Robert Grier

11/9/2024

Simple Traffic Sim V4

Overview

My project implements a simple traffic simulation in a grid-based city environment. The goal is to allow the player to place city objects like buildings and roads to build up the world. Then, vehicle agents will automatically navigate the world from one destination to another.

The game allows the player to view the world from a flying camera. The player can switch between tools for viewing, spawning buildings, spawning roads, and erasing. Each tool has a gizmo that appears when in that mode which follows the mouse. When left clicking, the tools may spawn or remove objects from the world. Objects are spawned onto a 2D grid overlayed on the floor plane. Objects can only be spawned if the grid is not occupied at that location. There is a user interface implemented with “egui” that provides buttons, controls, and stats about the game.

As roads and buildings are constructed and removed, a graph network is updated to reflect the state of the world. This graph network is queried by vehicles to calculate a path from one building to another. Vehicles follow their path by moving across road segments and turning at intersections. The game can be saved and the save state is automatically loaded on start.

Libraries

bevy – this is the main framework of the project. Bevy is modularized into many crates such as “bevy_ecs”.

bevy_infinite_grid – this is a third-party crate that overlays an infinite grid onto the world. This is helpful for debugging my grid system.

bevy_mod_raycast – this crate provides a simple raycast system for bevy projects. I use this on vehicle objects to detect other cars and slow down they are too close.

bevy_egui – integrates “egui” with bevy projects. This is used to create the user interface. I also use “catppuccin-egui” to customize the appearance of the egui elements.

serde, serde_json – I use the JSON serialization and deserialization crate to save and load the state of the world from a file.

rand – Used for randomization of speeds, spawning, etc.

Camera

The camera is a top-down camera that can be rotated, panned, and zoomed. The camera has detailed controls described below for using a mouse or just the keyboard.

Grid

The grid is represented as a 2d array of Option<Entity>. Queries to the grid take the form Result<Option<Entity>> where the result will be an error if the request is out of bounds or not valid. The option is “none” when the grid is empty, and it contains “some” entity when occupied at that location. I am using a crate to draw an infinite grid for debugging. There are helper classes for GridCell and GridArea to represent individual cells and areas of cells on the grid.

User Tools

Tools are swapped between using a state machine in toolbar.rs. The building tool spawns scaled cubes to represent the buildings. The road tool allows you to drag after clicking to determine the size of the road. The road tool automatically creates intersections, extends roads, and bridges roads based on adjacent data in the grid. The eraser tool deletes buildings and roads. The view tool just disables all other functionalities.

Graph

The graph connects buildings, roads, and intersections so that the city can be navigated. The graph observes spawning and deleting events to keep everything up to date. The graph is visualized with gizmos to show all the connections.

Vehicles

When spawned, vehicles calculate a path from one random building to another if a path is possible. They query the graph to do a simple DFS for the path. If part of the road path is edited after spawning the vehicle, the vehicle is notified and deleted. Vehicles drive freely and follow the desired lane on the current road. The desired lane is based on the upcoming turn direction in their search path. Vehicles must turn manually at intersections to end up in the correct direction and lane. In addition, vehicles use the “bevy_mod_raycast” crate to detect each other with a raycast. This allows them to slow down when they are close to each other, preventing a collision. This behavior causes traffic to build up when the roads become congested.

User Interface

The game integrates the “egui” system using the “bevy_egui” crate. I used this to add a very basic user interface for switching between tools, describing controls, and showing stats about the game state. Although the immediate mode GUI is not great for a game application, it is a good start for getting some basic info on screen and can be used for development tools. Interestingly, the engine’s native GUI system is not particularly powerful or concise, so I am avoiding it.

Saving

The game state can be saved to a file using the Serde Json crate. This generates a Json file containing all the data needed to recreate the city built by the player. When the game starts, the city data is loaded from the file and updates the world. The graph and grid components are idempotent, which allows the loaded items to be re-spawned easily. Currently, only one save file can be used.

How to play the demo

Start the game and view the saved city loaded from the save file. This city can be extended by the player. Place some roads using the road tool. Intersections will be generated on adjacent roads (in most cases). Place some buildings adjacent to the roads using the building tool. Erase buildings and roads as needed. Observe the graph update by toggling the visualization with [H]. Spawn a vehicle with [P] and observe that it generates and follows a path from one building to another. Press [V] to view the path and debug the ai behavior. Press [L] and vehicles will begin spawning automatically. Observe that vehicles detect each other and slow down to prevent collisions. The vehicles should also change lanes based on what turn they need to make. Press [F5] to save any changes to the city structure.

Observations of Rust

One positive of rust is the ability to represent null objects in a safe way with options and results. I use options in the grid system to represent if a cell is occupied or empty. I use the result type to represent invalid arguments such as the grid cell being out of bounds. A downside of error checking is that it is required everywhere and makes the code hard to read. It prevents common errors when developing, but once the system is correct, all the error checking is left over and makes it more difficult to identify the meaningful parts of the code.

The immutability of variables by default is beneficial because it leads to simpler code that only modifies what it needs. The programmer must intentionally make something mutable only when necessary. This makes it easier to think about when something is safe to use because the compiler ensures it is either a read-only shared reference or an exclusive mutable reference.

Enumerations are very powerful compared to C style enumerations. It is really great to be able to add methods and rich behavior to Enums such as my direction class, where you can say *Dir::East.inverse() == Dir::West*, for example. This allows for data driven code in-place of switch statements when working with Enums.

The best part of Rust, when compared to a systems language like C++, is the cargo system. Cargo allows for dependency management, building, and linting in a perfect way. It is always correct, and it is usually fast. One issue I had is when VS Code is using cargo to scan things, there is some sort of lock system that prevents you from building in a separate terminal until VS Code is done.

Observations of the Bevy Engine

The Bevy Engine implements some impressive modules with a smart design philosophy. However, I am unsure what the true payoff really is from this approach and if it is practical for the creative process of game development.

The entity-component-system is dogmatic. This means that each function added to the engine has to be a system, where all references needed are listed as parameters to the function. This parameter list tells the engine which references are mutable or immutable. You can request resources, query entities, and get other utilities like events. The main benefit of this is that the engine can identify all the systems in the game, identify how they mutate state, and can create a schedule to maximize parallelism. This is possible in any language, but Rust guarantees that the system author follows the rules of mutability, and that everything runs smoothly.

One issue with this is verbosity. If many references are needed, they all need to be added to this parameter list. This makes it difficult to write clean code because extracting smaller functions from the system requires passing around many parameters. You can break up the system into several smaller systems instead. These systems can communicate with each other using events and indirection tools. This feels good to write and makes this code seem more decoupled. Unfortunately, the simplification of the decoupled systems did not seem to make up for the added complexity and reduced readability of all the intercommunication and scheduling of these systems.

The parallel scheduling aspect of Bevy is very appealing. I liked how much control you can have over the schedule of the systems. The ability to conditionally run systems based on a state machine (which you can define as an Enum) is elegant. I also liked how each module of code can be organized into a plugin which individually attaches systems to the application. This encourages more decoupled systems, and I was able to write modules that can be commented out without breaking the game. For example, I can remove the entire UI with one line of code and I can be confident that the rest of the game will still work. It is also nice to have the structure of the plugin described entirely in the plugin implementation block.

One concern is that the schedule must be modified substantially to ensure determinism and ordering. The need for specific ordering in a game context, such as reading input and spawning something on the same frame, may limit the benefits from parallel scheduling such that organizing the entire project around the rigid ecs becomes more of a burden than a benefit.

Overall, I learned a lot from the project because it is a completely different approach to software architecture than what I am used to. I was exposed to some ideas that challenge preexisting ideas in something like object-oriented programming. However, I would argue that Rust and Bevy could benefit from more ergonomics, especially when it comes to prototyping and experimenting with creative applications like a game.

Controls

Save the game – [F5]

Spawn – [P]

Toggle automatic spawning – [L]

Toggle graph visualization – [H]

Toggle grid visualization – [G]

Switch to view tool – [`]

Switch to building tool – [1]

Switch to road tool – [2]

Change road orientation – [Tab]

Switch to eraser tool – [3]

Adjust tool size – [F/R]

Camera rotate – (Middle Mouse) or [Q / E] or [Left Control] + (Left Mouse)

Camera panning – (Right Mouse) or [WASD] or [Left Alt] + (Left Mouse)

Camera zoom – (Scroll Wheel)