Software Design and Architecture

Introduction

This document outlines the architecture for a Rust-based motor vehicle simulation running on the Bevy Game Engine. It will have the option to be either connectionless (a single session with no server required), or in a P2P session (a session that is hosted by one of the computers, and synchronized with the other users). While the architecture for this program likely won't be complicated, it is still a good idea to make sure that it is well-defined so that future developers who work on the program are able to understand exactly how that different parts of the program work, making it easy for them to build upon the existing code.

Our project is entirely contained within a single application that runs on Windows, Mac, or Linux. Our project does not depend on any kind of database to function, and all data for the program is stored within the project files. Our program will have the ability to save and load states from files, which will also be stored locally.

Architectural Goals and Principles

This architecture's priority is to provide a fast, low latency connection between users if the users are connected via a session that is being hosted on one of the computers. For singleplayer sessions, the structure is intended to be as simple as possible, because it is only a single application with no subprocesses. With both possible cases in mind, we design the architecture to be robust, understandable, scalable, secure, and complete. We want the architecture to be able to be expanded upon in the future as well as having no security issues as an internet based connection is needed.

System Overview

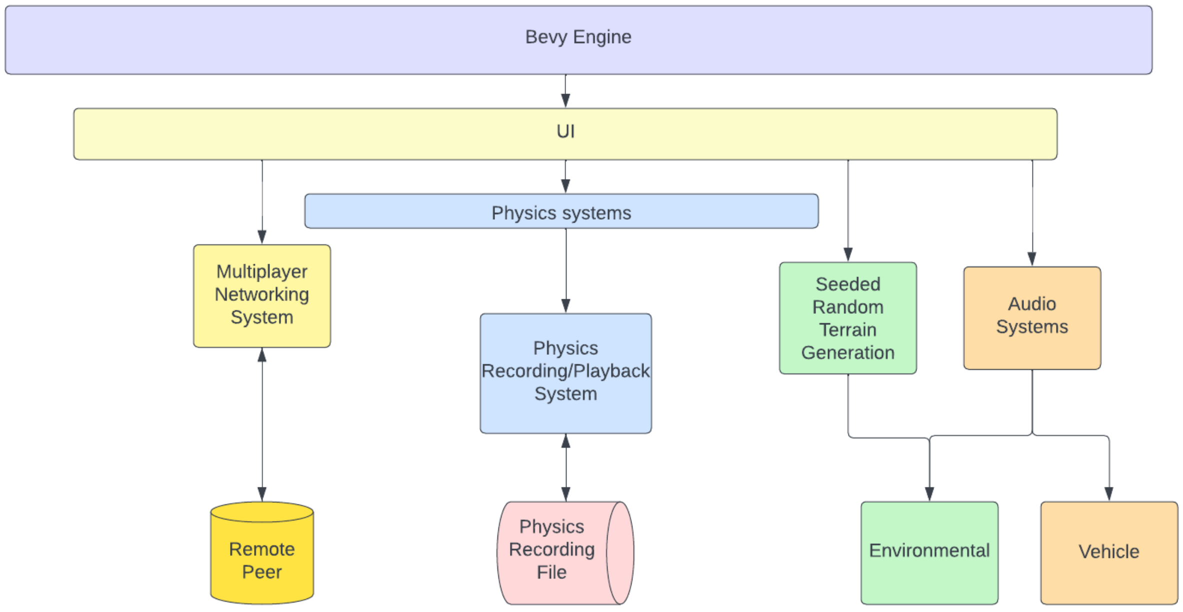
For a local session:

The Application World and Renderer both run in parallel, and are synchronized once per frame. For more information, see this page: https://bevy-cheatbook.github.io/gpu/intro.html.

For a multiplayer P2P session:

The Application World and Renderer both run in parallel on each machine, and are synchronized once per frame, like in a local session.

System Component Overview Diagram:


Bevy Engine
UI
Physics systems
Multiplayer Networking System
Physics Recording/Playback System
Seeded Random Terrain Generation
Audio Systems
Remote Peer
Physics Recording File
Environmental
Vehicle

Architectural Patterns

A model-view-controller pattern would be accurate for this project as the game is broken into components with the camera, physics, and input controllers. Where the camera controller manages the scene views of the vehicle and the input component manages user inputs to control the vehicle and camera view.

We plan to use the client-server pattern for our multiplayer sessions, where more than one user is in the same session at the same time. The server will be hosted on one of the client's machines, rather than on a cloud-based service like Google Cloud.

The server itself that will be hosting the connection between two or more clients is a microservice, so we are using the microservices pattern as well.

Component Descriptions

- Bevy: Underlying basis for this project as it is the game engine that houses the core components for everything such as UI, Audio, 3D Graphics, etc.
- User Interface: Uses Bevy's UI systems to display and interact with content pertaining to Terrain, Physics, Networking, and Audio. Physics parameters, Audio settings, and Network settings will be the primary UI interactions for this project, with additional focus on smaller features such as user interactions, display data, save files, load files, and navigation throughout software.

  [See Interface Design section for more details]
- Physics systems: This includes Bevy's basic physics systems as well as the ones included in Chris's demo. This will allow us to modify the physics parameters of the vehicles in real time using the UI.
  - Recording / Playback System: Record physics vehicle states to load/save to a file on the hard drive.
- Multiplayer Networking System: Transmit packets between peers for our multiplayer functionality. Syncs physics and position data between clients.
- Terrain Generation: Terrain can be randomly generated using UI before starting the game. This would also interact with the Audio system through environmental Audio.
- Audio Systems:
  - Environmental: Specific terrain assets will also have Audio tied to them, such as forested areas having bird sounds, areas with water having sounds of flowing water. All with 3D audio.
  - Vehicle: Simulate a simple X-piston engine in 3D spatial audio depending on vehicle parameters, such as acceleration.

Data Management

All data and assets will be either stored on the user's machine, or the hosting server (P2P sessions only). As for the file structure, we are using the standard Bevy structure, which consists of the root folder, which contains a “public” and “src” folder. The “src” folder contains most of the code, as well as the assets (like images, or models) that will be used in the program.

For saving sessions to be loaded later, we will likely store those files within the “src” folder, and the user will be able to save and load files from that folder using the UI that we will develop for our program.

Data regarding the simulation such as;

- User inputs
- Car properties
- Generated terrain

Will be saved into a custom file for easy recreation and sharing of results

Interface Design

- Main Menu:
    - Settings Menu:
        - Modify audio volume
        - Modify control input
        - Modify video settings
    - Multiplayer Menu:
        - Allow the user to connect with other users, or host a session for other users to connect to
    - Allow the user to choose a seed to generate the terrain and start a game
    - Load vehicle state and terrain from a file
- In-Game Menu:
    - When giving input to the vehicle UI will show basic information:
        - Speed
        - Coordinates
        - Gravity
        - etc.
    - Modify vehicle parameters, there will be text fields for
        - Custom max/min speed
        - Mass of car
        - Gravity
        - Friction
        - Terrain intensity
        - Vehicle acceleration rate
    - Record and Load vehicle state to a file
        - Load/export options:
            - File locations
    - View Controls

Considerations

Security

For the multiplayer features of this project, we are going to implement some important security measures.

When connecting to a host, it's important that no more information than necessary is transmitted to the host, and we will also add a warning for the user before connecting to any kind of server to tell them something along the lines of "Never connect to a server that you don't trust". If any sensitive information needs to be transmitted, which probably won't be necessary, we can look into encryption methods for that.

Plans to possibly introduce the following measures to prevent potential security exploits:

- Password system
- User list
- Whitelisting feature
- Banning feature
- Prevent any files from being downloaded from the other person, and prevent any shell commands from being run from the other person via the P2P connection (for security purposes)

We will encrypt packets with a handshake between peer's to ensure security between the clients. We will need to do more research into this, but mitigating insecurities with network functionality would be ideal.

Performance

With performance in mind, we are using the Bevy Engine which uses Rust, a very efficient language. We intend for our program to be able to run on desktop computers and laptops that are at least reasonably new (i.e. no computers from the 90's), and I don't foresee our program ever having any major performance issues so long as we use best practices in our coding, and keep everything well-documented.

Performance should be prioritized in regards to terrain generation as we would not want the user to have to wait 5+ minutes just to get in the game and start playing. We could monitor and record how performance is impacted by changes that we make to the generation system. In regard to network performance we don't want netcode that will cause too much latency or visual latency for the clients that will make it hard to understand what is going on. Connections aren't always consistent, so we could do some form of client prediction based off of the previous packets of the other player, to make it appear to be 'smoother' than it actually is. Settings to optimize performance over visuals/audio may also be implemented to remedy over stressing.

The two sessions communicate through a signaling server, which uses an unreliable, but fast UDP connection in order to have a connection with minimal latency. The library that will be used for this is Matchbox for Bevy: https://github.com/johanhelsing/matchbox. With this method, our program will prioritize speed over data reliability, because for a program like ours, it is much preferable to have the user's car stutter, or "rubber band" from time to time, rather than always have a latency of 200 milliseconds or more, even on reliable connections.

The simulation should run at least 30 frames per second for most reasonably-new systems. This can be done by optimizing the code for efficiency. There should also be a stable connection of below 60 ms for interstate connections. This can be done by optimizing code and upgrading the server.

Since all that will be required for our multiplayer sessions is simply for two different sessions of the game to be synchronized as two users interact with the same simulation, this seems like the simplest and most practical solution. We also chose this pattern because our program shouldn't be using any microservices, meaning that the server only has to connect with two sessions (or more, if we add the functionality for more than two users to be in the same session at a time).

Maintenance and Support

Maintenance and support may be necessary for another capstone team in the future. Our team members would then be the ones to gather documentation for the project once we are done. If another capstone team will not take the project over we can always publish it as an open source project for community members to keep it alive if there is interest. Documentation of systems and how to add onto them would be vital for either outcome. We can gather feedback through surveys or a public issues board to gather data and prioritize them. We will also let our project partner, Chris Patton, know of anything else that he should be aware of.

Deployment Strategy

Deployment will be through Bevy builds that will be released on our github. We could set up a CI/CD system for building the project whenever there are merges with the main branch. This would utilize github's resources to build the project each time as it runs the scripts.

We will likely keep our project on Github using the Github Releases system, though our project partner, Chris Patton, may choose to release it elsewhere as well.

Testing Strategy

There is a testing framework for the Bevy Game Engine, but for our project, such a framework would not be very applicable. It would take much longer to build an automated testing framework for each feature than to manually test it ourselves, which would defeat the whole purpose of an automated testing framework. So, we will just be doing our testing manually.

Unit Tests:

- Individual tests will first be performed by the team member responsible for development if said feature.
- Once another member has also tested the feature and given the green light, we can begin integration testing.

Integration Tests:

- This can be done by any team member or outside source willing/wanting to test it.
- If everything interacts with everything as planned, we can finalize implementation of the added feature into the current build.
- Any issues will be reported, changes will be made, and integration testing can continue.

Glossary

CI/CD: Continuous Integration and Continuous Development

UI: User Interface

P2P: Peer-to-peer. A type of connection over the internet that forms a decentralized network between individual computers, or “peers”. Each peer has equal privileges, and each can send requests to the other peers in the network. For our purposes, we will be allowing computers running our simulation program to create peer-to-peer connections with each other in order to play in the same session as each other.

Rubber Band: A term used to describe the phenomenon that occurs when a network synced object stutters between locations due to a packet missing in the sequence.