Stage 5

▼ **MASTER**

▼ **Environment**

1. Art
2. Prefabs
3. Pipelines
4. Shaders
5. TMP
6. Aircrafts

- Other things will also be added here
- Soon the whole environment asset package will be added here
- Aircrafts have their own materials shaders art etc.

▼ **Manager**

1. Test Manager ⇒ Controls the game mechanics for the test cases
2. Console Manager ⇒ Conrols the input and output of displayed console for test cases

- Both of these scripts can be extended later on as the testing requirements will increase

▼ **Units**

▼ Aircraft

1. Aircraft Base ⚾ (Scriptable Object)
2. Aircrafts 🏓
3. Aircraft Controller 🕹️
4. Aircraft Collider 🕹️
5. Aircraft Camera 🕹️

▼ Situation

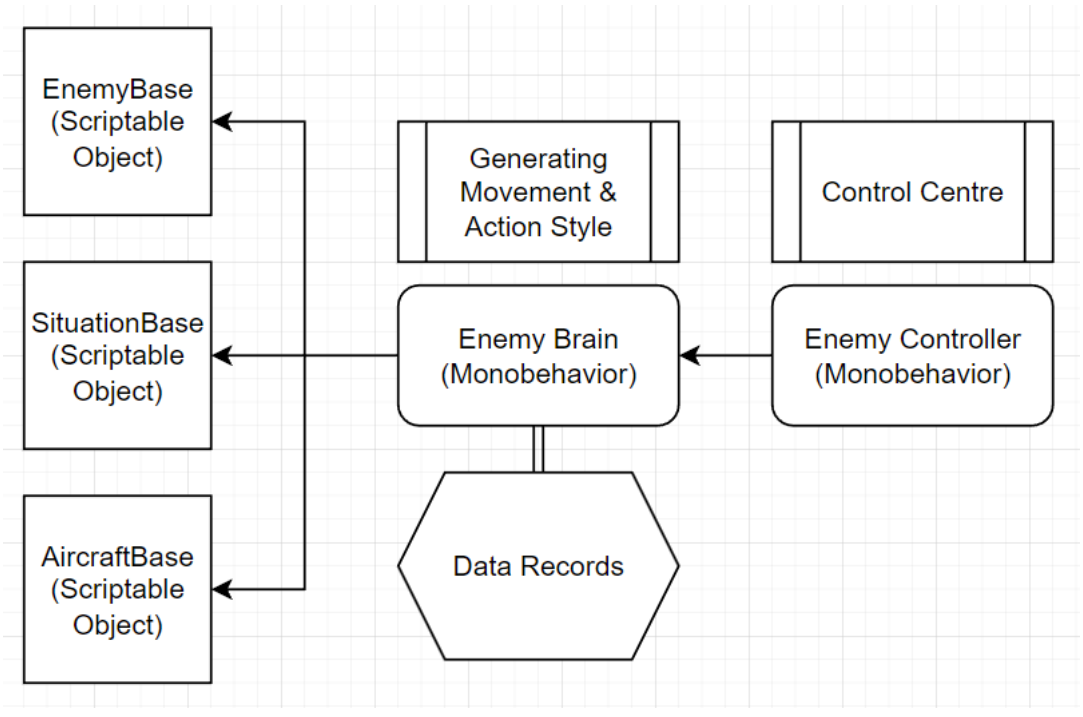
1. Situation Base ⚾ (Scriptable Object)
2. Situations 🏓

▼ Enemy

1. Enemy Base ⚾ (Scriptable Object)
2. Enemies 🏓
3. Enemy Controller derived from Aircraft Controller 🕹️
4. Prefabs 🎁
5. Enemy Brain 🧠

▼ Player

1. Player Controller derived from Aircraft Controller 🕹️
2. Prefabs 🎁



▼ **PRESENT 5A**

"The project is a 3D action strategy simulation game that focuses on hyper-realistic military aerial combat. In this game, the player will engage in aerial combat against enemy jets controlled by advanced AI.

The enemy AI consists of two scripts, EnemyBrain and EnemyController. The EnemyBrain processes waypoints and analyzes scriptable objects (EnemyBase, SituationBase, and AircraftBase) to determine the movement style, objectives, and constraints of the enemy. Currently, the SituationBase consists of a roam range which is read by the EnemyBrain, and the EnemyBrain randomly generates waypoints within this range. The EnemyController is responsible for moving the aircraft to these waypoints, and if the enemy is about to collide with an obstacle, the EnemyController requests a new waypoint from the EnemyBrain. The EnemyController is also responsible for deciding whether or not to use the turbo boost, based on the enemy's offensive property in the EnemyBase.

This was just a demo of how the various elements will integrate with each other to create a dynamic and sophisticated enemy AI system."

▼ **FUTURE**

1. Movement and Scenario Building
2. Pathfinding and Escape
3. Dogfight and Maneuvers
4. Data Saving & Post-Run Learning

▼ **BASE CLASSES**

Enemy

Aa Name	Category	Type
Initiative	Strategy	MasterAttribute
Positional Awareness	Strategy	MasterAttribute
Calculative Agility	Strategy	MasterAttribute
Tactical Awareness	Strategy	MasterAttribute
Experience	Wisdom	MasterAttribute
Resilience	Wisdom	MasterAttribute
Courage	Wisdom	MasterAttribute
Stabliity	Wisdom	MasterAttribute
Risk Tolerance	Wisdom	MasterAttribute
Aggression	Personality	MasterAttribute
Honor	Personality	MasterAttribute
Imagination	Personality	MasterAttribute
Obedience	Personality	MasterAttribute
Communication	Personality	MasterAttribute
Gear Mastery	Skill	MasterAttribute
Maneuver Mastery	Skill	MasterAttribute
Control Mastery	Skill	MasterAttribute
Memory	Skill	MasterAttribute
Rank Order	Skill	MasterAttribute
Maneuver Preference	Behavior	List<Maneuver>
Altitude Preference	Behavior	MasterAttribute
Speed Preference	Behavior	MasterAttribute

Situation

Aa Name	Type
Roam Range	Range
Objective Given	ObjectiveType

Aircraft

Aa Name	Type
Max Safe Altitude	Float
Search Distance	Float

▼ About Enemy Attributes

▼ Strategy

1. Tactical Awareness
    a. Meaning ⇒ Overall accuracy of enemy knowing the available strategies and acting upon the most effective one.
    b. Description ⇒ A pilot with high tactical awareness might be able to quickly understand the strategic situation and make decisions about when to engage the player, when to evade, and when to regroup with other aircraft. Conversely, a pilot with low tactical awareness might be more prone to making poor decisions or being caught off guard by the player.
    c. Implementation ⇒ This will be used when AI will be processing strategies and calculating decisions.
2. Positional Awareness
    a. Meaning ⇒ Overall accuracy of enemy knowing the positions of aircrafts in the scenario and their relationships.
    b. Description ⇒ A pilot with high positional awareness might be able to effectively maneuver to get into an advantageous position to engage the player, while a pilot with low positional awareness might be more prone to collisions or being caught off guard by the player's movements.
    c. Implementation ⇒ This will be used as an offset value when AI will calculate the aircrafts positioning.
3. Stratagem Agility
    a. Meaning ⇒ Speed at which enemy can make tactical decisions and adjust their strategy in real-time.
    b. Description ⇒ A pilot with high stratagem agility might be able to quickly and effectively adjust their tactics in response to changing circumstances, while a pilot with low stratagem agility might be slow to react or struggle to adapt to new situations.
    c. Implementation ⇒ This will be used as an offset value when AI will calculate and implement new tactics during gameplay.
4. Calculative Agility
    a. Meaning ⇒ Speed at which enemy can perform mathematical and positional calculations during gameplay.
    b. Description ⇒ A pilot with high calculative agility might be able to quickly and accurately determine the best course of action based on their positional awareness and other situational factors, while a pilot with low calculative agility might struggle to make effective calculations or be slow to react.
    c. Implementation ⇒ This will be used as an offset value when AI will calculate the aircraft's position and make decisions about movement during gameplay.

5. Initiative
    a. Meaning: The ability to take charge and make quick, decisive decisions in combat situations.
    b. Description: A pilot with high initiative might be more likely to take the lead and take charge of the situation, while a pilot with low initiative might be more likely to wait for someone else to take the lead or be more passive in their approach.
    c. Implementation: This property will affect the pilot's behavior when facing combat situations, such as taking the lead in attacking the player or making the first move in a dogfight.

▼ Wisdom

1. Risk Tolerance
    a. Meaning ⇒ The extent to which the enemy is willing to take risks to achieve their goals.
    b. Description ⇒ An enemy with high risk tolerance might be more likely to engage in aggressive or dangerous maneuvers in order to achieve their objectives, while an enemy with low risk tolerance might be more cautious and defensive.
    c. Implementation ⇒ This will be used to determine the likelihood of the enemy taking certain actions, such as engaging the player directly or retreating to a safer position.
2. Stability
    a. Meaning ⇒ The ability of the enemy to maintain their emotional and mental well-being in stressful and dangerous situations.
    b. Description ⇒ A pilot with high stability is more likely to remain calm and focused, even under intense pressure. Conversely, a pilot with low stability may be more prone to making poor decisions or becoming distracted. This can be influenced by factors such as their morale and past experiences.
    c. Implementation ⇒ This property will affect how the pilot responds to different situations in the game, such as being under fire or encountering unexpected obstacles.
3. Courage
    a. Meaning ⇒ The ability of the pilot to face danger, fear, or adversity with bravery and determination.
    b. Description ⇒ A pilot with high courage may be more willing to engage in dangerous or risky situations, while a pilot with low courage may be more prone to evasive actions or retreat.
    c. Implementation ⇒ This property will affect the pilot's behavior when faced with challenging or dangerous situations in the game, such as engaging in aerial combat or navigating through hazardous terrain.
4. Resilience
    a. Meaning ⇒ The ability to recover from difficulties, stress, or challenges.
    b. Description ⇒ A pilot with high resilience might be more likely to maintain their composure and keep fighting despite setbacks, while a pilot with low resilience might become discouraged or give up more easily.
    c. Implementation ⇒ This will be used to influence the pilot's decision-making and ability to keep fighting despite challenges.
5. Experience
    a. Meaning ⇒ The amount of exposure to combat and survival situations.
    b. Description ⇒ A pilot with more experience might be more skilled at adapting to changing circumstances and anticipating enemy actions, while a pilot with less experience might be more prone to making mistakes or being caught off guard.
    c. Implementation ⇒ This will be used as an offset value when the AI will be making decisions in response to the combat situation, such as choosing when to engage the player, when to evade, and when to regroup with other aircraft.

▼ Personality

1. Communication
    a. Meaning: The ability to effectively convey ideas and information to others.
    b. Description: A pilot with strong communication skills might be better at relaying information to their teammates or giving clear commands in combat situations, while a pilot with poor communication skills might struggle to be understood or effectively give orders.
    c. Implementation: This property will affect the pilot's ability to work effectively in a team, such as their ability to give clear and concise orders to their teammates or successfully coordinate their actions in combat.
2. Obedience
    a. Meaning: The willingness to follow orders or rules.
    b. Description: A pilot with high obedience might be more likely to follow rules and regulations, while a pilot with low obedience might be more likely to act independently or bend the rules.
    c. Implementation: This property will affect the pilot's behavior within the context of the game, such as their willingness to follow orders from their commanding officer or their likelihood to follow the rules of engagement.
3. Imagination
    a. Meaning: The ability to think creatively and come up with new and innovative ideas.
    b. Description: A pilot with a strong imagination might be better at finding new and unique solutions to problems, while a pilot with limited imagination might struggle to think outside the box.
    c. Implementation: This property will affect the pilot's ability to come up with unique strategies in combat, such as finding new and innovative ways to outmaneuver their opponents or finding creative solutions to challenging situations.
4. Honor
    a. Meaning: The adherence to a set of moral and ethical principles.
    b. Description: A pilot with strong honor might be more likely to follow a strict code of ethics, while a pilot with low honor might be more likely to act in their own self-interest or bend their principles for personal gain.
    c. Implementation: This property will affect the pilot's behavior within the game, such as their adherence to the rules of engagement or their willingness to engage in unethical actions like firing on defenseless targets.
5. Aggression
    a. Meaning ⇒ The degree to which the enemy is inclined to be aggressive and assertive in their actions.
    b. Description ⇒ An enemy with high aggression might be more likely to take the offensive and actively engage the player, while an enemy with low aggression might be more passive and defensive.
    c. Implementation ⇒ This will be used to determine the likelihood of the enemy taking certain actions, such as initiating an attack on the player or pursuing them relentlessly.

▼ Skill

1. Rank Order
    a. Meaning: The ranking system that assigns positions to everyone in their hierarchy.

b. Description: A pilot with a higher rank order might be more of a priority target for the player, while a lower ranked enemy might pose less of a threat. Rank order also determines the communication.
c. Implementation: This property will determine the priority of the enemies and also how the communication should be held.

2. Memory
   a. Meaning: The ability to recall information, such as enemy positions and strategies, from past encounters or on-going scenario.
   b. Description: A pilot with good memory might be better at recalling information and using it to their advantage in combat, while a pilot with poor memory might struggle to remember important details and be more prone to making mistakes.
   c. Implementation: This property will affect the pilot's behavior in combat, influencing the choices they make and their ability to recall information from past encounters.
3. Control Mastery
   a. Meaning: The ability to effectively control and maneuver the aircraft in various combat situations.
   b. Description: A pilot with high control mastery will be able to make precise and accurate movements with their aircraft, making them more effective in dodging enemy fire and outmaneuvering their opponents.
   c. Implementation: This property will affect the pilot's ability to maneuver their aircraft in combat, such as making tight turns or flying at high speeds.
4. Maneuver Mastery
   a. Meaning: The ability to perform various combat maneuvers with precision and speed.
   b. Description: A pilot with high maneuver mastery will be able to execute complex and dangerous combat maneuvers with ease, giving them an advantage in dogfights and other aerial combat situations.
   c. Implementation: This property will affect the pilot's ability to perform different types of combat maneuvers, such as rolling, diving, or banking.
5. Gear Mastery
   a. Meaning: The ability to effectively use the aircraft's various systems and equipment, including weapons, fuel, and other systems.
   b. Description: A pilot with high gear mastery will be able to make the most of their aircraft's systems and equipment, using them to their fullest potential in combat situations.
   c. Implementation: This property will affect the pilot's ability to use the aircraft's systems and equipment, such as choosing the right weapons for the situation or efficiently managing fuel to stay in the air for longer.

▼ Behavior

1. Speed Preference
   a. Meaning: The preferred speed of the pilot when in combat.
   b. Description: Some pilots might prefer faster speeds, while others might prefer slower speeds. This property will affect the pilot's behavior in combat situations, such as how they approach the player and how they react to different situations.
   c. Implementation: This property will influence the pilot's behavior in combat situations, such as their preferred speed when attacking the player or when attempting to evade the player.
2. Altitude Preference
   a. Meaning: The preferred altitude of the pilot when in combat.
   b. Description: Some pilots might prefer flying at higher altitudes, while others might prefer flying at lower altitudes. This property will affect the pilot's behavior in combat situations, such as how they approach the player and how they react to different situations.
   c. Implementation: This property will influence the pilot's behavior in combat situations, such as their preferred altitude when attacking the player or when attempting to evade the player.

▼ About Situation Attributes

1. Roam Range - Cube/Cuboid/Sphere
2. Objective Type - Fight/Follow/Escape/Patrol/Scout

▼ About Aircraft Attributes

1. Max Safe Altitude
2. Search Distance

Note: There are a lot more attributes but those are the casual ones for controller which will be changed in future.

▼ **GENERATE WAYPOINT PATTERN (ROAM)**

To create dynamic movement and roaming behavior for the enemy AI, we will generate a list of vector3 waypoints with v1 length based on the properties of the enemy's base classes. Rather than having the enemy move randomly, the enemy AI will generate waypoints close to the next waypoint based on a fixed and tested distance variable, which will be represented by v2.

The height of the new waypoint, represented by v3, will be dependent on attributes such as altitude preferences or aircraft limitations. Enemy AI attributes will mostly be contained in a custom class called MasterAttribute, which will have three properties: statBase, statCons, statMarg.

v4 and v5 will be used for x-directional change and z-directoinal change which will make interesting patterns. The movement generated by the enemy AI will create natural and smooth patterns while also remembering the last waypoint result and altering it slightly. The direction will also be remembered to avoid abrupt changes in movement.

These changes will result in unique patterns and distinct roaming behavior for each randomly generated enemy AI. As development progresses, movement restrictions for aircraft limitations and obstacle avoidance can be incorporated.

Paths and patterns are created successfully.

💡 Testing scenario is that the planes are spawned they roam in a similar height around their range and then stop at their processed preferred height

▼ **SCRIPTS LOG CURRENT**

▼ Situation Base

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[CreateAssetMenu(fileName = "Situation", menuName = "ScriptableObjects/Situation", order = 2)]
public class SituationBase : ScriptableObject
{
    #region Variables to be assigned
    [SerializeField] private TheRange roamRange;
    [SerializeField] private ObjectiveType objectiveGiven;
    #endregion

    #region Variables to be accessed
    public TheRange RoamRange { get { return roamRange; } }
    public ObjectiveType ObjectiveGiven { get { return objectiveGiven; } }
```

```
    #endregion

    public void MakeAircraftNucleus(TheRange whichRange, Transform aircraftPosition)
    {
        whichRange.ShapeNucleus = aircraftPosition.position;
    }
}
[System.Serializable]
public class TheRange
{
    #region Variables to be assigned
    [SerializeField] private RangeType rangeType;
    [SerializeField] private Vector3 shapeNucleus;
    [SerializeField] private bool aircraftNucleus = false;
    [SerializeField] private float st1;
    [SerializeField] private float st2;
    [SerializeField] private float st3;
    #endregion

    #region Variables to be accessed
    public RangeType RangeType { get { return rangeType; } }
    public Vector3 ShapeNucleus { get; set; }
    public bool AircraftNucleus { get { return aircraftNucleus; } }
    public float ST1 { get { return st1; } }
    public float ST2 { get { return st2; } }
    public float ST3 { get { return st3; } }
    #endregion

}

public enum RangeType
{
    Cube, //ST1 = Radius
    Cuboid, //ST1 = Length, ST2 = Width, ST3 = Height
    Sphere //STl = Radius
}

public enum ObjectiveType
{
    Escape,
    Follow,
    Fight
}
```

▼ Enemy Base

```
using System.Collections;
using System.Collections.Generic;
using System.Reflection;
using UnityEngine;

[CreateAssetMenu(fileName = "EnemyName", menuName = "ScriptableObjects/Enemy", order = 1)]
public class EnemyBase : ScriptableObject
{
    #region Variables to be assigned
    [SerializeField] private MasterAttribute strategyTacticalAwareness;
    [SerializeField] private MasterAttribute strategyPositionalAwareness;
    [SerializeField] private MasterAttribute strategyStrategemAgility;
    [SerializeField] private MasterAttribute strategyCalculativeAgility;
    [SerializeField] private MasterAttribute strategyInitiative;

    [SerializeField] private MasterAttribute wisdomRiskTolerance;
    [SerializeField] private MasterAttribute wisdomStability;
    [SerializeField] private MasterAttribute wisdomCourage;
    [SerializeField] private MasterAttribute wisdomResilience;
    [SerializeField] private MasterAttribute wisdomExperience;

    [SerializeField] private MasterAttribute personalityCommunication;
    [SerializeField] private MasterAttribute personalityObedience;
    [SerializeField] private MasterAttribute personalityImagination;
    [SerializeField] private MasterAttribute personalityHonor;
    [SerializeField] private MasterAttribute personalityAgression;

    [SerializeField] private MasterAttribute skillRankOrder;
    [SerializeField] private MasterAttribute skillMemory;
    [SerializeField] private MasterAttribute skillControlMastery;
    [SerializeField] private MasterAttribute skillManeuverMastery;
    [SerializeField] private MasterAttribute skillGearMastery;

    [SerializeField] private List<Maneuver> behaviorManeuverPreference = new List<Maneuver>();
    [SerializeField] private MasterAttribute behaviorSpeedPreference;
    [SerializeField] private MasterAttribute behaviorAltitudePreference;
    #endregion

    #region Variables to be accessed
    public MasterAttribute StrategyTacticalAwareness { get { return strategyTacticalAwareness; } set { strategyTacticalAwareness = value; } }
    public MasterAttribute StrategyPositionalAwareness { get { return strategyPositionalAwareness; } set { strategyPositionalAwareness = value; } }
    public MasterAttribute StrategyStrategemAgility { get { return strategyStrategemAgility; } set { strategyStrategemAgility = value; } }
    public MasterAttribute StrategyCalculativeAgility { get { return strategyCalculativeAgility; } set { strategyCalculativeAgility = value; } }
    public MasterAttribute StrategyInitiative { get { return strategyInitiative; } set { strategyInitiative = value; } }

    public MasterAttribute WisdomRiskTolerance { get { return wisdomRiskTolerance; } set { wisdomRiskTolerance = value; } }
    public MasterAttribute WisdomStability { get { return wisdomStability; } set { wisdomStability = value; } }
    public MasterAttribute WisdomCourage { get { return wisdomCourage; } set { wisdomCourage = value; } }
    public MasterAttribute WisdomResilience { get { return wisdomResilience; } set { wisdomResilience = value; } }
    public MasterAttribute WisdomExperience { get { return wisdomExperience; } set { wisdomExperience = value; } }

    public MasterAttribute PersonalityCommunication { get { return personalityCommunication; } set { personalityCommunication = value; } }
    public MasterAttribute PersonalityObedience { get { return personalityObedience; } set { personalityObedience = value; } }
    public MasterAttribute PersonalityImagination { get { return personalityImagination; } set { personalityImagination = value; } }
    public MasterAttribute PersonalityHonor { get { return personalityHonor; } set { personalityHonor = value; } }
    public MasterAttribute PersonalityAgression { get { return personalityAgression; } set { personalityAgression = value; } }

    public MasterAttribute SkillRankOrder { get { return skillRankOrder; } set { skillRankOrder = value; } }
    public MasterAttribute SkillMemory { get { return skillMemory; } set { skillMemory = value; } }
    public MasterAttribute SkillControlMastery { get { return skillControlMastery; } set { skillControlMastery = value; } }
    public MasterAttribute SkillManeuverMastery { get { return skillManeuverMastery; } set { skillManeuverMastery = value; } }
    public MasterAttribute SkillGearMastery { get { return skillGearMastery; } set { skillGearMastery = value; } }

    public List<Maneuver> BehaviorManeuverPreference { get { return behaviorManeuverPreference; } set { behaviorManeuverPreference = value; } }
    public MasterAttribute BehaviorSpeedPreference { get { return behaviorSpeedPreference; } set { behaviorSpeedPreference = value; } }
    public MasterAttribute BehaviorAltitudePreference { get { return behaviorAltitudePreference; } set { behaviorAltitudePreference = value; } }
    #endregion

    #region Randomize
    public static EnemyBase CreateRandomEnemyBase()
    {
        EnemyBase enemyBase = CreateInstance<EnemyBase>();

        PropertyInfo[] properties = typeof(EnemyBase).GetProperties();
        foreach (PropertyInfo property in properties)
        {
            if (property.CanWrite)
            {
                if (property.PropertyType == typeof(MasterAttribute))
                {
                    MasterAttribute ma = new MasterAttribute();
```

```
                    ma.StatBase = Random.Range(0f, 100f);
                    ma.StatCons = Random.Range(0f, 1f);
                    ma.StatMarg = Random.Range(0f, 1f);
                    if(property.Name == "BehaviorAltitudePreference") { Debug.Log(ma.StatBase); }
                    property.SetValue(enemyBase, ma);
                    if (property.Name == "BehaviorAltitudePreference") { Debug.Log(enemyBase.behaviorAltitudePreference.StatBase); }

                }
                else if (property.PropertyType == typeof(bool))
                {
                    property.SetValue(enemyBase, Random.value > 0.5f);
                }
                // Add more property types if necessary
            }
        }
        return enemyBase;
    }
    #endregion
}

#region Base
public enum Maneuver
{
    None,
}

[System.Serializable]
public class MasterAttribute
{
    [Range(0, 100)]
    [SerializeField] float statBase; //Base Value
    [Range(0, 1)]
    [SerializeField] float statCons; //Consistency
    [Range(0, 1)]
    [SerializeField] float statMarg; //Margin

    public float StatBase { get { return statBase; } set { statBase = value; } }
    public float StatCons { get { return statCons; } set { statCons = value; } }
    public float StatMarg { get { return statMarg; } set { statMarg = value; } }

    public float StatGet
    {
        get
        {
            //Use all three and get the stat random
            return statBase;
        }
    }
}
#endregion
```

▼ Player Base

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PlayerController : AircraftController
{
    #region Base
    private void Start()
    {
        //Setup speeds
        maxSpeed = aircraftBase.DefaultSpeed;
        currentSpeed = aircraftBase.DefaultSpeed;

        //Get and set rigidbody
        rb = GetComponent<Rigidbody>();
        rb.isKinematic = true;
        rb.useGravity = false;
        rb.collisionDetectionMode = CollisionDetectionMode.ContinuousSpeculative;

        SetupColliders(crashCollidersRoot);
    }

    private void Update()
    {
        AudioSystem();

        //Airplane move only if not dead
        if (!planeIsDead)
        {
            Movement();
            Dyanmics();

            //Rotate propellers if any
            if (propellers.Length > 0)
            {
                RotatePropellers(propellers);
            }
        }
        else
        {
            ChangeWingTrailEffectThickness(0f);
        }

        //Control lights if any
        if (turbineLights.Length > 0)
        {
            ControlEngineLights(turbineLights, currentEngineLightIntensity);
        }

        //Crash
        if (!planeIsDead && HitSometing())
        {
            Crash();
        }
    }
    #endregion

    #region Movement
    private void Movement()
    {
        //Move forward
        transform.Translate(Vector3.forward * currentSpeed * Time.deltaTime);

        //Rotate airplane by inputs
        transform.Rotate(Vector3.forward * -Input.GetAxis("Horizontal") * currentRollSpeed * Time.deltaTime);
        transform.Rotate(Vector3.right * Input.GetAxis("Vertical") * currentPitchSpeed * Time.deltaTime);

        //Rotate yaw
        if (Input.GetKey(KeyCode.E))
        {
            transform.Rotate(Vector3.up * currentYawSpeed * Time.deltaTime);
        }
        else if (Input.GetKey(KeyCode.Q))
        {
```

```
            transform.Rotate(-Vector3.up * currentYawSpeed * Time.deltaTime);
        }
    }
    #endregion

    #region Dyanamics & Turbo
    private void Dyanmics()
    {
        //Accelerate and deacclerate
        if (currentSpeed < maxSpeed)
        {
            currentSpeed += aircraftBase.Accelerating * Time.deltaTime;
        }
        else
        {
            currentSpeed -= aircraftBase.Deaccelerating * Time.deltaTime;
        }

        //Turbo
        if (Input.GetKey(KeyCode.LeftShift))
        {
            //Set speed to turbo speed and rotation to turbo values
            maxSpeed = aircraftBase.TurboSpeed;

            currentYawSpeed = aircraftBase.YawSpeed * aircraftBase.YawTurboMultiplier;
            currentPitchSpeed = aircraftBase.PitchSpeed * aircraftBase.PitchTurboMultiplier;
            currentRollSpeed = aircraftBase.RollSpeed * aircraftBase.RollTurboMultiplier;

            //Engine lights
            currentEngineLightIntensity = aircraftBase.TurbineLightTurbo;

            //Effects
            ChangeWingTrailEffectThickness(aircraftBase.TrailThickness);

            //Audio
            currentEngineSoundPitch = aircraftBase.TurboSoundPitch;
        }
        else
        {
            //Speed and rotation normal
            maxSpeed = aircraftBase.DefaultSpeed;

            currentYawSpeed = aircraftBase.YawSpeed;
            currentPitchSpeed = aircraftBase.PitchSpeed;
            currentRollSpeed = aircraftBase.RollSpeed;

            //Engine lights
            currentEngineLightIntensity = aircraftBase.TurbineLightDefault;

            //Effects
            ChangeWingTrailEffectThickness(0f);

            //Audio
            currentEngineSoundPitch = aircraftBase.DefaultSoundPitch;
        }

    }
    #endregion
}
```

▼ Aircraft Base

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[CreateAssetMenu(fileName = "Aircraft", menuName = "ScriptableObjects/Aircraft", order = 0)]
public class AircraftBase : ScriptableObject
{
    #region Controller
    #region Variables to be assigned
    [Header("Wing trail effects")]
    [Range(0.01f, 1f)] [SerializeField] private float trailThickness = 0.045f;

    [Header("Rotating speeds")]
    [Range(5f, 500f)] [SerializeField] private float yawSpeed = 50f;

    [Range(5f, 500f)] [SerializeField] private float pitchSpeed = 100f;

    [Range(5f, 500f)] [SerializeField] private float rollSpeed = 200f;

    [Header("Rotating speeds multiplers when turbo is used")]
    [Range(0.1f, 5f)] [SerializeField] private float yawTurboMultiplier = 0.3f;

    [Range(0.1f, 5f)] [SerializeField] private float pitchTurboMultiplier = 0.5f;

    [Range(0.1f, 5f)] [SerializeField] private float rollTurboMultiplier = 1f;

    [Header("Moving speed")]
    [Range(5f, 100f)] [SerializeField] private float defaultSpeed = 10f;

    [Range(10f, 200f)] [SerializeField] private float turboSpeed = 20f;

    [Range(0.1f, 50f)] [SerializeField] private float accelerating = 10f;

    [Range(0.1f, 50f)] [SerializeField] private float deaccelerating = 5f;

    [Header("Engine sound settings")]
    [SerializeField] private float defaultSoundPitch = 1f;

    [SerializeField] private float turboSoundPitch = 1.5f;

    [Header("Engine propellers settings")]
    [Range(10f, 10000f)] [SerializeField] private float propelSpeedMultiplier = 100f;

    [Header("Turbine light settings")]
    [Range(0.1f, 20f)] [SerializeField] private float turbineLightDefault = 1f;

    [Range(0.1f, 20f)] [SerializeField] private float turbineLightTurbo = 5f;
    #endregion

    #region Variables to be accessed
    public float TrailThickness { get { return trailThickness; } }
    public float YawSpeed { get { return yawSpeed; } }
    public float PitchSpeed { get { return pitchSpeed; } }
    public float RollSpeed { get { return rollSpeed; } }
    public float YawTurboMultiplier { get { return yawTurboMultiplier; } }
    public float PitchTurboMultiplier { get { return pitchTurboMultiplier; } }
    public float RollTurboMultiplier { get { return rollTurboMultiplier; } }
    public float DefaultSpeed { get { return defaultSpeed; } }
    public float TurboSpeed { get { return turboSpeed; } }
    public float Accelerating { get { return accelerating; } }
    public float Deaccelerating { get { return deaccelerating; } }
    public float DefaultSoundPitch { get { return defaultSoundPitch; } }
    public float TurboSoundPitch { get { return turboSoundPitch; } }
    public float PropelSpeedMultiplier { get { return propelSpeedMultiplier; } }
    public float TurbineLightDefault { get { return turbineLightDefault; } }
```

```
    public float TurbineLightTurbo { get { return turbineLightTurbo; } }
    #endregion
    #endregion

    #region Values to be assigned
    [SerializeField] private float maxSafeAltitude;
    [SerializeField] private float searchDistance;

    #endregion

    #region Values to be accesseds
    public float MaxSafeAltitude { get { return maxSafeAltitude; } }
    public float SearchDistance { get { return searchDistance; } }

    #endregion
}
```

▼ Aircraft Controller

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class AircraftCollider : MonoBehaviour
{
    public bool collideSometing;

    private void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.GetComponent<AircraftCollider>() == null)
        {
            collideSometing = true;
        }
    }
}
```

```
using UnityEngine;
using System.Collections.Generic;

[RequireComponent(typeof(Rigidbody))]
public class AircraftController : MonoBehaviour
{
    #region Variables that are protected
    protected List<AircraftCollider> airPlaneColliders = new List<AircraftCollider>();
    protected float maxSpeed = 0.6f;
    protected float currentYawSpeed;
    protected float currentPitchSpeed;
    protected float currentRollSpeed;
    protected float currentSpeed;
    protected float currentEngineLightIntensity;
    protected float currentEngineSoundPitch;
    protected bool planeIsDead;
    protected Rigidbody rb;

    #endregion

    #region Variables to be assigned
    [SerializeField] protected TrailRenderer[] wingTrailEffects;

    [SerializeField] protected AudioSource engineSoundSource;

    [SerializeField] protected GameObject[] propellers;

    [SerializeField] protected Light[] turbineLights;

    [SerializeField] protected Transform crashCollidersRoot;

    [SerializeField] protected AircraftBase aircraftBase;
    #endregion

    #region Audio
    protected void AudioSystem()
    {
        engineSoundSource.pitch = Mathf.Lerp(engineSoundSource.pitch, currentEngineSoundPitch, 10f * Time.deltaTime);

        if (planeIsDead)
        {
            engineSoundSource.volume = Mathf.Lerp(engineSoundSource.volume, 0f, 0.1f);
        }
    }
    #endregion

    #region Private methods

    protected List<Collider> crashColliders;

protected void SetupColliders(Transform _root)
{
    //Get colliders from root transform
    Collider[] colliders = _root.GetComponentsInChildren<Collider>();

    //If there are colliders put components in them
    for (int i = 0; i < colliders.Length; i++)
    {
        //Change collider to trigger
        colliders[i].isTrigger = true;

        GameObject _currentObject = colliders[i].gameObject;

        //Add airplane collider to it and put it on the list
        AircraftCollider _airplaneCollider = _currentObject.AddComponent<AircraftCollider>();
        airPlaneColliders.Add(_airplaneCollider);

        //Add rigid body to it
        Rigidbody _rb = _currentObject.AddComponent<Rigidbody>();
        _rb.useGravity = false;
        _rb.isKinematic = true;
        _rb.collisionDetectionMode = CollisionDetectionMode.ContinuousSpeculative;
    }

    crashColliders = new List<Collider>(crashCollidersRoot.GetComponentsInChildren<Collider>());
}

protected void RotatePropellers(GameObject[] _rotateThese)
{
    float _propelSpeed = currentSpeed * aircraftBase.PropelSpeedMultiplier;

    for (int i = 0; i < _rotateThese.Length; i++)
    {
        _rotateThese[i].transform.Rotate(Vector3.forward * -_propelSpeed * Time.deltaTime);
    }
}

protected void ControlEngineLights(Light[] _lights, float _intensity)
{
    float _propelSpeed = currentSpeed * aircraftBase.PropelSpeedMultiplier;
```

```
    for (int i = 0; i < _lights.Length; i++)
    {
        if (!planeIsDead)
        {
            _lights[i].intensity = Mathf.Lerp(_lights[i].intensity, _intensity, 10f * Time.deltaTime);
        }
        else
        {
            _lights[i].intensity = Mathf.Lerp(_lights[i].intensity, 0f, 10f * Time.deltaTime);
        }

    }
}

protected void ChangeWingTrailEffectThickness(float _thickness)
{
    for (int i = 0; i < wingTrailEffects.Length; i++)
    {
        wingTrailEffects[i].startWidth = Mathf.Lerp(wingTrailEffects[i].startWidth, _thickness, Time.deltaTime * 10f);
    }
}

protected bool HitSometing()
{
    for (int i = 0; i < airPlaneColliders.Count; i++)
    {
        if (airPlaneColliders[i].collideSometing)
        {
            return true;
        }
    }

    return false;
}

protected void Crash()
{
    //Set rigidbody to non cinematic
    rb.isKinematic = false;
    rb.useGravity = true;

    //Change every collider trigger state and remove rigidbodys
    for (int i = 0; i < airPlaneColliders.Count; i++)
    {
        airPlaneColliders[i].GetComponent<Collider>().isTrigger = false;
        Destroy(airPlaneColliders[i].GetComponent<Rigidbody>());
    }

    //Kill player
    planeIsDead = true;

    //Here you can add your own code...
    var tester = GameObject.FindGameObjectWithTag("GameController");
    if (tester)
    {

        tester.GetComponent<TestManager>().canRestart = true;
        tester.GetComponent<ConsoleManager>().TextUpdate("Press R to restart level", ConsoleTaskType.StayDefaultOff);
    }
}
#endregion

#region Variables
//Returns a percentage of how fast the current speed is from the maximum speed between 0 and 1
public float PercentToMaxSpeed()
{
    float _percentToMax = currentSpeed / aircraftBase.TurboSpeed;

    return _percentToMax;
}

public bool PlaneIsDead()
{
    return planeIsDead;
}

public bool UsingTurbo()
{
    if (maxSpeed == aircraftBase.TurboSpeed)
    {
        return true;
    }

    return false;
}

public float CurrentSpeed()
{
    return currentSpeed;
}
    #endregion
}
```

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Cinemachine;

public class AircraftCamera : MonoBehaviour
{
    [Header("References")]
    [SerializeField] private AircraftController airPlaneController;
    [SerializeField] private CinemachineFreeLook freeLook;
    [Header("Camera values")]
    [SerializeField] private float cameraDefaultFov = 60f;
    [SerializeField] private float cameraTurboFov = 40f;

    private void Start()
    {
        //Lock and hide mouse
        Cursor.lockState = CursorLockMode.Locked;
        Cursor.visible = false;
    }

    private void Update()
    {
        CameraFovUpdate();
    }

    private void CameraFovUpdate()
    {
        //Turbo
        if (!airPlaneController.PlaneIsDead())
        {
            if (Input.GetKey(KeyCode.LeftShift))
            {
                ChangeCameraFov(cameraTurboFov);
```

```
            }
            else
            {
                ChangeCameraFov(cameraDefaultFov);
            }
        }
    }

    public void ChangeCameraFov(float _fov)
    {
        float _deltatime = Time.deltaTime * 100f;
        freeLook.m_Lens.FieldOfView = Mathf.Lerp(freeLook.m_Lens.FieldOfView, _fov, 0.05f * _deltatime);
    }
}
```

▼ Player Controller

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PlayerController : AircraftController
{
    #region Base
    private void Start()
    {
        //Setup speeds
        maxSpeed = aircraftBase.DefaultSpeed;
        currentSpeed = aircraftBase.DefaultSpeed;

        //Get and set rigidbody
        rb = GetComponent<Rigidbody>();
        rb.isKinematic = true;
        rb.useGravity = false;
        rb.collisionDetectionMode = CollisionDetectionMode.ContinuousSpeculative;

        SetupColliders(crashCollidersRoot);
    }

    private void Update()
    {
        AudioSystem();

        //Airplane move only if not dead
        if (!planeIsDead)
        {
            Movement();
            Dyanmics();

            //Rotate propellers if any
            if (propellers.Length > 0)
            {
                RotatePropellers(propellers);
            }
        }
        else
        {
            ChangeWingTrailEffectThickness(0f);
        }

        //Control lights if any
        if (turbineLights.Length > 0)
        {
            ControlEngineLights(turbineLights, currentEngineLightIntensity);
        }

        //Crash
        if (!planeIsDead && HitSometing())
        {
            Crash();
        }
    }
    #endregion

    #region Movement
    private void Movement()
    {
        //Move forward
        transform.Translate(Vector3.forward * currentSpeed * Time.deltaTime);

        //Rotate airplane by inputs
        transform.Rotate(Vector3.forward * -Input.GetAxis("Horizontal") * currentRollSpeed * Time.deltaTime);
        transform.Rotate(Vector3.right * Input.GetAxis("Vertical") * currentPitchSpeed * Time.deltaTime);

        //Rotate yaw
        if (Input.GetKey(KeyCode.E))
        {
            transform.Rotate(Vector3.up * currentYawSpeed * Time.deltaTime);
        }
        else if (Input.GetKey(KeyCode.Q))
        {
            transform.Rotate(-Vector3.up * currentYawSpeed * Time.deltaTime);
        }
    }
    #endregion

    #region Dyanamics & Turbo
    private void Dyanmics()
    {
        //Accelerate and deacclerate
        if (currentSpeed < maxSpeed)
        {
            currentSpeed += aircraftBase.Accelerating * Time.deltaTime;
        }
        else
        {
            currentSpeed -= aircraftBase.Deaccelerating * Time.deltaTime;
        }

        //Turbo
        if (Input.GetKey(KeyCode.LeftShift))
        {
            //Set speed to turbo speed and rotation to turbo values
            maxSpeed = aircraftBase.TurboSpeed;

            currentYawSpeed = aircraftBase.YawSpeed * aircraftBase.YawTurboMultiplier;
            currentPitchSpeed = aircraftBase.PitchSpeed * aircraftBase.PitchTurboMultiplier;
            currentRollSpeed = aircraftBase.RollSpeed * aircraftBase.RollTurboMultiplier;

            //Engine lights
            currentEngineLightIntensity = aircraftBase.TurbineLightTurbo;

            //Effects
            ChangeWingTrailEffectThickness(aircraftBase.TrailThickness);

            //Audio
            currentEngineSoundPitch = aircraftBase.TurboSoundPitch;
```

```
        }
        else
        {
            //Speed and rotation normal
            maxSpeed = aircraftBase.DefaultSpeed;

            currentYawSpeed = aircraftBase.YawSpeed;
            currentPitchSpeed = aircraftBase.PitchSpeed;
            currentRollSpeed = aircraftBase.RollSpeed;

            //Engine lights
            currentEngineLightIntensity = aircraftBase.TurbineLightDefault;

            //Effects
            ChangeWingTrailEffectThickness(0f);

            //Audio
            currentEngineSoundPitch = aircraftBase.DefaultSoundPitch;
        }

    }
    #endregion
}
```

▼ Enemy Brain

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class EnemyBrain : MonoBehaviour
{
    #region Base Data
    private EnemyBase eBase;
    private SituationBase sBase;
    private AircraftBase aBase;

    public void Init(SituationBase sb, EnemyBase eb, AircraftBase ab)
    {
        eBase = eb;
        sBase = sb;
        aBase = ab;

        FixingBaseData();
        GenerateWayPoints();

        print(eBase.BehaviorAltitudePreference.StatGet);
    }

    private void FixingBaseData()
    {
        sBase.MakeAircraftNucleus(sBase.RoamRange, this.transform);
    }
    #endregion

    #region WayPoints
    [SerializeField] private bool randomWayPoints;
    [SerializeField] private float v1 = 10; //number of way points
    [SerializeField] private float v2 = 10; //distance between wayPoints
    public List<Vector3> wayPoints { get; set; }
    public void GenerateWayPoints()
    {
        if(randomWayPoints)
        {
            GenerateRandomWayPoints();
            return;
        }

        Vector3 currentWayPoint = this.transform.position;
        wayPoints = new List<Vector3>();
        for (int i = 0; i < v1; i++)
        {
            Vector3 newWayPoint = new Vector3();
            Vector3 lastWayPoint = (i == 0) ? currentWayPoint : wayPoints[i-1];

            //v1
            newWayPoint.z = lastWayPoint.z + v2;

            //v2
            newWayPoint.y = lastWayPoint.y + v3Analyzing(wayPoints, lastWayPoint);

            //v3
            newWayPoint.z = lastWayPoint.x + Random.Range(-1, 1);
            wayPoints.Add(newWayPoint);
        }
    }
    public void GenerateRandomWayPoints()
    {
        wayPoints = new List<Vector3>();
        for (int i = 0; i < v1; i++)
        {
            wayPoints.Add(GenerateSingleRandomWayPoint());
        }
    }
    public Vector3 GenerateSingleRandomWayPoint()
    {
        return BrainUtilities.RandomVectorInRange(sBase.RoamRange);
    }
    #endregion

    #region Analyzing Base Classes

    private float v3Analyzing(List<Vector3> currentWayPointsList, Vector3 lastWayPoint)
    {
        var msa = aBase.MaxSafeAltitude;
        var bap = eBase.BehaviorAltitudePreference.StatGet;

        int direction = (lastWayPoint.y >= bap*msa/100) ? 0 : 1;

        return direction*v2;
    }

    private float sensitivity = 1f;
    private System.Random random = new System.Random();
    public bool turboAnalyzing()
    {
        float offenseValue = eBase.PersonalityAgression.StatGet;
        float normalizedValue = offenseValue / 20f;
        float threshold = (float)random.NextDouble();
        if (threshold <= normalizedValue * sensitivity)
        {
            return true;
        }
        else
        {
            return false;
        }
```

```
    }

    #endregion

    #region Utility
    public static class BrainUtilities
    {
        public static Vector3 RandomVectorInRange(TheRange tRange)
        {
            Vector3 res;
            if (tRange.RangeType == RangeType.Cube)
            {
                Vector3 n = tRange.ShapeNucleus;
                float st = tRange.ST1;
                res = new Vector3(UnityEngine.Random.Range(n.x + st / 2, n.x - st / 2), UnityEngine.Random.Range(n.y + st / 2, n.y - st / 2), UnityEngine.Random.Range(n.z + st / 2, n.z - st / 2));
            }
            else if (tRange.RangeType == RangeType.Cuboid)
            {
                Vector3 n = tRange.ShapeNucleus;
                float st1 = tRange.ST1;
                float st2 = tRange.ST2;
                float st3 = tRange.ST3;
                res = new Vector3(UnityEngine.Random.Range(n.x + st1 / 2, n.x - st1 / 2), UnityEngine.Random.Range(n.y + st3 / 2, n.y - st3 / 2), UnityEngine.Random.Range(n.z + st2 / 2, n.z - st2 / 2)
            }
            else if (tRange.RangeType == RangeType.Sphere)
            {
                Vector3 n = tRange.ShapeNucleus;
                float r = tRange.ST1;
                float phi = UnityEngine.Random.Range(0, 2 * Mathf.PI);
                float theta = UnityEngine.Random.Range(0, Mathf.PI);
                float x = r * Mathf.Sin(theta) * Mathf.Cos(phi) + n.x;
                float y = r * Mathf.Sin(theta) * Mathf.Sin(phi) + n.y;
                float z = r * Mathf.Cos(theta) + n.z;
                res = new Vector3(x, y, z);
            }
            else
            {
                print("What is this range type bruh");
                return new Vector3(0, 0, 0);
            }
            return res;
        }
    }
    #endregion
}
```

▼ Enemy Controller

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[RequireComponent(typeof(EnemyBrain))]
public class EnemyController : AircraftController
{
    #region Base
    [SerializeField] private string heightPreference;
    private void Start()
    {
        //Setup speeds
        maxSpeed = aircraftBase.DefaultSpeed;
        currentSpeed = aircraftBase.DefaultSpeed;

        //Get and set rigidbody
        rb = GetComponent<Rigidbody>();
        rb.isKinematic = true;
        rb.useGravity = false;
        rb.collisionDetectionMode = CollisionDetectionMode.ContinuousSpeculative;

        SetupColliders(crashCollidersRoot);
        StartMovement();
    }

    public void Init()
    {
        EnemyBase randomEnemyBase = EnemyBase.CreateRandomEnemyBase();
        this.enemyBase = randomEnemyBase;
    }

    private void Update()
    {
        heightPreference = enemyBase.BehaviorAltitudePreference.StatGet.ToString();

        AudioSystem();

        //Airplane move only if not dead
        if (!planeIsDead)
        {
            SimpleMovement();
            Dyanmics();

            //Rotate propellers if any
            if (propellers.Length > 0)
            {
                RotatePropellers(propellers);
            }
        }
        else
        {
            ChangeWingTrailEffectThickness(0f);
        }

        //Control lights if any
        if (turbineLights.Length > 0)
        {
            ControlEngineLights(turbineLights, currentEngineLightIntensity);
        }

        //Crash
        if (!planeIsDead && HitSometing())
        {
            Crash();
        }
    }
    #endregion

    #region Movement
    [SerializeField] SituationBase situationBase;
    [SerializeField] EnemyBase enemyBase;
    private EnemyBrain enemyBrain;
    private int currentWaypointIndex = 0;
    public float sphereRadius = 2.0f; // Add this line
    private bool isTurbo = false;
    private void StartMovement()
    {
        enemyBrain = this.GetComponent<EnemyBrain>();
```

```
        enemyBrain.Init(situationBase, enemyBase, aircraftBase);
    }

    private void SimpleMovement()
    {
        CheckObstacle();
        isTurbo = enemyBrain.turboAnalyzing();

        // Get the current waypoint
        Vector3 currentWaypoint = enemyBrain.wayPoints[currentWaypointIndex];

        // Move towards the current waypoint
        Vector3 direction = currentWaypoint - transform.position;
        direction.Normalize();
        transform.position += direction * currentSpeed * Time.deltaTime;
        transform.rotation = Quaternion.Slerp(transform.rotation, Quaternion.LookRotation(direction), Time.deltaTime);

        // Check if the enemy has reached the current waypoint
        float distance = Vector3.Distance(transform.position, currentWaypoint);
        if (distance < 0.1f) // Adjust this value to match your desired threshold for reaching a waypoint
        {
            // Increment the current waypoint index
            currentWaypointIndex++;

            // Check if all waypoints have been visited
            if (currentWaypointIndex >= enemyBrain.wayPoints.Count)
            {
                // All waypoints have been visited, generate a new set of waypoints
                enemyBrain.GenerateWayPoints();

                // Reset the current waypoint in dex
                currentWaypointIndex = 0;
            }
        }
    }

    private void CheckObstacle()
    {
        foreach (var crashCollider in crashColliders)
        {
            Collider[] colliders = Physics.OverlapSphere(crashCollider.transform.position, crashCollider.bounds.size.magnitude);
            foreach (var collider in colliders)
            {
                if (collider.CompareTag("Obstacle"))
                {
                    enemyBrain.wayPoints[currentWaypointIndex] = enemyBrain.GenerateSingleRandomWayPoint();
                    break;
                }
            }
        }
    }

    #endregion

    #region Dyanamics & Turbo
    private void Dyanmics()
    {
        //Accelerate and deacclerate
        if (currentSpeed < maxSpeed)
        {
            currentSpeed += aircraftBase.Accelerating * Time.deltaTime;
        }
        else
        {
            currentSpeed -= aircraftBase.Deaccelerating * Time.deltaTime;
        }

        //Turbo
        if (isTurbo)
        {
            //Set speed to turbo speed and rotation to turbo values
            maxSpeed = aircraftBase.TurboSpeed;

            currentYawSpeed = aircraftBase.YawSpeed * aircraftBase.YawTurboMultiplier;
            currentPitchSpeed = aircraftBase.PitchSpeed * aircraftBase.PitchTurboMultiplier;
            currentRollSpeed = aircraftBase.RollSpeed * aircraftBase.RollTurboMultiplier;

            //Engine lights
            currentEngineLightIntensity = aircraftBase.TurbineLightTurbo;

            //Effects
            ChangeWingTrailEffectThickness(aircraftBase.TrailThickness);

            //Audio
            currentEngineSoundPitch = aircraftBase.TurboSoundPitch;
        }
        else
        {
            //Speed and rotation normal
            maxSpeed = aircraftBase.DefaultSpeed;
            currentYawSpeed = aircraftBase.YawSpeed;
            currentPitchSpeed = aircraftBase.PitchSpeed;
            currentRollSpeed = aircraftBase.RollSpeed;

            //Engine lights
            currentEngineLightIntensity = aircraftBase.TurbineLightDefault;

            //Effects
            ChangeWingTrailEffectThickness(0f);

            //Audio
            currentEngineSoundPitch = aircraftBase.DefaultSoundPitch;
        }

    }
    #endregion
}
```

▼ Test Manager

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class TestManager : MonoBehaviour
{
    [SerializeField] private GameObject enemyPrefab;
    [SerializeField] private Transform enemySpawnLocation;
    [SerializeField] private int maxEnemyCount = 1;
    [SerializeField] private int enemySpawnAllowWhen = 3;
    private int currentEnemyCount = 0;
    private bool allowEnemySpawn = false;
    public bool canRestart { get; set; } = false;
    [SerializeField] private string restartThisLevel;
```

```
    [SerializeField] private string nextLevel;
    ConsoleManager CM;

    private void Start()
    {
        CM = this.gameObject.GetComponent<ConsoleManager>();
        StartCoroutine(enemySpawn());
    }

    private IEnumerator enemySpawn()
    {
        yield return new WaitForSeconds(enemySpawnAllowWhen);
        CM.TextUpdate($"Number of enemies that can be created is {maxEnemyCount-currentEnemyCount} by pressing F key",  ConsoleTaskType.StayDefault);
        allowEnemySpawn = true;
    }

    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.F) && currentEnemyCount < maxEnemyCount && allowEnemySpawn)
        {
            GameObject enemyCreated = Instantiate(enemyPrefab, enemySpawnLocation.parent);
            enemyCreated.transform.position = enemySpawnLocation.position;
            enemyCreated.GetComponent<EnemyController>().Init();

            currentEnemyCount += 1;
            CM.TextUpdate("Enemy Created!", ConsoleTaskType.Temp, 3);
        }

        if (Input.GetKeyDown(KeyCode.R) && canRestart)
        {
            SceneManager.LoadScene(restartThisLevel);
        }
        if (Input.GetKeyDown(KeyCode.N) && nextLevel.Length > 1)
        {
            SceneManager.LoadScene(nextLevel);
        }
    }
```

▼ Console Manager

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using TMPro;

public enum ConsoleTaskType
{
    Temp,
    Stay,
    StayDefault,
    StayDefaultOff
}
public class ConsoleManager : MonoBehaviour
{
    [SerializeField] private TextMeshProUGUI consoleTMP;
    private string defaultText;
    public bool isWorking { get; set; } = true;
    public void TextUpdate(string newText, ConsoleTaskType ctt, int deleteWhen = 0)
    {
        if (!isWorking) { return; }
        consoleTMP.text = newText;

        if(ctt == ConsoleTaskType.Stay) { return; }
        if(ctt == ConsoleTaskType.StayDefault) { defaultText = newText; return; }
        if (ctt == ConsoleTaskType.StayDefaultOff) { defaultText = newText; isWorking = false; return; }

        StartCoroutine(TextUpdateWorker(Mathf.Clamp(deleteWhen, 0, Mathf.Abs(deleteWhen))));

    }
    private IEnumerator TextUpdateWorker(int deleteWhen)
    {
        yield return new WaitForSeconds(deleteWhen);
        ClearConsole();
    }
    private void ClearConsole()
    {
        consoleTMP.text = defaultText;
    }

    private void Start()
    {
        TextUpdate("Console messages are here", ConsoleTaskType.StayDefault);
    }
}
```

▼ **CONCLUSION**

1. Aircraft controller basic for building initial AI ✅
2. Poor Object avoidance for enemy AI ✅
3. Idea about attributes for enemy, situation and aircraft base classes ✅
4. Spawning mechanisms and setup build ✅
5. Testing single behavior ✅
    a. Multiple aircrafts fly in the same height
    b. Roam around range
    c. They stop at calculated preffered height by reading situation, enemy and aircraft base classes