs to restart the simulation (often termed interpolation) process.

To address the widest range of these obstacles, I built a basic action roleplaying game. Rather than settling with turns and lockstep, I wanted to experience the full challenge of simulating real time interactivity, and to understand how FP could be helpful or prohibitive during the development process. Furthermore, by using the web as a client platform, I show that FP is not exclusive or unwieldy, but rather a powerful and applicable tool for building interactive systems. Please note that my game is designed for clients using fast connections. While I did not perform testing to identify specific connection speed requirements or performance thresholds, fast wireless connections and wired broadband connections suitable for commercial games worked best for my testers.
Design and Implementation

My application consists of both a .NET server application running purely F# and a JavaScript web client optimized for Google Chrome. I will first describe the design and implementation of the application’s networking infrastructure, which is built on the .NET signalR library (see ChatHub.fs and MessageHandler.fs). I will then proceed to discuss implementation of game code on the serve (see Startup.fs and Game.fs). I will finally discuss the client (see Index.html and scripts/game.js).

Networking Infrastructure

Configuring a high frequency messaging solution was a prerequisite for proceeding with game development; my first objective was to write basic server and client code using the signalR Hub class and to test the functionality of web sockets. It was easy to extend the signalR hub class with member functions that allow the server and client to call functions on one another. I demonstrated this with a chatroom example, which allows clients to call the “send” function on the server, taking username and message strings as arguments. When this function fires, it calls the “broadcastMessage” function on each connected client, passing the same arguments. Clients then append the new message to its chat log using JQuery. Examples from todobackend.com were helpful in understanding how to mount general .NET middleware components onto barebones F# server startup file.

While the .NET Hub maintains a list of connected clients, it is necessary to implement a separate data structure for associating additional information with each client. While Microsoft Support resources offer guidance on integrating signalR with C# applications, similar resources for F# developers do not exist. Thus, while it was clear that signalR hub functions execute asynchronously, and that I could not simply update a mutable dictionary due to thread conflicts, it was not immediately apparent how to proceed. After a lengthy trial and error process, I discovered the F# Mailbox Processor. This processor is a standard F# class which offers a “dedicated message queue running on its own logical thread” (wikibooks.org). The processor executes a recursive loop which stalls until new messages are enqueued; pattern matching is then used to identify message type and execute corresponding code. Using the mailbox processor, I created a thread safe API layer between the signalR Hub and a .NET <string, string> Map. This provided for association of usernames with Hub connection ids, and was the starting point for developing game related server/client interactivity. Changing this map to store an original record type User with several fields including a concurrent queue of type Input (another original record type for storing client keystroke information) allowed for buffering of user.

Server Implementation

Server game state exists in an infinite recursive function “gameLoop”, whose arguments include an object of type State, and a concurrent queue of type ServerEvent. The State object contains lists of type Player and type Ai; these classes are used to model client controlled game characters and server controlled enemies. The ServerEvent queue is populated during MailboxProcessor execution; ServerEvents are discriminated union types that instruct the game loop to create, destroy or respawn players, triggered by Hub functions and client connection/disconnection events. During each execution of the gameLoop function, the function serverUpdate processes all available ServerEvents, returning an intermediary copy of the state object. Then, the function gameUpdate is called, which returns another version of State with updates made to each Player and Ai.
gameUpdate is the most significant server function, and demonstrates valuable modularity and security improvements gained by using F#. This function maps a sequence of functions onto the Player and AI lists provided in the state object using the F# forward pipe operator. Each Player and AI updated in these mapping sequences is responsible only for updating itself. Because each function receives only a static reference to its previous self and specific elements of the outside game world, it is impossible for functions to mutate other objects and cause unintended side effects. Because elements must still affect one another in a game world that models interactivity, an alternative strategy is used. Before element processing, the gameUpdate function spawns GameEvents which exist for the duration of a gameUpdate execution, and serve as instructions to elements that an interaction is occurring. These events are also created by observing exclusively static references. During each Player or AI element’s update step, if event instructions exist that are designated for that element, an additional function is applied. While my game’s use of events is limited to the example of combat, these instructions could be used to implement arbitrary interactions between character elements and the environment. Design of this entity and event system was influenced by John Carmack’s 2013 keynote speech at QuakeCon, as well as general information about “Entity Component Systems,” which briefly outline the theoretical composition and advantages of ECS design patterns in game development using FP.

At the end of every gameLoop execution, the state object is condensed into a snapshot form, which is broadcasted to game clients. While my implementation uses snapshots that contain every elements’ state, the nature of these snapshots could vary drastically for a given games’ requirements.

Client Implementation

After connecting to the game server, clients simulate real time player state by optimistically applying user input to a local player model, while other elements are rendered with the most recently available server information. When a player’s local model and server model do not agree – for example, when a player is affected by an event and unexpectedly changes position, loses health or dies – the client accepts the server’s update as authoritative, performs necessary adjustments, then restarts optimistic processing. The client also buffers server updates, which may be received as single frames or groups of frames, allowing for continuous rendering of the game world; if a connection slows to the extent that the client’s frame buffer empties, rendering may appear to stutter. While my client code is not novel (my implementation decisions closely follow advice provided by cited Gambetta and Feildler blogs) it was a substantial programming challenge and consumed a large amount of development time.

Discussion

Obstacles

Both the server and client implementation processes were challenging for various reasons. Despite some prior exposure to functional programming, it was hard to fully conceptualize organization and implementation of many server functions without trial, error and reformulation. Conversely, while most client code was easy to write, debugging and tuning interpolation and reconciliation was extremely difficult.

While working with the .NET environment was pleasant, a lack of available documentation on signalR introduced several major hiccups during my development process. For example, while I was under the impression that the library handled packet ordering in both directions, this was ultimately false. It was not until late in my development process that I discovered ordering issues to be causing discrepancies between client and server simulations. This was quickly resolved by introducing sequence numbers and an additional buffering and sorting step during server processing of user inputs.
Findings

I anticipated that numerous performance related issues would arise with FP as a result of high volume data copying, and that sequences of mapping and events spawning/searching would be too slow and memory inefficient to serve in an application requiring high frequency execution. To my delight, I experienced no such troubles; F# proved fast and efficient enough to support use of exclusively immutable types for all elements within the game loop. My experience with signalR was also positive; signalR websocket connections were stable and fast enough to support the high demands of my application. Though I did not perform testing with more than five concurrent users (the maximum number of concurrent websocket connections allowed on a free Azure instance), I am confident and excited that future web applications will increasingly make use of this technology and expand standards for content delivery and interactivity.

Future Work

While my project achieved the goal of implementing a multiplayer game using F# and the web, it is only a shallow exploration of how FP can be used to simplify and improve game and web development. A second version of my game could include numerous implementations of additional classic game algorithms in a functional style to achieve a variety of additional gameplay and performance related improvements. The highest priority addition would be a spatial partitioning algorithm to reduce current processing and searching overhead associated with spawning events. For example, Players/Ai’s and events could be sorted using a quad tree; they would only search for events in their immediate proximity

While some events may effect players regardless of their position and would require a separate application step, partitioning would greatly reduce computational burden associated with adding collision detection necessary for game worlds with complex environmental features (walls, mountains, etc.). A second addition would involve more advanced treatment of Ai behavior. Without environment features, Ai’s can easily find a line path to a player. However, in order to navigate around obstacles, Ai’s must use a “shortest path” algorithm – usually a variant of Dijkstra’s algorithm – to compute a good path toward a desired position through a graph modeling navigable space. A third addition might be the optimization of server to client messages by removing redundant updates and reducing the overall size of messages via additional serialization and compression

While my game’s state never grows to a size that threatens performance, this step would be necessary for supporting other genres, and promoting my infrastructure as a framework for arbitrary multiplayer games. Finally, ongoing improvement to networking code would be necessary for producing a game with greater responsiveness and sensitivity to user input. Potential improvements may include “adaptive” buffering of inputs and rendering frames, which could modulate server and client processing rates to better fit the latencies of connected players. Additional precision could also be added to hit detection by attempting close estimates of users’ input timing and evaluating collision events in specific historical frame records, rather than processing these events continuously.
Conclusion

Throughout the semester, I have noticed that both disciplines of game development and functional programming are commonly misunderstood and dismissed by individuals with various technical backgrounds. While game development may seem trivial, unimportant, or unintellectual, successful games must overcome numerous challenges within fields of ongoing practical and theoretical significance, including networking, computational geometry and computer security. Likewise, while functional programming carries a reputation for alleged practical unimportance, I have experienced firsthand its legitimacy and efficacy for addressing problems ubiquitous in modern software applications. Though I remain a student in both disciplines, I have greatly benefited from exposure to such a wide variety of challenges, and am extremely happy to have worked on this project.

Acknowledgements

I extend special thanks to my advisor, Professor Ruzica Piskac, for her assistance and encouragement throughout the semester.

Thank you to my family and friends who have inspired me and taken time to test my game.

Thank you to the F# community for providing additional support for the language within Visual Studio and upon the .NET server platform, which helped me to get started quickly and to enjoy ongoing learning.
Appendix

Additional Challenges

While the majority of my development time was spent addressing challenges described in my design and implementation and discussion sections, several other programming and production tasks required noteworthy time and effort.

1. All client code is original. No existing JavaScript game or animation framework was used.
2. Management of audio resources with JavaScript is unintuitive and poorly documented in language specifications. In order to achieve the audio results in my final project, I had to implement a “sound pool”\(^6\), which required unplanned refactoring and hacking.
3. I produced original artwork (sprite sheets) for the project
Referenced Materials: