Ricochet: A DOTS-powered Twin Stick Shooter

What is Ricochet?

Ricochet is a game prototype for a twin stick shooter in which your bullets persist! You pilot a single starship stuck in an everlasting dogfight. You must dodge asteroids, space junk, and enemies that threaten to destroy your starship, but you must be careful! Every shot that does not hit your enemy will persist and may spell your doom in the near future!

What are the core mechanics of the game?

Ricochet consists of five main actors:

1. **Players:** each player is responsible for controlling a single starship. Currently, we are limited to one player for simplicity. More can be added though!
2. **Enemies:** each enemy is like a heat seeking missile in search for your starship. You must destroy them before they reach you!
3. **Obstructions:** various obstructions float throughout the battlefield. You must avoid them at all costs!
4. **Bullets:** ricochet throughout the battlefield until they either destroy an enemy or damage your ship. Be careful when you fire your weapons, you'll never know if that bullet will come back for vengeance!
5. **Collectibles:** collectibles serve as a way to demonstrate the trigger system, but they aren't included in the final prototype at the moment.

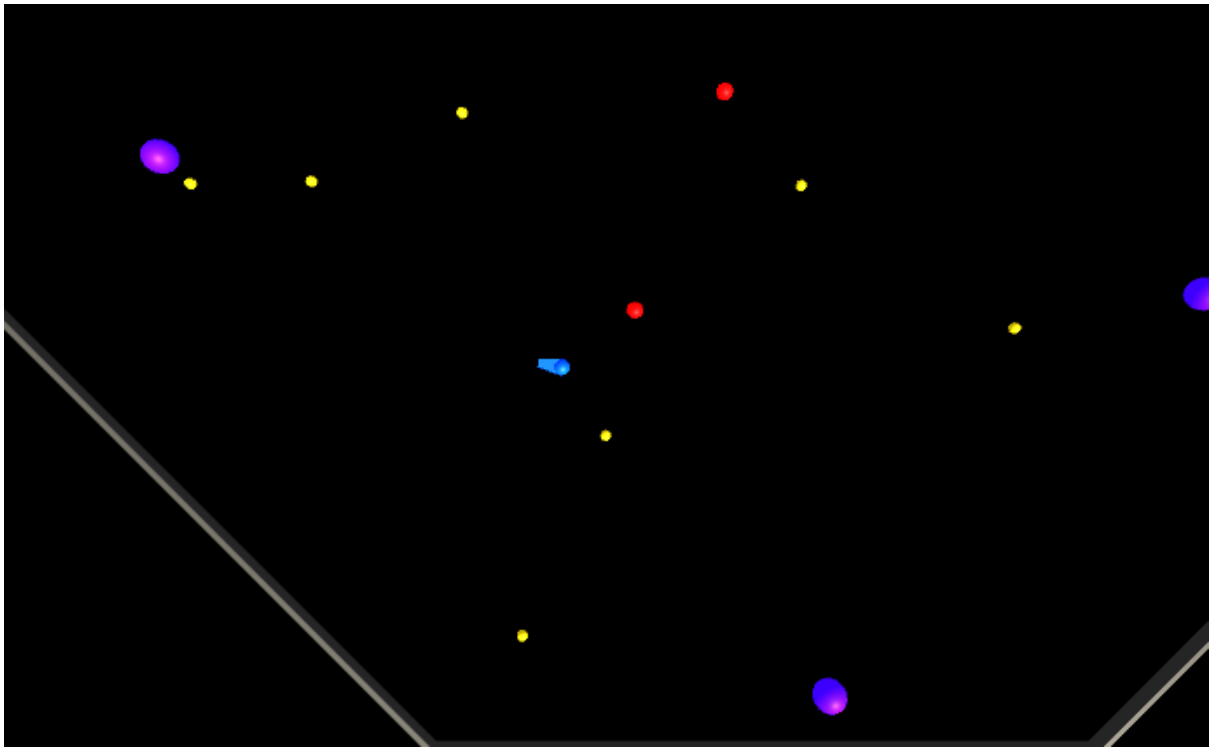

Figure 1: a screenshot of Ricochet in action. The blue actor is the player. The yellow actors are the ricocheting bullets. The red actors are enemies. The purple actors are obstructions. Finally, the grey actor is the level's boundary.

What is the end state for the game?

You need to survive for as long as possible. If you take four hits from either bullets, enemies, or obstructions, then your ship will be destroyed and it's game over.

How does it work?

Ricochet has been implemented using Unity's Data-Oriented Tech Stack. It includes the following systems:

1. **CollectionSystem:** manages the collection of collectible items.
2. **CollisionSystem:** tracks interactions between actors and collision or trigger volumes.
3. **ConstantForceSystem:** ensures entities translate at a constant rate (i.e. we're using elastic collisions here.)
4. **DamageSystem:** manages interactions between things that can apply and receive damage.
5. **EnemySystem:** manages the behaviour of enemies within the game world.
6. **FreezePositionSystem:** ensures entities exist within spatial confines (e.g. on a plane along the x and y axes.)
7. **GameStateSystem:** tracks and enforces the game's end state.
8. **MovableSystem:** implements any kinematic-style movements.
9. **PlayerSystem:** parses player input and translates that into game actions. This system is very simple, but it can be extended.
10. **RandomGenSystem:** manages random number generators for each thread used by the game.
11. **SpawnerSystem:** spawns entities according to a set of restrictions based on spawn points.
12. **SpawnPointSystem:** determines if a spawn point is occluded or not.
13. **WeaponSystem:** manages any projectile attacks made by an actor.

Each system is generally backed by a component. We have regular components as well as buffers. Typically, these buffers are used for tracking collisions and triggers. Furthermore, we have component authoring classes as needed.

What improvements can be made?

This game prototype was implemented on a strict time budget and therefore many improvements could be made. For now, let's keep in mind that this is a simple prototype and we're not concerned about things like UI etc. We're really just capturing the essence of the core mechanics here. So, here's what we're left with:

- Implementing a proper input system. Currently, we're restricted to the firs available game pad.
- Instead of destroying bullets, enemies, and players we could reuse them instead of spawning replacements.
- We could simplify the ConstantForceSystem to only enforce velocity after a collision has been recorded. Given the lack of any sort of drag, we can rely on the fact that linear and angular velocity will be maintained.
- Add a proper flow for setting up and tearing down the game. Currently, we're just setting up a single game and restarting after you've hit a game over.
- Make use of agent steering behaviour to make enemies appear more realistic… as if they want to avoid bullets etc.
- Establishing a better bridge between Monobehaviours and entities (e.g. the camera tracking is not ideal. For more information, see FollowEntity.cs and CameraFollowEntityAuthoring.cs.)