GAME PROGRAMMING

Digital Assignment:
Dronie: FPV Drone Simulator

Submitted by:
Venkatesan M
22BAI1259

VIT®
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

Submitted to:
Dr. Graceline Jasmine S

Project Overview

This is a drone simulator I built in Unity. The main goal was to see if I could create a realistic flight simulation by connecting a real RC radio controller (a Radiomaster Pocket) to Unity.

The core of the project is a C# script `DroneController.cs`. This script acts as the drone's "flight controller." It takes the processed stick inputs from the radio and turns them into physical forces and torques. I then apply these to a Rigidbody component, allowing Unity's physics engine to simulate all the movement.

The simulator accurately models three different flight modes (Angle, Horizon, and Acro), incorporates essential input processing features such as expo curves and dead zones (to provide a realistic feel), and features a simple On-Screen Display (OSD) to display flight data.

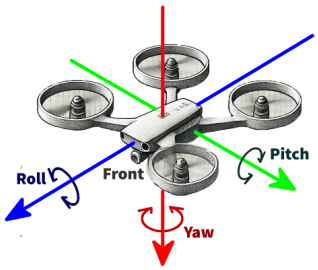
Background: What is an FPV Drone?

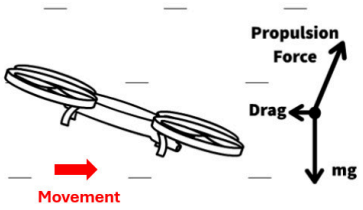
A "drone" is just a quadcopter. An FPV (First-Person View) drone is a special type where the pilot flies by looking through a camera on the front of the drone, similar to VR Goggles.

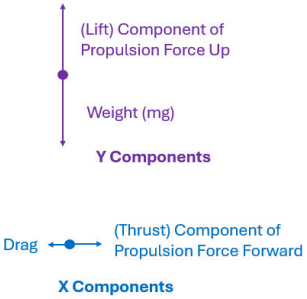
- Camera Drones (like a DJI): These are "flying cameras" designed to be stable and easy to fly for shooting video.
- FPV Drones (Freestyle/Racing): These are built for high-speed, acrobatic flight. The pilot wears goggles that get a direct video feed from the drone. It feels like you're in the cockpit. This project simulates this type of FPV drone.

The Physics of Quadcopter

A quadcopter flies by changing the speed of its four motors. It doesn't have rudders or flaps. All movement comes from a balance of thrust and torque.







1. **Throttle**: Controls altitude. All four motors spin faster or slower together to make the drone go up or down.
2. **Yaw** (Rotation): Controls spinning left or right. The two motors spinning clockwise speed up, and the two motors spinning counter-clockwise slow down. This creates an unbalanced rotational force (torque) without changing the total lift, so the drone spins.
3. **Pitch** (Forward/Backward): Controls tilting. To pitch forward, the two back motors speed up and the two front motors slow down. This lifts the back of the drone, causing it to tilt forward. The drone's main thrust now pushes it forward.
4. **Roll** (Left/Right): Controls tilting side-to-side. To roll right, the two left motors speed up and the two right motors slow down, lifting the left side and tilting the drone to the right.

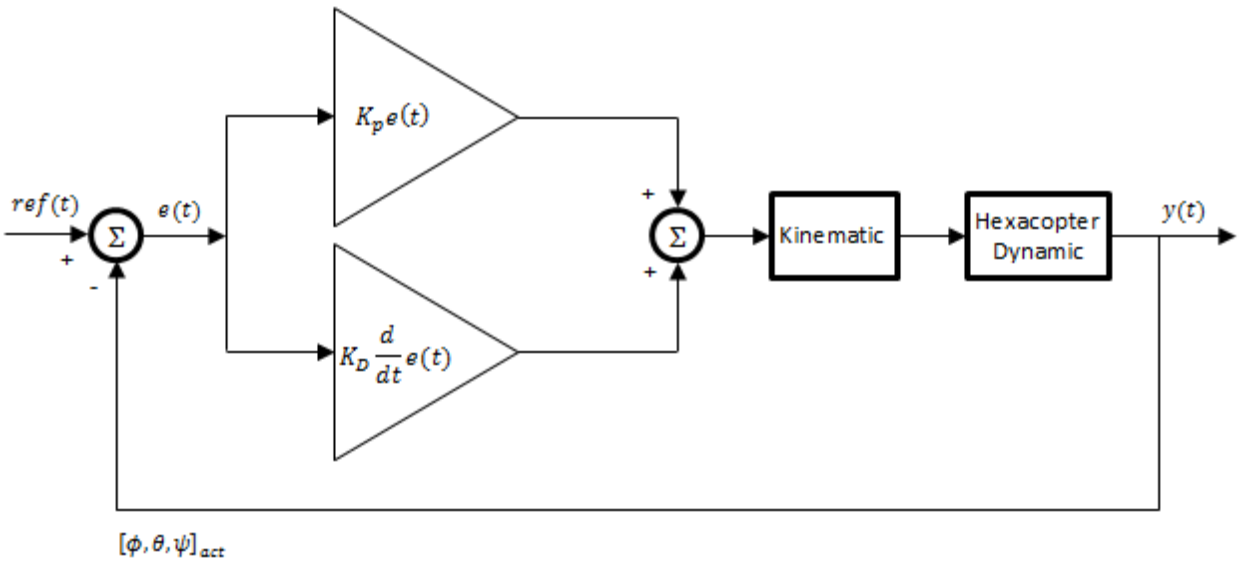
The "Brain": Flight Controllers & PID/PD Loops

A quadcopter is inherently unstable. It needs a computer (a Flight Controller) making thousands of adjustments every second just to stay level.

This flight controller runs a PID controller. This is a control loop algorithm that's standard in robotics. It works by looking at an error and calculating a correction. It tries to make the drone's current state match the target state (what the pilot is telling it to do with the sticks).

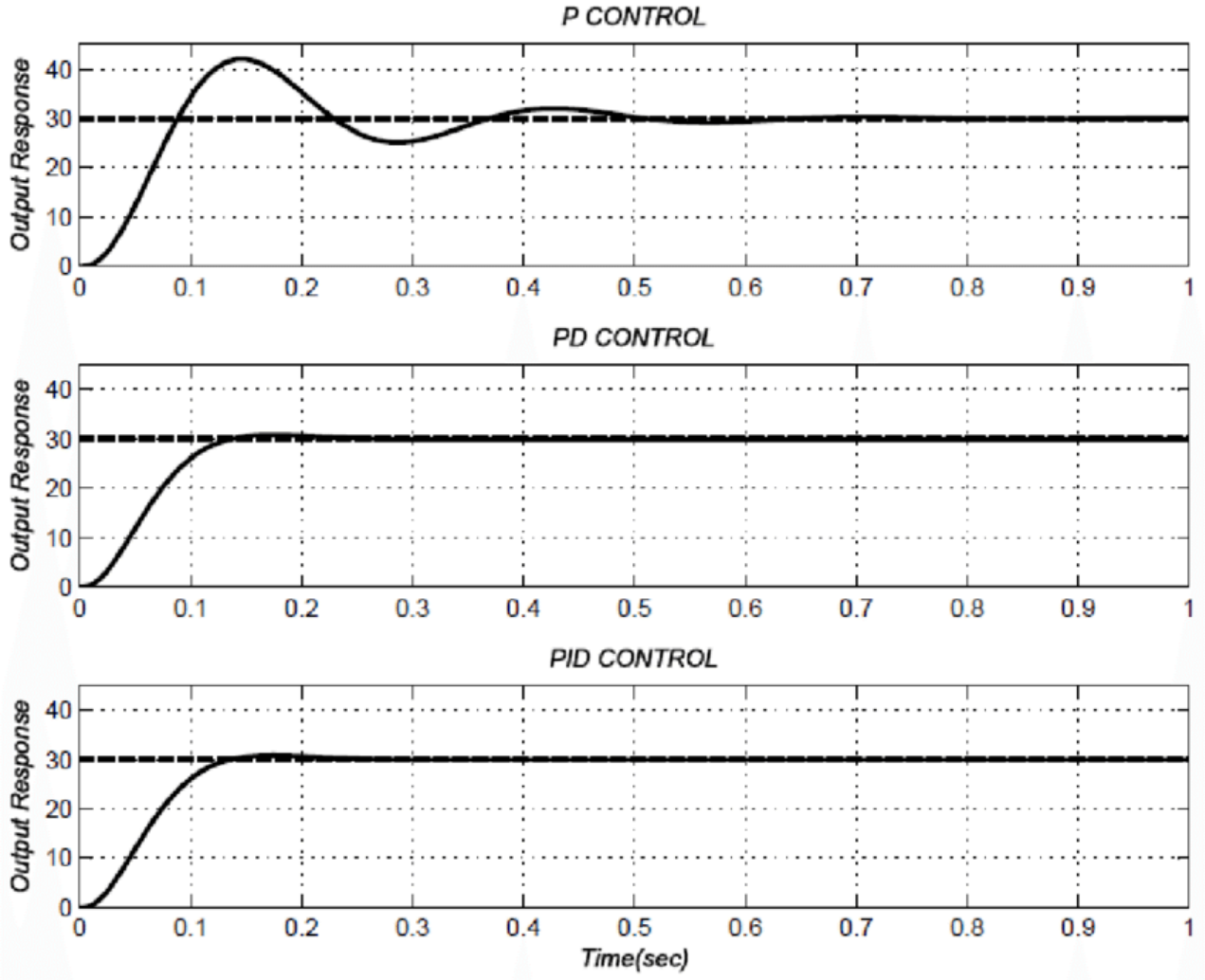
PID stands for:

- P (Proportional): Looks at the present error. "How far am I from my target angle?" The bigger the error, the harder it pushes back. This is the main stabilisation force.
- I (Integral): Looks at the past error. "Have I been drifting for a while?" This part corrects for small, constant errors, like a slight wind. (We did not implement this part, as it's complex and not strictly needed for a basic sim.)
- D (Derivative): Looks at the future error. "How fast am I approaching my target?" This acts as the "brake" or "damping." It stops the P-term from overshooting the target, which is what causes a "wobbly" or oscillating drone.



PD Controller Block Diagram

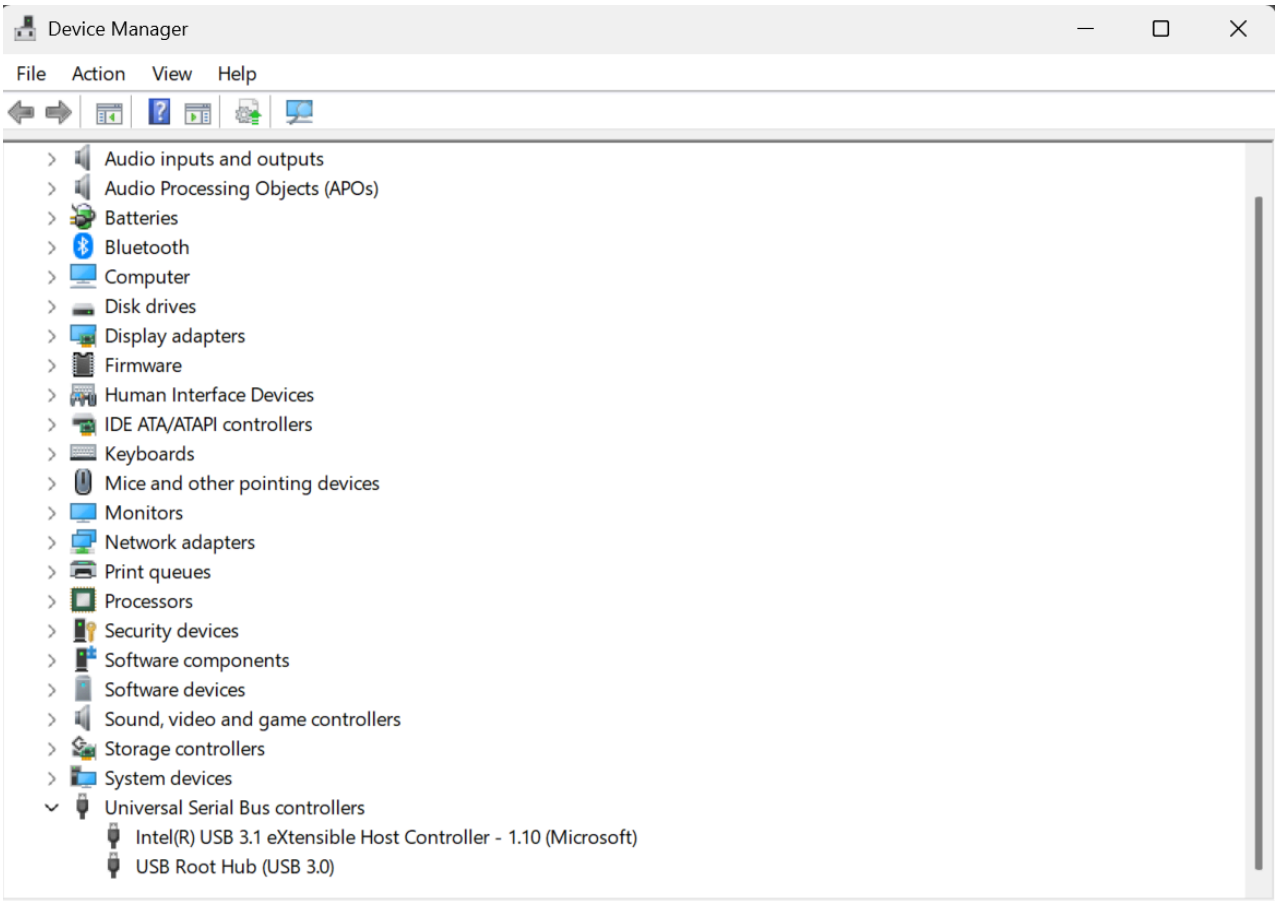
In our project's Angle Mode, we built a PD Controller. We use the P term (stabilizationStrength) as the main self-levelling force and the D term (stabilizationDamping) to act as the brake, making the flight smooth.

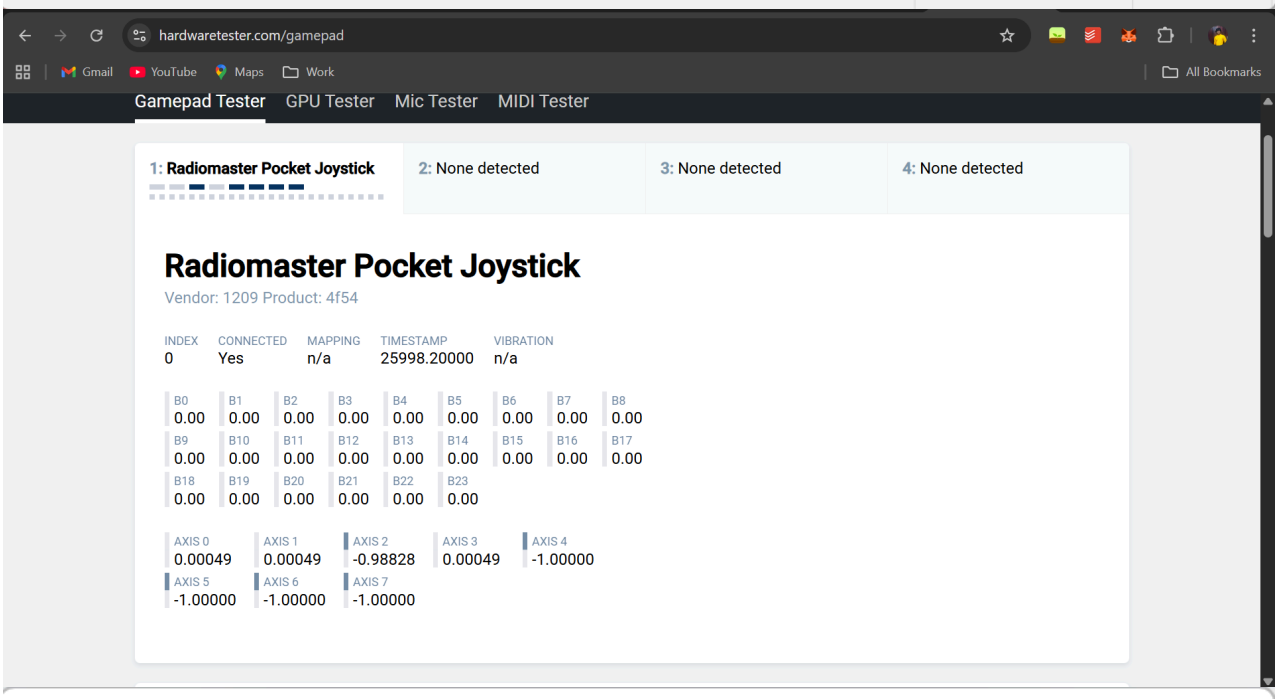


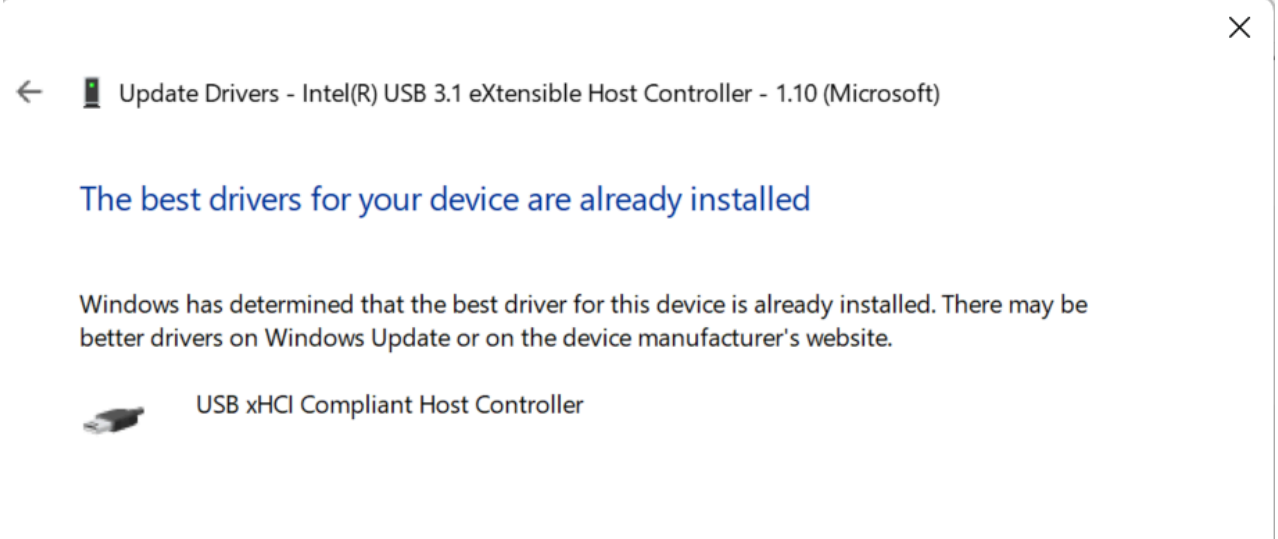
Simulator System Architecture

The data flow for the simulator is:

1. Hardware (Input): User moves the sticks on the Radiomaster Pocket.
2. How the Controller Connects (HID): The radio is plugged in via USB in "Joystick" mode. The OS (Windows/macOS) just sees it as a generic Human Interface Device (HID). No special drivers are needed (Depends on OS).
3. Input System (Unity): Unity's Input System Package reads the raw joystick data. My Input Action Asset (DroneControls.inputactions) maps the raw axes (like Stick/X, Z) to my own logical Actions (Roll, Throttle).


Device Manager
File Action View Help
Audio inputs and outputs
Audio Processing Objects (APOs)
Batteries
Bluetooth
Computer
Disk drives
Display adapters
Firmware
Human Interface Devices
IDE ATA/ATAPI controllers
Keyboards
Mice and other pointing devices
Monitors
Network adapters
Print queues
Processors
Security devices
Software components
Software devices
Sound, video and game controllers
Storage controllers
System devices
Universal Serial Bus controllers
Intel(R) USB 3.1 eXtensible Host Controller - 1.10 (Microsoft)
USB Root Hub (USB 3.0)

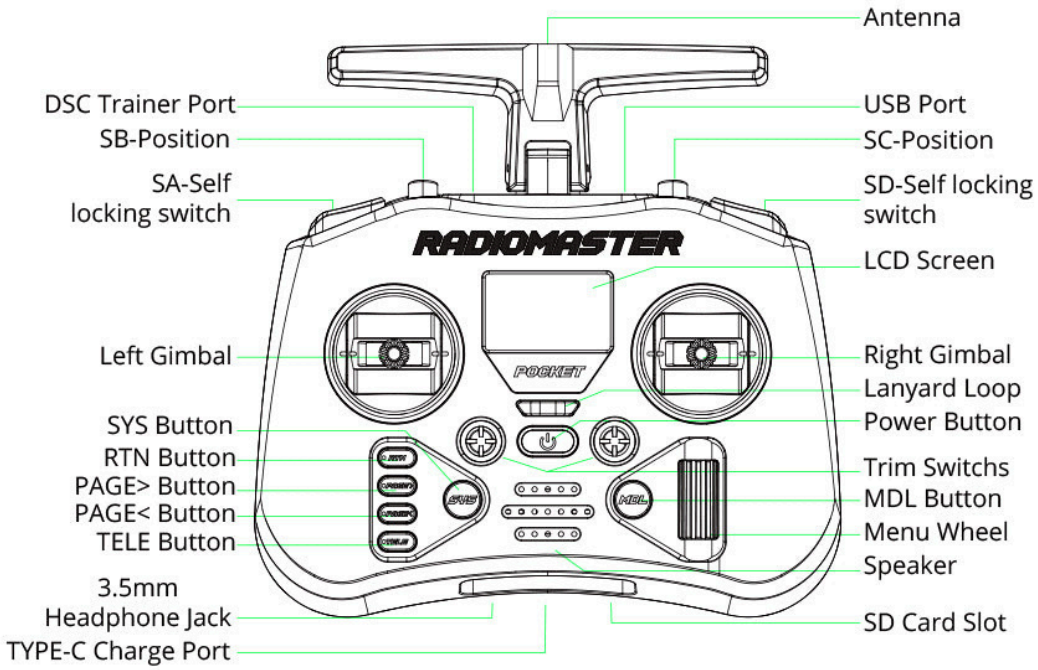

hardwaretester.com/gamepad
Gamepad Tester
GPU Tester
Mic Tester
MIDI Tester
1: Radiomaster Pocket Joystick
2: None detected
3: None detected
4: None detected
Radiomaster Pocket Joystick
Vendor: 1209 Product: 4f54
INDEX 0
CONNECTED Yes
MAPPING n/a
TIMESTAMP 25998.20000
VIBRATION n/a
AXIS 0 0.00049
AXIS 1 0.00049
AXIS 2 -0.98828
AXIS 3 0.00049
AXIS 4 -1.00000
AXIS 5 -1.00000
AXIS 6 -1.00000
AXIS 7 -1.00000


Update Drivers - Intel(R) USB 3.1 eXtensible Host Controller - 1.10 (Microsoft)
The best drivers for your device are already installed
Windows has determined that the best driver for this device is already installed. There may be better drivers on Windows Update or on the device manufacturer's website.
USB xHCI Compliant Host Controller

1. Input Component (Scripting): A PlayerInput component on the drone uses Unity Events to call functions in my DroneController script (like OnThrottle, OnPitch) whenever an Action happens.
2. Flight Controller (Physics Logic): My DroneController.cs script gets these inputs. It applies the Deadzone and Expo processing and then, based on the currentMode (Angle, Acro, or Horizon), it calculates the physics forces.
3. Physics Engine (Simulation): In FixedUpdate(), the DroneController applies these forces (e.g., rb.AddForce()) to the drone's Rigidbody. Unity's physics engine then simulates the motion.
4. Displaying the Simulation (UI & Camera):
    - The CameraController.cs script runs in LateUpdate() to get the drone's new position and smoothly follow it.
    - The OSDController.cs script reads public variables from the DroneController (like velocity and altitude) and updates the UI Canvas text.

Hardware Integration

For this project, I used a physical RC (Radio-Controlled) transmitter, the Radiomaster Pocket. This is a compact, open-source radio controller used by FPV pilots to fly real drones.

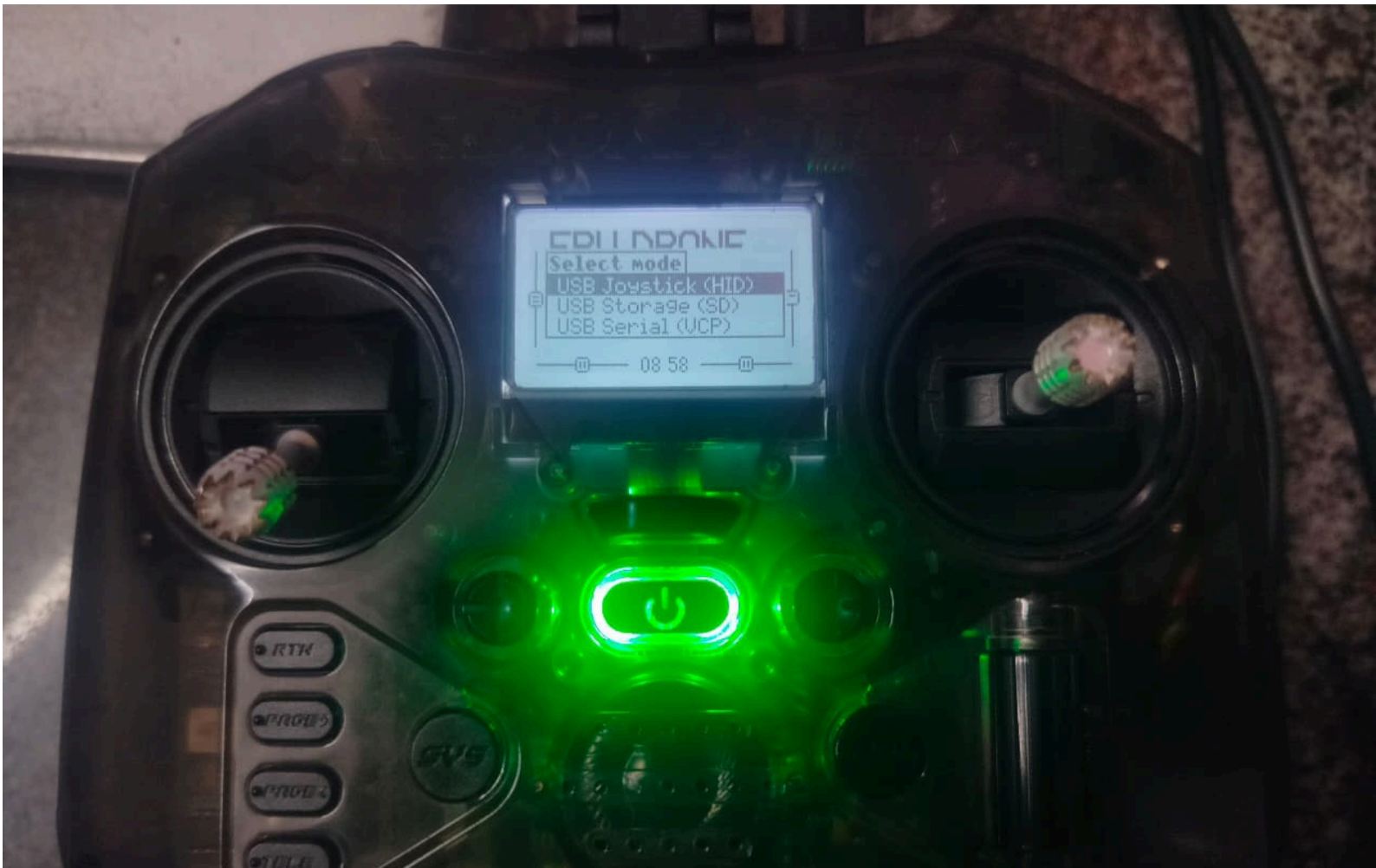


It's important to understand its two main functions:

1. Radio Transmission (ELRS): The "ELRS" stands for ExpressLRS. This is a modern, high-performance, open-source radio transmission protocol. It's known for its extremely long range (many kilometers) and low latency (fast response time), which is why it's popular in the FPV hobby. This protocol is what the radio uses to communicate wirelessly with a receiver on a real drone. This function is not used for our project, but it's what the "ELRS" in the name means.
2. USB Simulator Support (HID): This is the function I did use. The controller's internal computer (running EdgeTX, an open-source operating system) has a feature where it can stop acting as a radio. When plugged in via USB-C, it can tell the computer it's a generic USB Joystick (or HID - Human Interface Device).

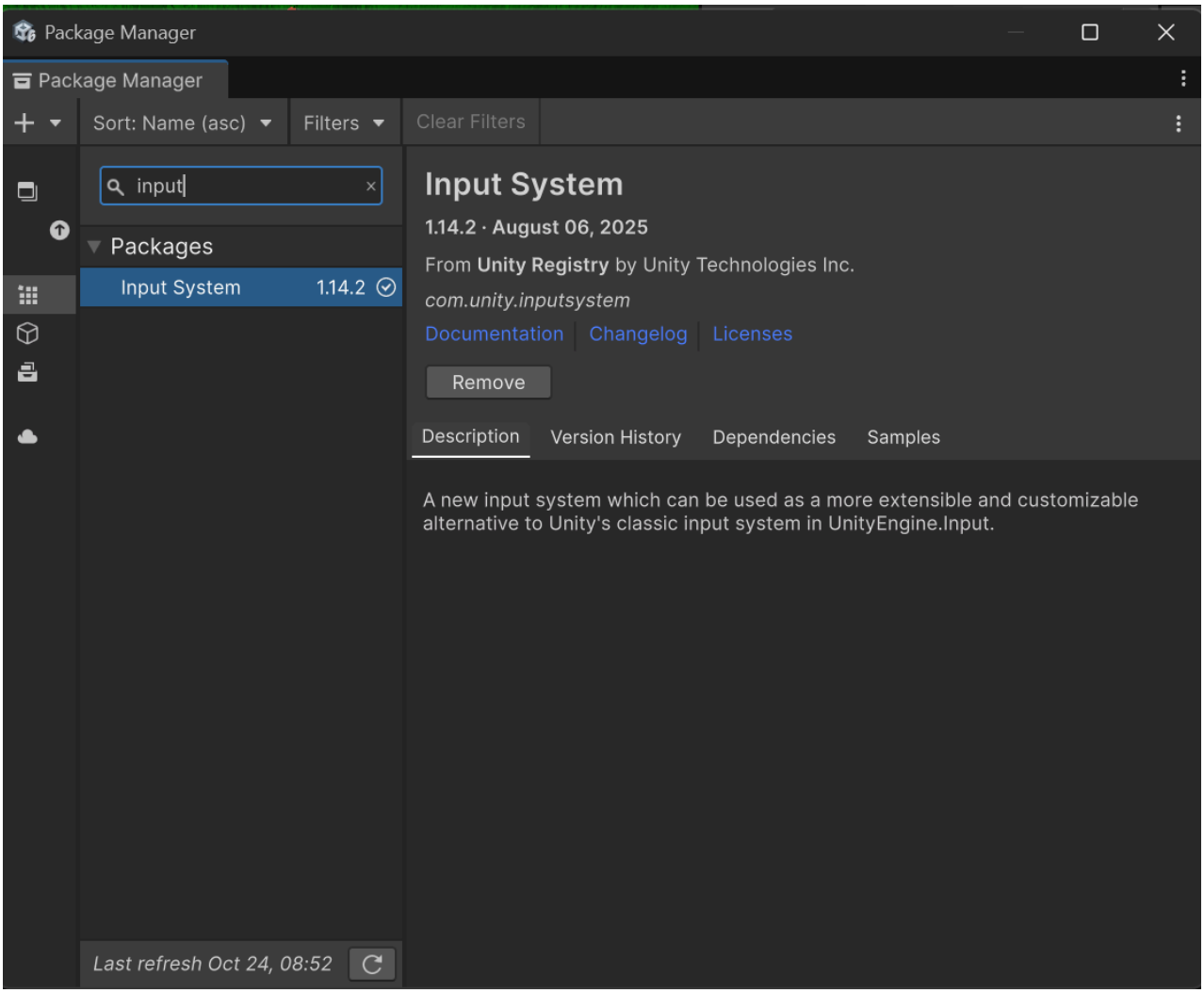
This "Joystick" mode is a standard feature in modern RC radios, designed specifically for training on simulators like this one. It provides a much more realistic and precise experience than a typical video game controller, as it has the same high-precision gimbals (sticks) and switches that a pilot would use in the real world.

HID Detection and Drivers



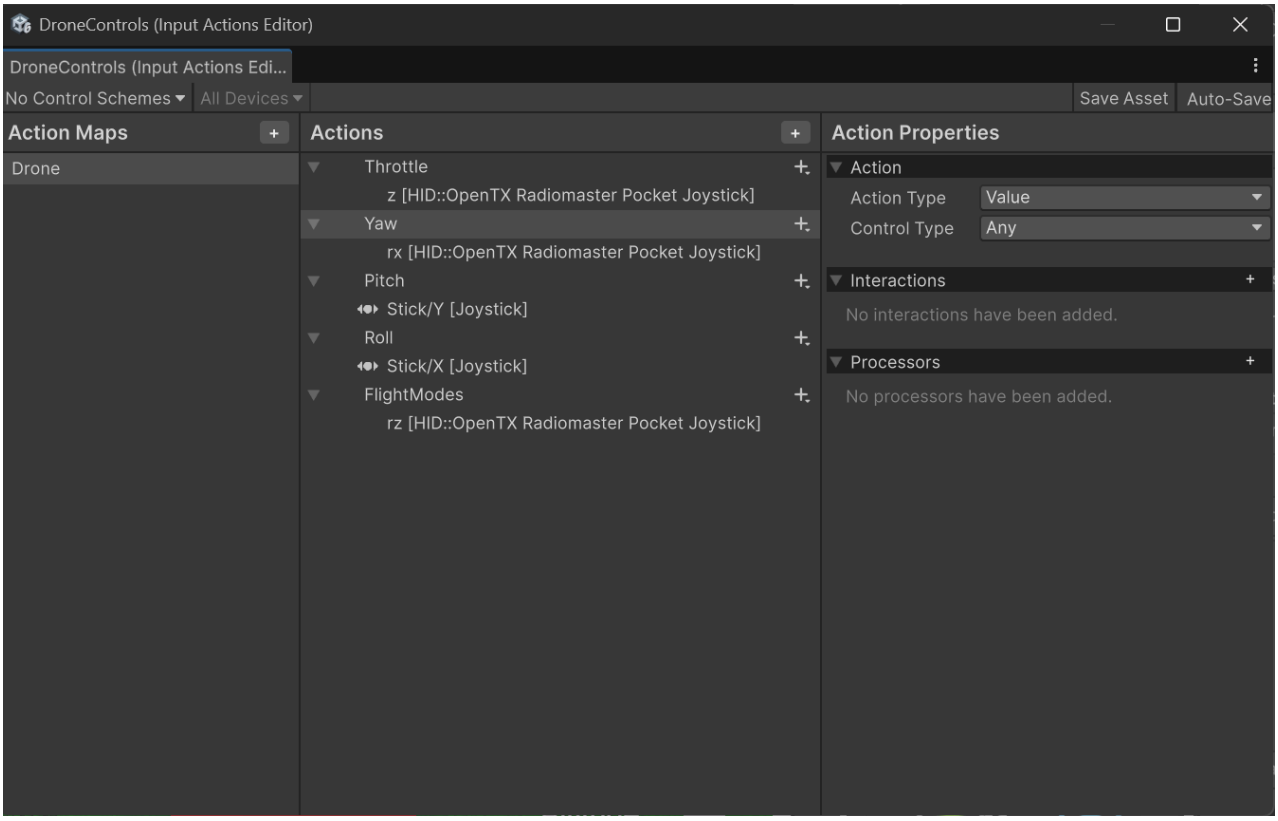
I connected the controller to my PC with a USB-C cable. In the controller's settings, I set the USB Mode to "Joystick". The controller's firmware then just tells the computer it's a standard game controller. Windows (or macOS) automatically uses its built-in HID-compliant game controller drivers, so no special driver installation was needed. Unity's Input System then just saw it as another available joystick.

Unity Input System Configuration



I used Unity's modern Input System package to handle the controls.

1. Input Action Asset: I created a DroneControls>input>actions asset.
2. Action Map: I made an Action Map called "Drone" for all flight controls.
3. Actions & Bindings: I defined five key Actions. The most important step was setting the Action Type to Value and the Control Type to Any for the sticks. This makes Unity read the full analogue float value (from -1.0 to 1.0) instead of treating it like a digital button.



Here's how I mapped the bindings:

Action	Action Type	Mapped Binding (Path)	Physical Control (Mode 2)
Throttle	Value	Z Axis	Left Stick (Up/Down)
Yaw	Value	Rx Axis	Left Stick (Left/Right)
Pitch	Value	Y Axis (Stick/Y)	Right Stick (Up/Down)
Roll	Value	X Axis (Stick/X)	Right Stick (Left/Right)
FlightModes	Value	Rz Axis	3-Position Switch (SA/SB)

List of Modules

- Unity Input System: Manages all hardware inputs from the RC controller.

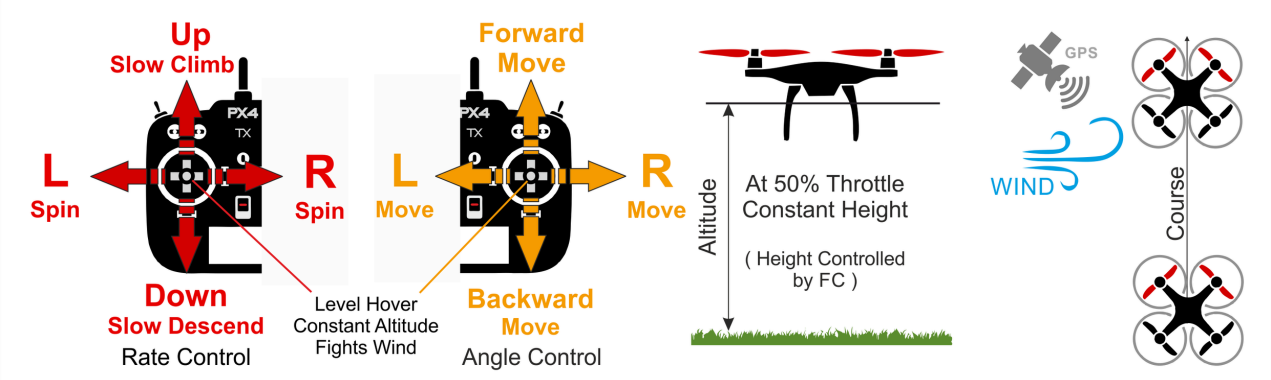
- Unity Physics Engine: Handles all simulation via the Rigidbody component.
- Unity UI (Canvas & TextMeshPro): Renders the On-Screen Display.
- DroneController.cs: The primary C# script acting as the drone's "Flight Controller."
- CameraController.cs: C# script for a smooth 3rd-person follow-camera.
- OSDController.cs: C# script that reads flight data and updates the UI.

Functionalities

DroneController.cs (The Flight Controller)

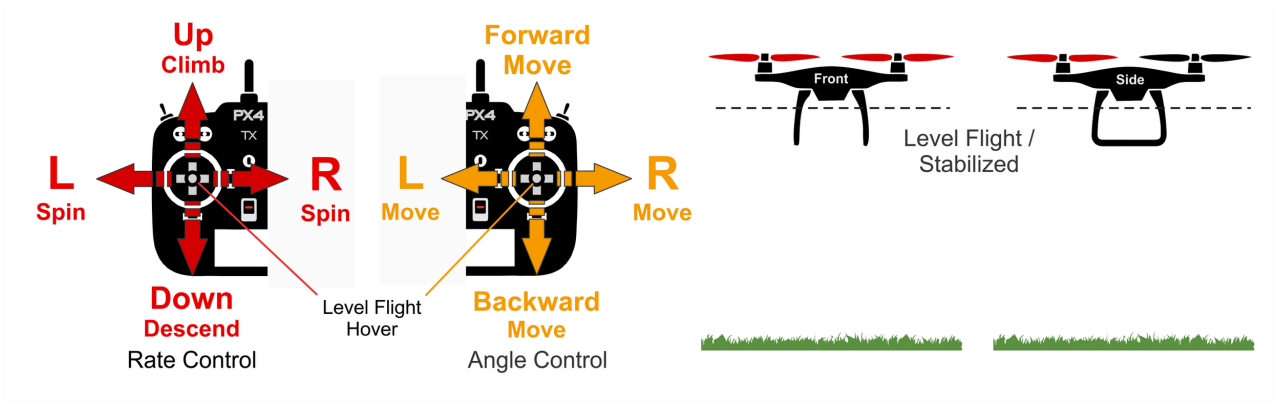
This is the most important script in the project. It receives inputs and applies physics forces to the drone's Rigidbody in FixedUpdate().

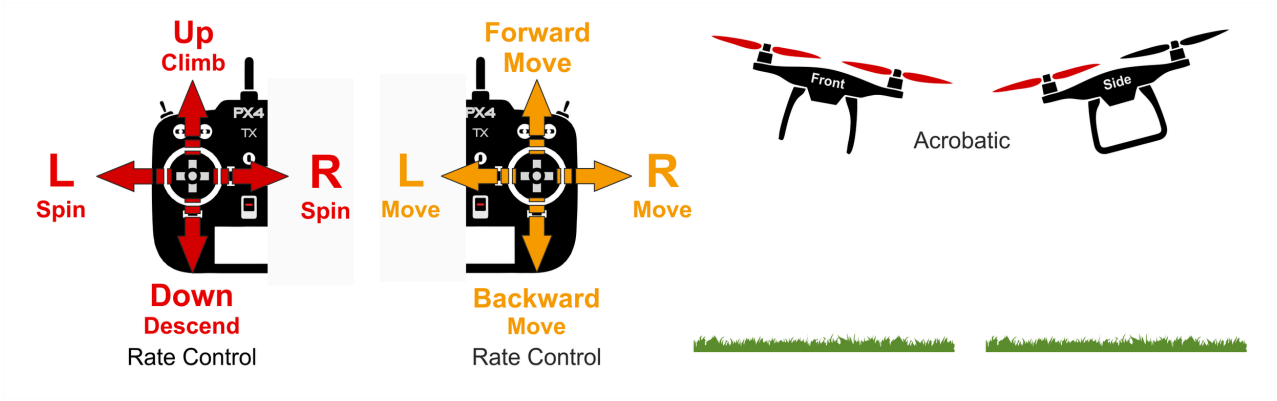
- **Physics-Based Flight**: Movement isn't faked with transform.Translate. I use Rigidbody.AddForce() for thrust and Rigidbody.AddRelativeTorque() for rotation. This makes the drone react to gravity and collisions.
- **Input Processing**: To get the right "feel," I had to process the raw stick inputs:
  - **InputDeadzone**: This fixes "stick drift." My radio's sticks don't return to a perfect 0.00. This function makes the code treat any very small input (e.g., < 0.07) as 0.
  - **AcroExpo / ThrottleExpo**: This applies a cubic curve to the inputs. It makes the sticks less sensitive for small movements (for fine-tuning a hover or a turn) and more sensitive at the edges (for fast flips and full throttle).
  - **YawDamping**: This was a key fix. It actively applies a "brake" torque to stop the drone from spinning (yawing) when the yaw stick is centered. This gives the drone a "locked-in" feel.



- **Flight Mode Logic**: This is the core logic. A 3-way switch on my radio changes the currentMode variable, and the HandleMovement function calls one of these three methods:

1. **Angle Mode**: This is the self-levelling mode.
    - How it works: It calls the HandleAngleMode() function. This function takes the pilot's stick input (pitchInput, rollInput) and treats it as a target angle (e.g., "I want to be tilted 20 degrees forward").
    - It then gets the drone's current angle (from transform.localEulerAngles).
    - It calculates the error (difference) between the target angle and the current angle.
    - Finally, it uses this error in our PD Controller formula: (error * stabilizationStrength) - (currentRotationSpeed * stabilizationDamping). This calculates a corrective torque to force the drone to the target angle and stops it from wobbling. When the stick is centered, the target angle becomes 0, so the drone self-levels.
2. **Acro Mode**: This is the manual/expert mode.
    - How it works: It calls HandleAcroMode(). In this function, the stick input is not treated as an angle.
    - Instead, the input (after expo is applied) is multiplied directly by pitchTorque or rollTorque to create a rotational force.
    - This means the stick controls the rate of rotation. Stick forward = rotate forward at X degrees/sec.
    - When the stick is centered, the input becomes 0 (after the deadzone). This means the torque becomes 0, the rotation stops, and the drone holds its current angle. This is the key feature that allows for flips and rolls.
3. **Horizon Mode**: This is a hybrid of the other two.
    - How it works: It calls HandleHorizonMode(). This function literally just calls the logic from both Acro and Angle mode at the same time.
    - It applies the Acro torque (so you can do full flips if you push the stick hard enough) and it applies a weaker version of the Angle mode's self-leveling torque.
    - The result is a drone that feels acrobatic but will self-level if you let go of the sticks, making it a good "in-between" mode.





CameraController.cs

This is a simple script to get a smooth 3rd-person camera. It runs in LateUpdate() (which is after all physics is done). It just uses Vector3.Lerp() to smoothly move the camera towards the drone's position, which prevents the jerky motion I'd get if I just parented the camera to the drone.

OSDController.cs

This script runs the "On-Screen Display." It just has public TextMeshPro variables that I dragged my UI text elements onto in the Inspector. In Update(), it gets the public properties from the DroneController (like currentMode and ThrottleInput) and formats them into a string for the UI text to display.

Code Snippets of Key Functionalities

Snippet 1: HandleMovement() (The Core Logic Loop)
This function from DroneController.cs runs every physics step. It applies thrust, handles the yaw damping, and then calls the correct function for the active flight mode.

```
private void HandleMovement()
{
   float processedThrottle = ApplyExpo(throttleInput, throttleExpo);
       rb.AddForce(transform.up * (processedThrottle * thrustForce), ForceMode.Acceleration);

   float processedYaw = ApplyExpo(yawInput, acroExpo) * yawTorque;
                                    float currentYawRate = transform.InverseTransformDirection(rb.angularVelocity).y;
   // Apply a damping force to stop unwanted yaw rotation
   float yawDampingTorque = -currentYawRate * yawDamping;

            rb.AddRelativeTorque(Vector3.up * (processedYaw + yawDampingTorque), ForceMode.Acceleration);

   switch (currentMode)
   {
      case FlightMode.Acro:
         HandleAcroMode();
         break;
      case FlightMode.Angle:
         HandleAngleMode();
         break;
      case FlightMode.Horizon:
         HandleHorizonMode();
         break;
   }
}
```

Snippet 2: HandleAngleMode() (PD Controller)
This is the implementation of the PD (Proportional-Derivative) self-leveling controller.

```
private void HandleAngleMode()
{
   float targetPitch = pitchInput * maxTiltAngle;
   float targetRoll = -rollInput * maxTiltAngle;

   Vector3 localEuler = transform.localEulerAngles;
   float currentPitch = ConvertAngle(localEuler.x);
   float currentRoll = ConvertAngle(localEuler.z);

   float pitchError = targetPitch - currentPitch;
   float rollError = targetRoll - currentRoll;

                                   Vector3 localAngularVel =
transform.InverseTransformDirection(rb.angularVelocity);

   // The core PD controller calculation
            float pitchPD = (pitchError * stabilizationStrength) -
(localAngularVel.x * stabilizationDamping);
    float rollPD = (rollError * stabilizationStrength) - (localAngularVel.z *
stabilizationDamping);

              rb.AddRelativeTorque(new Vector3(pitchPD, 0, rollPD),
ForceMode.Acceleration);
}
```

Snippet 3: HandleAcroMode() and ApplyExpo()
This shows the Acro (rate) mode logic. It also includes the ApplyExpo function which is key to the "feel" of the controls.

```
private void HandleAcroMode()
{
   float processedPitch = ApplyExpo(pitchInput, acroExpo);
   float processedRoll = ApplyExpo(rollInput, acroExpo);
    Vector3 acroTorque = new Vector3(processedPitch * pitchTorque, 0,
-processedRoll * rollTorque);
   rb.AddRelativeTorque(acroTorque, ForceMode.Acceleration);
}
```

```
private float ApplyExpo(float input, float expo)
{
    // This cubic formula makes the center of the stick less sensitive
    return (expo * (input * input * input)) + ((1f - expo) * input);
}
```

Snippet 4: Input System Handlers
These functions are set up to be called by the PlayerInput component. This is how the input values get into the script.

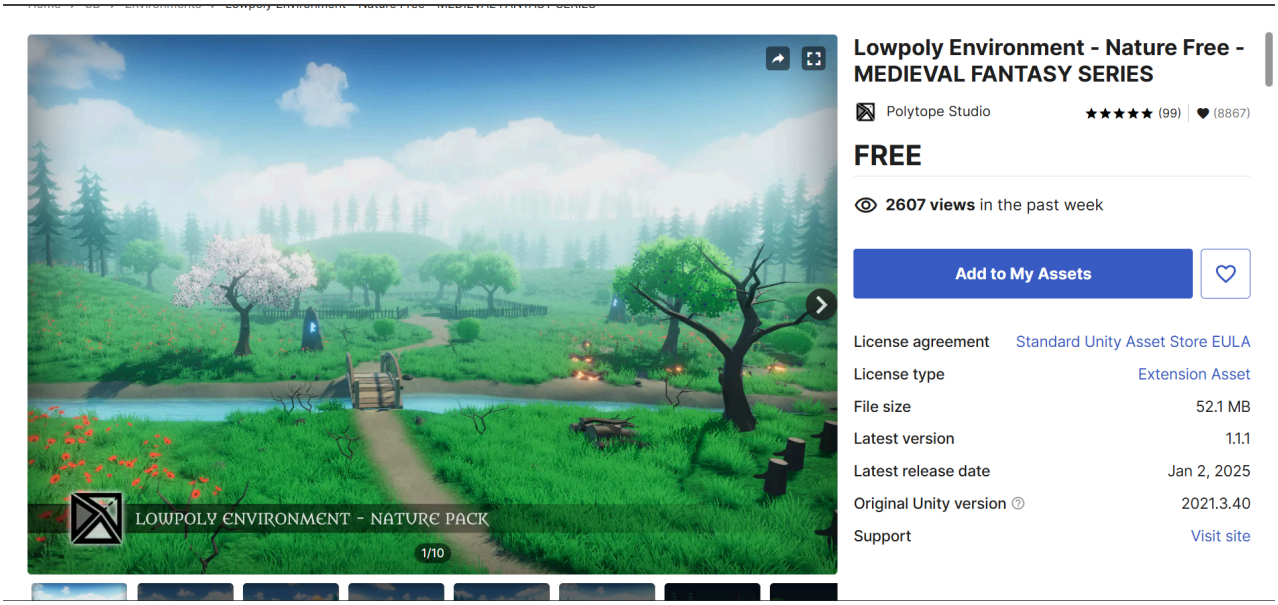
```
public void OnThrottle(InputValue value)
{
    float rawInput = value.Get<float>();
    // Remap stick from [-1, 1] range to [0, 1] range
    throttleInput = (rawInput + 1f) / 2f;
}
public void OnYaw(InputValue value)
{
    // Apply deadzone immediately to prevent drift
    yawInput = ApplyDeadzone(value.Get<float>());
}
public void OnPitch(InputValue value)
{
    pitchInput = ApplyDeadzone(value.Get<float>());
}
public void OnFlightModes(InputValue value)
{
    float rzValue = value.Get<float>(); // 3-pos switch gives -1, 0, or 1
    if (rzValue < -0.5f)
    {
        currentMode = FlightMode.Acro;
    }
    else if (rzValue > 0.5f)
    {
        currentMode = FlightMode.Angle;
    }
    else
    {
        currentMode = FlightMode.Horizon;
    }
}
```

Snippet 5: OSDController.cs Update Loop

This shows how the OSD script reads data from the drone controller and updates the screen text every frame.

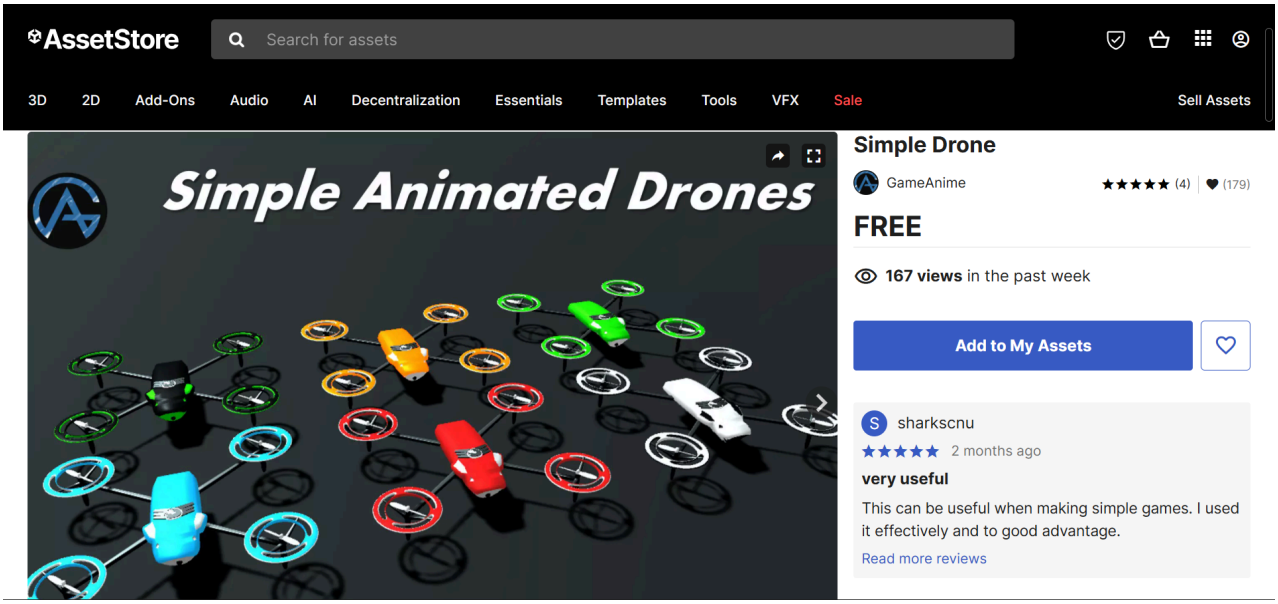
```
void Start()
{
   // Get the starting altitude so we can measure relative height
   if (drone != null)
   {
      initialAltitude = drone.transform.position.y;
   }
}

void Update()
{
   if (drone == null) return; // Don't do anything if drone isn't linked

   // Update all the text fields
   modeText.text = $"MODE: {drone.CurrentMode.ToString().ToUpper()}";
   throttleText.text = $"THR: {drone.ThrottleInput:P0}";

   float altitude = drone.transform.position.y - initialAltitude;
   altitudeText.text = $"ALT: {altitude:F1} m";

   float velocity = drone.Rb.velocity.magnitude;
   velocityText.text = $"VEL: {velocity:F1} m/s";
}
```

Assets Used



https://assetstore.unity.com/packages/3d/environments/lowpoly-environment-nature-free-medieval-fantasy-series-187052

https://youtu.be/WbZpj8WcjN0?si=Oum9ZBz3oIvZDdbL



https://assetstore.unity.com/packages/3d/vehicles/air/simple-drone-190684

Demo Video

https://youtu.be/6hnOIN6PqiQ

https://github.com/Venkatesan-M/UnityFPVDroneSimulator

Thank you !

Venkatesan M

venkatesan.m2022@vitstudent.ac.in