Engine Design

February 7, 2020

1 Game Loop

This section details one run through the game loop, including calculating the next state, applying user inputs, notifying peers, and drawing a frame. The following steps happen right after the JS event loop calls the gameLoop function.

1. **State Changes Proto Created**: An empty EngineState protobuf is created to represent the state changes that have happened due to peer inputs, user inputs, and NPC inputs. When adding entries to this proto, state may be copied from the last frame's state to ensure that the minimum size chunk of state is set. See item 7 for more info on minimum chunk size.

2. **State Changes from Peers**: Buffered state changes from peers are applied to the state changes proto. These state changes are filtered first to make sure only UUIDs owned by the peer are being changed.

3. **State Changes from User**: Buffered user inputs are applied to the state changes proto.

4. **State Changes from Owned NPCs**: Buffered NPC inputs are applied to the state changes proto.

5. **Apply State Changes to Engine**: State changes in the State Changes proto are applied to the engine.

6. **Step the Engine**: The engine is stepped forward by the amount of time that has passed since the last frame. This step also retrieves the engine's state as a proto.

7. **Change Detection for Peers**: Any changes between this frame's state and last frame's state that could not have been predicted by peers are retrieved from the current state. These changes include changes from the player's inputs, changes from the inputs of owned NPCs, and other changes that might be predictable, but that we report anyway, such as when the player's ship takes damage or dies. Predictable changes, like changes in position and velocity, are not included.

The minimum chunk of state change that can be sent is the full state of a space object. The result of this change detection can only include the states of owned objects, such as owned NPCs and the player's ship.

8. **Notify Peers**: Send the changes to all peers.

9. **Give NPCs Current State** NPCs who want the current state (usually those whose state just got applied) are given a copy of the current state. They're given a copy because they might try to change parts of it they shouldn't change.

10. **Draw a Frame**: The state of the system that the player is in is given to the frame Drawer, and it draws the current frame.

2 Serialization

Protobufs are used for serialization of the game state. They comprise the interface between the Engine object and the rest of the game via the *setState* and *getState* functions, and they are the format in which state changes are communicated between peers over the wire. The entire state of the game is represented in the *EngineState* proto, which is a tree structure that includes Systems, Ships and other types detailed below.

2.1 Why proto3

We use proto3 since it is recommended for new projetcs. This decision might seem strange since we are going to be sending a lot of deltas for state updates, which proto2 is better at. Proto2 lets you tell if a primitive field was set to null or not set at all, which is important if you want to, for example, update a ship's rotation without zeroing its shields and armor. However, it turns out that you'd almost never want to update just one property, and sending the full ship state only costs about 300 bytes. Since we only send updates when an unpredictable state change happens, that can peak at 18kBps or 144kbps, given a generous mashing speed of 60 keys per second. Sending the full state also takes care of drift issues that can easily pile up with the delta approach, and if for some reason we really need to omit a primitive field, the field can be boxed. See `https://www.crankuptheamps.com/blog/posts/2017/10/12/protobuf-battle-of-the-syntaxes/#` for more.

2.2 Mapped Fields and SubStates

To avoid sending the entire game state when just a single object is updated, most mapped fields from UUIDs to states allow filling in just the UUIDs that have changed and leaving the other UUIDs empty in the map. To signifiy that the map is not trying to delete the keys that were not filled in, a keys field is included in the proto as well. For example, the *SystemState* proto represents ships with two fields: a map named *ships* that maps from UUID to state, and a list *shipsKeys* that contains the UUIDs of all the ships in the map. When setting the state of a single ship, just that ship's state is included in the *ships* map, but the UUIDs of all ships are present in the *shipsKeys* list so no ships are deleted. To delete a ship, it is simply omitted from the *shipskeys* list. This simplified example does not include discussion of clients' permissions to delete objects, which are handled in the communication section 3.

2.3 EngineState

The engine's state is represented by the *EngineState* protobuf, which has a mapped field from system IDs (not generated UUIDs, parsed IDs) to System-State protos. The field supports SubStates as described above.

2.4 SystemState

A system's state is represented by the *SystemState* protobuf, which has five mapped fields that support SubStates. The fields are as follows:

ships: A map from a ship's UUID to ShipState.

planets: A map from a planet's UUID to PlanetState.

asteroids: A map from an asteroid's UUID to AsteroidState.

projectiles: A map from a projectile's UUID to ProjectileState. Projectiles are usually not sent in the state. They're only included when someone first joins the game or enters a system. Otherwise, each client creates their own projectiles when weapons are fired, and uses deterministic RNG to try to make sure they get fired at the same angels.

3 Communication

4 Engine Organization

The Engine's responsibility is to compute the next state of the universe. It plays no part in drawing, change detection, communication, or player interaction. Its interface with the world is the *setState* function, which sets the current state of the Engine given a protobuf (possibly partial) state, and the *getState* function, which returns a protobuf of the current state of the universe.

What follows are implementation details of the TypeScript version of the Engine. They don't really matter to something that wants to use the engine, as that consumer should just refer to the interface above.

4.1 Stateful Interface

The Engine has lots of moving pieces. To provide a standardized interface and an easy way to set the state, most of these pieces implement the $Stateful$ interface. The interface defines how the state of a piece of the Engine can be serialized and deserialized. It takes a type parameter **State**, which is the representation of the object's state.

setState(s: State): Set the current state of the object from s.

getState(): State: Get the current state of the object as an instance of type *State*.

State is usually a protobuf.

4.2 Engine

The Engine is organized in a tree structure. An instance of Engine owns all the Systems in the universe. A system owns instances of Ships, Planets, Asteroids, and Projectiles.

It is expected that the Engine's *getState* method will be called every frame since it's the only way to get the information needed to draw the frame. Calling *setState* every frame is also expected, but the state given to the engine does not need to be the full state of the universe. Including only those objects that have changed is recommended.

4.3 System

A System represents a contiguous play area. Anything in a System can be travelled to by flying toward it. Travel between Systems usually requires a jump, and can not be achieved by flying in a direction, no matter how long a player flies for, as they will just loop around to the other side of the system they're in.

Things that can be in a System include Ships, Planets, Projectiles, and Asteroids. Things in the System usually own instances of SpaceObject, which represents an object in space that can move around, be drawn on the screen, and be collided with.

A System also owns an instance of a collision library, and the System registers things with the collision library when they are created or move into the System, and removes them when they leave. Things that support collisions implement the Collidable interface.

Everything in the System implements the Stateful interface.

A System is also responsible for creating Projectiles. It has a factory for creating projectiles that takes the following properties for the new projectile:

weaponType: The ID of the weapon the projectile came from.

target: The projectile's target, if any. A UUID indicates a single target. In the case of a submunition that fires at a random target, the weapon that is subbing is responsible for selecting the random target.

position: The projectile's initial position.

angle: The projectile's firing angle.

sourceVelocity: The projectile source or parent's velocity. Certain types of projectiles take on the velocity of their source plus their own firing velocity.

A System has a s

4.3.1 Moving Between Systems

This section is TODO pending being able to actually test the different ideas I have about how to solve this. What follows was a partial first attempt, but it should really be looked at again.

Ships can move between systems. This process is a lot more complicated than it might initially seem, involving several asynchronous steps and coordination between peers. The steps are detailed here:

1. **Initiate Jump**: A user commands their ship to jump from System A to System B. All conditions for the jump are met, and the jump begins.

2. **Begin Jump Animation**: The ship's jump process starts. This might include the ship stopping to get ready for the jump, or the ship might just immediately leave the system. This step includes all animation up to the point at which the ship is no longer in System A, and it can be interrupted if the user changes their mind or the ship dies. This step also registers the callback that gets called in step 5.

3. **Request State (async)**: The client requests the full state of System B from the server or another peer in the system. The client also subscribes to receive state updates to System B, but stores them instead of directly applying them to System B. This step also registers the callback that gets called in step 6.

4. **Notify Peers**: The ship's current state is used to build a copy of the ship in System B. This copy has the "hidden" property set so that it doesn't get rendered or updated. [1]

5. **On Complete Jump Animation**:

[1]The reason we use a property on the ship to indicate that it's hidden instead of storing the ship in a separate map specifically for ships that are entering the system is because the "hidden" property already exists for use with fighter bays. It would also be a pain to communicate in a state update that "a ship stored in an 'enteringShips' map has moved to the 'ships' map" instead of "a ship was deleted from the 'enteringShips' map and another ship was added to the 'ships' map".

6. **On Receive System B State**: Eventually, and as a non-blocking callback, the client receives System B's full state and applies it to the system. Then, all the stored state updates are applied to System B. New state updates are no longer stored, and instead get applied to the system directly as is normal. Once all the stored updates are applied, the client adds System B to its list of active systems (systems whose next states get computed every frame).

When a ship moves from system A to system B, it plays its jump animation. During the jump animation, the client fetches system B's full state from the server and subscribes to system B's state updates. While fetching the state, and right after the ship's jump animation causes it to leave system A, the engine serializes the ship's state, changes a few properties on the state like position and velocity according to the jump that was made, and gives it to system B to build. System B builds a ship from the state, and waits to receive the rest of the state from the server if not already present. System B is paused for the duration. After the rest of the state is received, the client starts rendering system B, and

4.4 SpaceObject

A SpaceObject represents an object in a system. It can move around, and its serialized representation can usually be drawn on the screen. Most things that appear in a system are SpaceObjects. Notable exceptions are the starfield, target reticles, and beam weapons.

SpaceObjects can have Outfits (because Ships and Planets need Outfits). An Outfit serves a few functions. It can

4.5 Ship

A Ship's Outfits are stored in its Equipment object.

4.6 Equipment

An Equipment is a collection of Outfits and, consequently, Weapons. Its interface has a function that, given the properties of a ship, returns a set of properties modified according to what Outfits are installed. It allows its owner access to its weapons via *setState*.

4.7 Outfit

Outfits are owned by Equipment. Outfits have various effects on a ship's properties, such as speed, turn rate, and energy capacity. Outfits can also have weapons associated with them. The interface used by outfits

4.8 Armament

An Armament is a collection of Weapons. It is usually created by, and associated with, an instance of Equipment. However, a Weapon that has submunitions creates an Armament without an associated Equipment since the submunition field in a weapon definition lists weapon IDs instead of outfit IDs.

4.9 Weapon

Weapons are owned by Armaments. Through its updateState function, a weapon can be set to fire at its fire rate. A weapon also exposes a function to fire a single projectile at a specified angle for use with submunitions.

It should be noted that Projectiles are not sent between peers, so for consistency, a weapon uses a counter-based random number generator for any random properties it has, like firing angle. This count is included in the state of the weapon, and when a peer learns that a weapon is being fired, it also learns the count at which that weapon started firing. Submunitions of a projectile fired by a weapon use a random number generator seeded by the random number used in the original projectile's inaccuracy calculation. That way, an arbitrary nesting of submunitions remains free of patterns while being deterministic on all peers, and the times at which a Weapon's count-based random number generator is queried are exactly those times when the Weapon is fired.[2]

In the first version of the engine, beam weapons do not support submunitions nor are they supported as submunitions. Beam weapons also do not require projectiles, but do implement the Collidable interface.

4.10 Projectile

Projectiles are fired by weapons. A weapon receives projectiles to fire from the System's projectile factory, which is passed to a Weapon when it is built and changed whenever the Weapon changes systems. This factory is replaced when a Weapon changes systems.

[2]We don't use the count-based random number generator for submunitions because if a weapon subs into itself (with a sub limit of, say, 10), all projectiles that result fire their subs at the same time, which makes it difficult to reason about which ones get which random numbers. You'd have to guarantee the order in which subs and subs of subs are fired, which is difficult to reason about. It's much easier to turn the initial random number into a bunch more numbers via seeded generators.