❎

Stage 7

- ▼ **SCENES (Workplace)**
    1. **Zero** ⇒ *Empty without game objects.*
    2. **One** ⇒ *Home Menu. Integration of obstacle avoidance, unique pattern generation & action decisions.*
- ▼ **HIERARCHY (Workflow)**
    - ▼ **Domain**

      (Mostly lighting, prefabs and reflection probes)

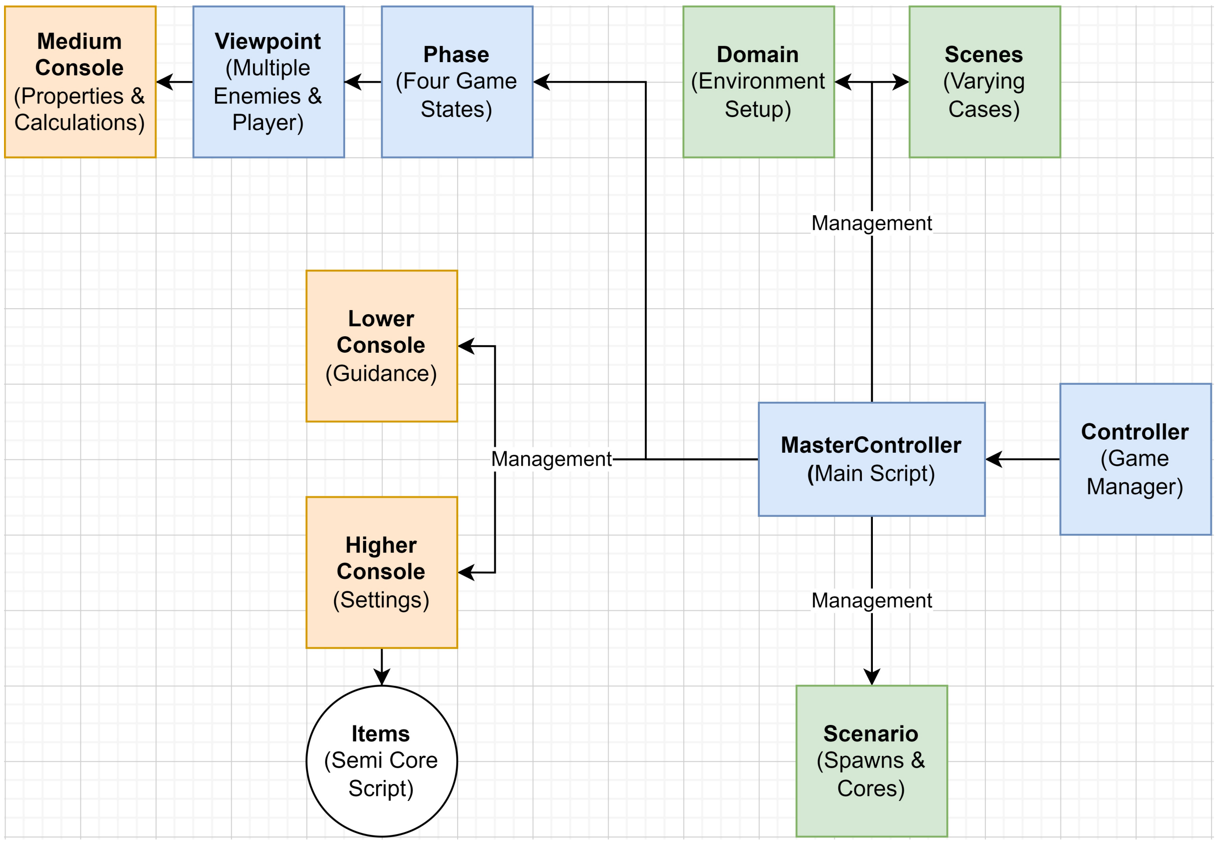
        1. Directional Light (Children)
        2. Buildings (Children)
        3. Land (Children)
    - ▼ **Controller**

      (Testing manager and console manager)

        1. Empty Camera (Children)
        2. Console Canvas (Children)
            a. Lower Console (Children)
            b. Higher Console (Children)
        3. Master Controller (Component Script)
        4. Event System (Children)
    - ▼ **Units**

      (Units like players and enemies)

        1. Player Position (Children)
        2. Enemy Position (Children)



Medium Console (Properties & Calculations)
Viewpoint (Multiple Enemies & Player)
Phase (Four Game States)
Domain (Environment Setup)
Scenes (Varying Cases)
Management
Lower Console (Guidance)
Management
MasterController (Main Script)
Controller (Game Manager)
Higher Console (Settings)
Management
Items (Semi Core Script)
Scenario (Spawns & Cores)

▼ DIRECTORY

1. **Domain**
   a. Aircraft (Package) ⇒ FBX, Materials, Prefabs, Sounds
   b. City (Package) ⇒ Models, Prefabs, Shaders, Skyboxes

   > 💡 Post-integration ⇒ The whole environment package will be here with multiple lightings and extra features. High-poly aircrafts and resources for other effects will also be here.

2. **Control**
   a. TMP ⇒ Fonts, Resources, Shaders, Sprites
   b. Controlling Scripts/Prefabs
      i. Master Controller
      ii. Higher Console Item Semi Core
      iii. Core Sequence
3. **Scenes**
   a. Zero ⇒ Empty
   b. One ⇒ Basic
4. **Units**
   a. Aircraft
   b. Enemy

c. Player

d. Situation

▼ SCRIPTS

▼ Control

▼ Master Controller

```
//Outside public class MasterController : MonoBehaviour
#region Using libraries
using UnityEngine; //Import Unity Engine library for Unity functions and types
using System.Collections.Generic; //Import System.Collections.Generic library for List and Dictionary
using TMPro; //Import TextMesh Pro library for text rendering
using System.Collections; //Import System.Collections library for IEnumerator and IEnumerable
using UnityEngine.SceneManagement; //Import Unity SceneManagement library for managing scenes
using System.Text; //Import System.Text library for string manipulation
using System.Reflection; //Import System.Reflection library for obtaining information about an object's type and its membe
#endregion

#region Phases of the whole test run
// This enumerated type describes the different phases of the test.
public enum GamePhase
{
    Preparation, // The preparation phase, before runtime
    Execution, // The execution phase, during test runtime
    Intermission, // The intermission phase, pausing test runtime
    Termination // The termination phase, after test runtime
}
#endregion

#region Item class for Preparation settings
// This class represents an item in the higher console, used for preparation settings.
[System.Serializable] // For making a list of these objects
public class HigherConsoleItem
{
    // This enumerated type describes the different types of settings that can be added in the higher console.
    public enum ItemType
    {
        intItem, // An integer setting
        floatItem, // A float setting
        stringItem, // A string setting
        aircraftItem, // An aircraft setting
        enemyItem, // An enemy setting
        situationItem // A situation setting
    }

    // This enumerated type describes the different names of the items in the higher console.
    public enum ItemName
    {
        maxEnemySpawn, // The maximum number of enemy spawns
        enemyAircraftCore, // The core type of enemy aircraft
        enemySituationCore, // The core type of enemy situation
        enemyEnemyCore, // The core type of enemy enemy
        playerAircraftCore, // The core type of player aircraft
        playerSpeedMultiplier // The speed multiplier of the player
    }

    [SerializeField] private ItemType consoleItemType; // The type of the console item
    public ItemType ConsoleItemType { get { return consoleItemType; } }

    [SerializeField] private ItemName consoleItemName; // The name of the console item
    public ItemName ConsoleItemName { get { return consoleItemName; } }

    [SerializeField] private string consoleInstruction; // The instruction of the console item
    public string ConsoleInstruction { get { return consoleInstruction; } }

    public GameObject ItemSpawned { get; set; } // The spawned item in the game
    public HigherConsoleItemSemiCore HCISemiCore { get { return ItemSpawned.GetComponent<HigherConsoleItemSemiCore>(); } }
}
#endregion
```

```
#region Lower, medium and higher console setup
    /// <summary>
    /// Handles the setup and management of the lower, medium and higher console interfaces.
    /// </summary>
    [Header("[Console]")]
    [SerializeField] private Canvas consoleCanvas; //The canvas containing the console interfaces
    [SerializeField] private TextMeshProUGUI lowerConsoleText; //The text element in the lower console interface
    [SerializeField] private Transform higherConsoleContentTrasform; //The transform containing the items in the higher co
    [SerializeField] private List<HigherConsoleItem> higherConsoleItemList; //The list of items in the higher console inte
    [SerializeField] private GameObject higherConsoleItemPrefab; //The prefab used to instantiate new items in the higher
    [SerializeField] private TextMeshProUGUI mediumConsoleText; //The text element in the medium console interface
    [SerializeField] private GameObject higherConsole; //The higher console interface
    [SerializeField] private GameObject lowerConsole; //The lower console interface
    [SerializeField] private GameObject mediumConsole; //The medium console interface

    //Helpers
    private string lowerConsoleDefaultText; //The default text displayed in the lower console interface
    public bool lowerConsoleIsWorking { get; set; } //Indicates if the lower console interface is currently functioning
    public enum LowerConsoleTaskType //The different types of tasks that can be assigned to the lower console interface
    {
        Temp,
        Stay,
        StayDefault,
        StayDefaultOff
    }
    #endregion

    #region General management setup
    [Header("[Level]")]
    [SerializeField] private string nextLevel; //The name of the next level to load

    [Header("[Essentials]")]
    [SerializeField] private Camera emptyCamera; //The camera used for displaying the opening settings canvas
    [SerializeField] private GameObject domainRoot; //The root object for the entire environment
    [SerializeField] private GameObject unitsRoot; //The root object for all units in the game

    //Helpers
    public GamePhase CurrentGamePhase { get; set; } //The current game phase
    private int currentViewpointIndex; //The index of the current viewpoint
    private Transform currentActiveCameraRoot; //The root object for the current active camera
    #endregion

    #region Spawning units setup
    [Header("[Player Spawn]")]
    [SerializeField] GameObject currentPlayer; //The player object
    [SerializeField] private Transform playerSpawnLocation; //The location where the player will spawn
    [SerializeField] private float playerSpeedMultiplier; //The speed multiplier for the player object

    [Header("[Enemy Spawn]")]
    [SerializeField] private int maxEnemyCount = 1; //The maximum number of enemies that can be spawned
    [SerializeField] private GameObject enemyPrefab; //The enemy prefab
    [SerializeField] private Transform enemySpawnLocation; //The location where enemies will spawn

    [Header("[Cores]")]
    [SerializeField] private List<AircraftCore> aircraftCores = new List<AircraftCore>(); //The list of aircraft cores
    [SerializeField] private List<SituationCore> situationCores = new List<SituationCore>(); //The list of situation cores
    [SerializeField] private List<EnemyCore> enemyCores = new List<EnemyCore>(); //The list of enemy cores

    //Helpers
    private int currentEnemyCount = 0; //The current number of enemies
    private bool allowEnemySpawn = true; //Indicates if enemy spawning is allowed
    private List<GameObject> enemySpawnedList = new List<GameObject>(); //The list of spawned enemy objects
    private GameObject playerSpawned; //The player object that has been spawned

    private AircraftCore selectedAircraftCore;
    private SituationCore selectedSituationCore;
    private EnemyCore selectedEnemyCore;

    #endregion
```

```
#region Basic
    // This function is called in the beginning of the session
    private void Awake()
    {
        PreparationRun();
    }

    // This function is called once per frame by Unity
    void Update()
    {
        // If the player presses the "R" key, and the game is in the "Termination" phase, restart the game
        if (Input.GetKeyDown(KeyCode.R))
        {
            if (CurrentGamePhase == GamePhase.Termination)
            {
                // Reload the current scene to restart the game
                SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);
            }
        }

        // If the player presses the "M" key, and the game is in the "Intermission" phase, start the "Execution" phase and
        if (Input.GetKeyDown(KeyCode.M))
        {
            if (CurrentGamePhase == GamePhase.Intermission)
            {
                // Set the current game phase to "Execution"
                CurrentGamePhase = GamePhase.Execution;

                // Call the "ExecutionContinue" function to begin the "Execution" phase
                ExecutionContinue();
            }
        }

        // If the player presses the "J" key...
        if (Input.GetKeyDown(KeyCode.J))
        {
            // ...and the game is in the "Execution" phase and there aren't already the maximum number of enemies in the g
            if (CurrentGamePhase == GamePhase.Execution && currentEnemyCount < maxEnemyCount)
            {
                // Add a new enemy to the game by instantiating the enemyPrefab at the enemySpawnLocation, and add it to t
                SpawnEnemy(enemyPrefab, enemySpawnLocation);
            }

            // ...or if the game is in the "Intermission" phase, switch the viewpoint
            if (CurrentGamePhase == GamePhase.Intermission)
            {
                // Call the "SwitchViewPoint" function to switch the viewpoint
                SwitchViewPoint();
            }
        }

        // If the player presses the "N" key, check the current game phase and execute the appropriate code
        if (Input.GetKeyDown(KeyCode.N))
        {
            // Call the "PhaseCheck" function to check the current game phase and execute the appropriate code
            PhaseCheck();
        }
    }

    // This function checks the current game phase and executes the appropriate code
    private void PhaseCheck()
    {
        switch (CurrentGamePhase)
        {
            // If the game is in the "Preparation" phase, call the "ExecutionRun" function to begin the "Execution" phase
            case GamePhase.Preparation:
                ExecutionRun();
                break;

            // If the game is in the "Execution" phase, call the "IntermissionRun" function to begin the "Intermission" ph
            case GamePhase.Execution:
                IntermissionRun();
                break;

            // If the game is in the "Intermission" phase, call the "TerminationRun" function to begin the "Termination" p
            case GamePhase.Intermission:

                TerminationRun();
                break;

            // If the game is in the "Termination" phase, load the next level
            case GamePhase.Termination:
                SceneManager.LoadScene(nextLevel);
                break;
        }
    }

    //This function helps in running phases majorly for intermission and execution
    private void PhaseRunnerHelper1(bool exec)
    {
        //During execution the cursor is locked and during intermisison the time scale is zero
        PhaseRunnerhelper2(exec);
        //Switching between execution and intermission
        mediumConsole.gameObject.SetActive(!exec);
    }

    private void PhaseRunnerhelper2(bool lockCursor)
    {
        //Lock cursor or not
        if (lockCursor) { Cursor.lockState = CursorLockMode.Locked; Time.timeScale = 1; }
        else { Cursor.lockState = CursorLockMode.None; Time.timeScale = 0; }
        Cursor.visible = !lockCursor;
    }
    #endregion
```

```
#region Running preperation
    private void PreparationRun()
    {
        // Check for duplicate ConsoleItemName in higherConsoleItemList
        CheckItemList(higherConsoleItemList);

        // Activate domainRoot and deactivate unitsRoot
        domainRoot.SetActive(true);
        unitsRoot.SetActive(false);

        // Activate emptyCamera and consoleCanvas
        emptyCamera.gameObject.SetActive(true);
        consoleCanvas.gameObject.SetActive(true);

        // Deactivate mediumConsole and set lowerConsoleIsWorking to true
        mediumConsole.gameObject.SetActive(false);
        lowerConsoleIsWorking = true;

        // Instantiate HigherConsoleItemSemiCore for each item in higherConsoleItemList
        foreach (HigherConsoleItem HCI in higherConsoleItemList)
        {
            GameObject newHCI = Instantiate(higherConsoleItemPrefab, higherConsoleContentTrasform);
            HigherConsoleItemSemiCore newHCISemiCore = newHCI.GetComponent<HigherConsoleItemSemiCore>();

            // Set ConsoleInstructionText and input restrictions
            newHCISemiCore.ConsoleInstructionText = HCI.ConsoleInstruction;
            RestrictInput(HCI, newHCISemiCore);

            // Save the spawned HigherConsoleItem as ItemSpawned in HigherConsoleItem
            HCI.ItemSpawned = newHCI;
        }

        // Set initial text for lower console and currentViewpointIndex to 0
        LowerConsoleUpdate($"Welcome to scene - {SceneManager.GetActiveScene().name}. Current Phase - {CurrentGamePhase}.\
        currentViewpointIndex = 0;
    }

    // Check for duplicate ConsoleItemName in higherConsoleItemList
    private void CheckItemList(List<HigherConsoleItem> HCIL)
    {
        HashSet<HigherConsoleItem.ItemName> names = new HashSet<HigherConsoleItem.ItemName>();
        foreach (HigherConsoleItem item in HCIL)
        {
            if (!names.Add(item.ConsoleItemName))
            {
                Debug.LogError("Fail#1: Two or more HigherConsoleItems have the same ConsoleItemName: " + item.ConsoleItem
            }
        }
    }

    // Fill a TMP_Dropdown with options of type T
    public void FillDropdown<T>(TMP_Dropdown dropdown, List<T> options) where T : UnityEngine.Object
    {
        // Clear the dropdown's current options
        dropdown.ClearOptions();

        // Create a new list of dropdown options
        List<TMP_Dropdown.OptionData> dropdownOptions = new List<TMP_Dropdown.OptionData>();

        // Loop through the list of options and add them to the dropdown
        foreach (T option in options)
        {
            string optionName = option.name;
            TMP_Dropdown.OptionData newOption = new TMP_Dropdown.OptionData(optionName);
            dropdownOptions.Add(newOption);
        }

        // Add the options to the dropdown
        dropdown.AddOptions(dropdownOptions);
    }

    public void RestrictInput(HigherConsoleItem HCI, HigherConsoleItemSemiCore newHCISemiCore)
    {
        // Get the input field component of the new console item
        TMP_InputField inputField = newHCISemiCore.ConsoleInputField;

        // Depending on the type of the console item, restrict the input field to the appropriate data type and/or fill a
        switch (HCI.ConsoleItemType)
        {
            case HigherConsoleItem.ItemType.intItem:
                // If the console item is an integer item, enable the input field and restrict it to integer numbers
                newHCISemiCore.InputOrDropdown(true);
                inputField.contentType = TMP_InputField.ContentType.IntegerNumber;
                break;
            case HigherConsoleItem.ItemType.floatItem:
                // If the console item is a float item, enable the input field and restrict it to decimal numbers
                newHCISemiCore.InputOrDropdown(true);
                inputField.contentType = TMP_InputField.ContentType.DecimalNumber;
                break;
            case HigherConsoleItem.ItemType.stringItem:
                // If the console item is a string item, enable the input field and allow standard text input
                newHCISemiCore.InputOrDropdown(true);
                inputField.contentType = TMP_InputField.ContentType.Standard;
                break;
            case HigherConsoleItem.ItemType.aircraftItem:
                // If the console item is an aircraft item, disable the input field and fill the dropdown with available a
                newHCISemiCore.InputOrDropdown(false);
                FillDropdown(newHCISemiCore.ConsoleDropdown, aircraftCores);
                break;
            case HigherConsoleItem.ItemType.situationItem:
                // If the console item is a situation item, disable the input field and fill the dropdown with available s
                newHCISemiCore.InputOrDropdown(false);
                FillDropdown(newHCISemiCore.ConsoleDropdown, situationCores);
                break;
            case HigherConsoleItem.ItemType.enemyItem:
                // If the console item is an enemy item, disable the input field and fill the dropdown with available enem
                newHCISemiCore.InputOrDropdown(false);
                FillDropdown(newHCISemiCore.ConsoleDropdown, enemyCores);
                break;
            default:
                break;
        }
    }
    #endregion
```

```
#region Running lower console
    public void LowerConsoleUpdate(string newText, LowerConsoleTaskType taskType, int deleteWhen = 0, bool? featureToTurnO
    {
        // If lower console is not working, don't update it.
        if (!lowerConsoleIsWorking) { return; }

        // Set the new text to the lower console text object.
        lowerConsoleText.text = newText;

        // Handle different task types.
        if (taskType == LowerConsoleTaskType.Stay) { return; } // Keep the current text.
        if (taskType == LowerConsoleTaskType.StayDefault) { lowerConsoleDefaultText = newText; return; } // Set default te
        if (taskType == LowerConsoleTaskType.StayDefaultOff) { lowerConsoleDefaultText = newText; lowerConsoleIsWorking =

        // Turn off the feature, if specified.
        if (featureToTurnOff != null)
        {
            featureToTurnOff = false;
        }

        // Start the coroutine to delete the text after a specified delay.
        StartCoroutine(TextUpdateWorker(Mathf.Clamp(deleteWhen, 0, Mathf.Abs(deleteWhen)), featureToTurnOff));
    }

    private IEnumerator TextUpdateWorker(int deleteWhen, bool? featureToTurnOff)
    {
        // Turn off the feature, if specified.
        if (featureToTurnOff != null)
        {
            featureToTurnOff = false;
        }

        // Wait for the specified delay.
        yield return new WaitForSeconds(deleteWhen);

        // Clear the lower console text to the default text.
        ClearConsole();

        // Turn the feature back on, if specified.
        if (featureToTurnOff != null)
        {
            featureToTurnOff = true;
        }
    }

    private void ClearConsole()
    {
        // Set the lower console text to the default text.
        lowerConsoleText.text = lowerConsoleDefaultText;
    }
    #endregion
```

```
#region Running execution
    private void ExecutionRun()
    {
        CurrentGamePhase = GamePhase.Execution;
        // Hide empty camera
        emptyCamera.gameObject.SetActive(false);

        // Activate PhaseRunnerHelper1
        PhaseRunnerHelper1(true);

        // Hide higher console and activate lower console
        higherConsole.gameObject.SetActive(false);
        lowerConsoleIsWorking = true;

        // Activate units root
        unitsRoot.SetActive(true);

        // Spawn player
        playerSpawned = Instantiate(currentPlayer, playerSpawnLocation);

        // Get current active camera root
        currentActiveCameraRoot = playerSpawned.GetComponent<PlayerController>().CameraRoot;

        // Read settings after spawning
        ReadSettings();

        // Update lower console with session start message
        LowerConsoleUpdate($"Session has started successfully. Current Phase - {CurrentGamePhase}\nPress N for Intermissio
    }

    private void SpawnEnemy(GameObject enemy, Transform where, bool console = true)
    {
         if (!allowEnemySpawn) { return; }

        // Add spawned enemy to the list and increase enemy count
        GameObject newEnemy = Instantiate(enemy, where);
        newEnemy.GetComponent<EnemyController>().Init(selectedAircraftCore, selectedSituationCore, selectedEnemyCore);
        enemySpawnedList.Add(newEnemy);
        currentEnemyCount += 1;

        if (!console) { return; }
        LowerConsoleUpdate($"Enemy Spawned. Current Phase - {CurrentGamePhase}\nPress N for Intermission. Press J for Enem
    }

    private void ExecutionContinue()
    {
        CurrentGamePhase = GamePhase.Execution;
        // Activate PhaseRunnerHelper1
        PhaseRunnerHelper1(true);

        // Update lower console with session start message
        LowerConsoleUpdate($"Session has resumed successfully. Current Phase - {CurrentGamePhase}\nPress N for Intermissio
    }
    private void SetSettings<T>(T what, ref T where)
    {
        // Check if 'what' is empty or not
        if (EqualityComparer<T>.Default.Equals(what, default(T)))
        {
            // If empty, return
            return;
        }
        // Assign 'what' to 'where'
        where = what;
    }

    private void ReadSettings()
    {
        // Loop through all the Higher Console items in the list.
        foreach (HigherConsoleItem HCI in higherConsoleItemList)
        {
            // If the Higher Console item is for the max enemy spawn setting:
            if (HCI.ConsoleItemName == HigherConsoleItem.ItemName.maxEnemySpawn)
            {
                // Try to parse the value from the input field as an integer.
                if (int.TryParse(HCI.HCISemiCore.ConsoleInputField.text, out int a))
                {
                    // If the parsing was successful, set the 'maxEnemyCount' variable to the parsed value.
                    SetSettings(a, ref maxEnemyCount);
                }
            }
            // If the Higher Console item is for the player speed multiplier setting:
            else if (HCI.ConsoleItemName == HigherConsoleItem.ItemName.playerSpeedMultiplier)
            {
                // Try to parse the value from the input field as a float.
                if (float.TryParse(HCI.HCISemiCore.ConsoleInputField.text, out float a))
                {
                    // If the parsing was successful, set the 'playerSpeedMultiplier' variable to the parsed value.
                    SetSettings(a, ref playerSpeedMultiplier);
                }
            }
            // If the Higher Console item is for the enemy aircraft core setting:
            else if (HCI.ConsoleItemName == HigherConsoleItem.ItemName.enemyAircraftCore)
            {
                // Get the selected option index from the dropdown.
                int selectedOptionIndex = HCI.HCISemiCore.ConsoleDropdown.value;

                // Get the selected aircraft core from the 'aircraftCores' list using the selected option index.
                AircraftCore selectedAircraftCore = aircraftCores[selectedOptionIndex];
                this.selectedAircraftCore = selectedAircraftCore;

            }
            // If the Higher Console item is for the enemy situation core setting:
            else if (HCI.ConsoleItemName == HigherConsoleItem.ItemName.enemySituationCore)
            {
                // Get the selected option index from the dropdown.
                int selectedOptionIndex = HCI.HCISemiCore.ConsoleDropdown.value;

                // Get the selected situation core from the 'situationCores' list using the selected option index.
                SituationCore selectedSituationCore = situationCores[selectedOptionIndex];
                this.selectedSituationCore = selectedSituationCore;
            }
            // If the Higher Console item is for the enemy enemy core setting:
            else if (HCI.ConsoleItemName == HigherConsoleItem.ItemName.enemyEnemyCore)
            {
                // Get the selected option index from the dropdown.
                int selectedOptionIndex = HCI.HCISemiCore.ConsoleDropdown.value;

                // Get the selected enemy core from the 'enemyCores' list using the selected option index.
                EnemyCore selectedEnemyCore = enemyCores[selectedOptionIndex];

                this.selectedEnemyCore = selectedEnemyCore;

            }
            // if the console item is for player aircraft core
            else if (HCI.ConsoleItemName == HigherConsoleItem.ItemName.playerAircraftCore)
            {
                // get the selected option index from the dropdown
                int selectedOptionIndex = HCI.HCISemiCore.ConsoleDropdown.value;
                // get the corresponding aircraft core from the list
                AircraftCore selectedAircraftCore = aircraftCores[selectedOptionIndex];
                // update the player's aircraft core
                playerSpawned.GetComponent<PlayerController>().CoreUpdate(selectedAircraftCore);
            }
        }
    }
    #endregion
```

```
#region Running intermission
    // This function is called when the game is in the "Intermission" phase
    // During this phase the session has been paused and the user can do data analysis of different enemies spawned by swi
    private void IntermissionRun()
    {
        CurrentGamePhase = GamePhase.Intermission;

        // Call the "PhaseRunnerHelper1" function with "false" as the argument to disable or enable certain UI elements an
        PhaseRunnerHelper1(false);

        // Disable the higher console game object
        higherConsole.gameObject.SetActive(false);

        MediumConsoleUpdate();

        // Update the lower console with a message indicating that the session has been paused and telling the player how
        LowerConsoleUpdate($"Session has been paused successfully. Current Phase - {CurrentGamePhase}\nPress N for Termina
    }

    // This function is called when the player presses the "J" key during the "Execution" or "Intermission" phases to swit
    public void SwitchViewPoint()
    {
        // Increment the current viewpoint index by 1
        currentViewpointIndex += 1;

        // This function works like there is a list of all the aircrafts, 0 being player and 1-x being enimies where x is
        // Number of aircrafts = Number of enemies (Index wise) because we have included 0 as player in the former

        // If the current viewpoint index is greater than the number of enemies in the "enemySpawnedList", reset the index
        if (currentViewpointIndex > enemySpawnedList.Count)
        {
            currentViewpointIndex = 0;
        }

        // If there is a currently active camera root, disable it
        if (currentActiveCameraRoot != null)
        {
            currentActiveCameraRoot.gameObject.SetActive(false);
        }

        // If the current viewpoint index is 0, set the active camera root to the player's camera root
        if (currentViewpointIndex == 0)
        {
            currentActiveCameraRoot = playerSpawned.GetComponent<PlayerController>().CameraRoot;
        }

        // Otherwise, set the active camera root to the camera root of the enemy at the current viewpoint index minus 1 (s
        else
        {
            currentActiveCameraRoot = enemySpawnedList[currentViewpointIndex - 1].GetComponent<EnemyController>().CameraRo
        }

        // Enable the game object associated with the current active camera root
        currentActiveCameraRoot.gameObject.SetActive(true);

        MediumConsoleUpdate();
    }

    private void MediumConsoleUpdate()
    {
        string res = "";

        // If current viewpoint index is 0, display the cores of player
        if (currentViewpointIndex == 0)
        {
            res += ReadCore(playerSpawned.GetComponent<PlayerController>().MyAicraftCore);
        }
        // Otherwise,  display the cores of enemy
        else
        {
            res += ReadCore(enemySpawnedList[currentViewpointIndex - 1].GetComponent<EnemyController>().MyAicraftCore);
            res += "\n" + ReadCore(enemySpawnedList[currentViewpointIndex - 1].GetComponent<EnemyController>().MyEnemyCore
        }
        // Update the text of the medium console with the result
        mediumConsoleText.text = res;
    }

    // Read and return a string representation of the properties of the given ScriptableObject
    private string ReadCore(ScriptableObject scriptableObject)
    {
        StringBuilder sb = new StringBuilder();

        // Get all the fields (including private and public) of the given ScriptableObject type
        var properties = scriptableObject.GetType().GetFields(BindingFlags.Instance | BindingFlags.Public | BindingFlags.N

        // For each field, append its name and value to the StringBuilder
        foreach (var property in properties)
        {
            var value = property.GetValue(scriptableObject);
            sb.AppendLine($"{property.Name} = {value}");
        }

        // Return the final string representation of the properties
        return sb.ToString();
    }

    #endregion

#region Running termination
    // This function is called when the game is in the "Termination" phase
    // During this phase the session has ended and the user just needs to choose between restarting the current scene or l
    public void TerminationRun()
    {
        CurrentGamePhase = GamePhase.Termination;

        PhaseRunnerhelper2(false);

        // Disable the medium console game object as viewpoint data analyisis feature is not required in termination
        mediumConsole.gameObject.SetActive(false);

        // Disable the "domainRoot" and "unitsRoot" game objects
        domainRoot.SetActive(false);
        unitsRoot.SetActive(false);

        // Enable the "emptyCamera" game object
        emptyCamera.gameObject.SetActive(true);

        // Update the lower console with a message indicating that the session has been terminated successfully and tellin
        LowerConsoleUpdate($"Session has been terminated successfully. Current Phase - {CurrentGamePhase}\nPress N for goi

        // Set the "lowerConsoleIsWorking" variable to false
        lowerConsoleIsWorking = false;
    }
    #endregion
}
```

▼ Core Sequence

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

[CreateAssetMenu(fileName = "SequenceName", menuName = "ScriptableObjects/Sequence", order = 1)]
public class CoreSequence : ScriptableObject
{
    public static List<EnemyCore> randomEnemyCores = new List<EnemyCore>();

    #region Randomize
    public static int CreateRandomEnemyCore()
    {
        EnemyCore EnemyCore = CreateInstance<EnemyCore>();

        PropertyInfo[] properties = typeof(EnemyCore).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);
                    property.SetValue(EnemyCore, ma);
                    //Debug.Log(ma.StatBase);
                }
                else if (property.PropertyType == typeof(bool))
                {
                    property.SetValue(EnemyCore, Random.value > 0.5f);
                }
                // Add more property types if necessary
            }
        }

        randomEnemyCores.Add(EnemyCore);
        return randomEnemyCores.Count-1;
    }
    #endregion
}
```

▼ Higher Console Item Semi Core

```
using UnityEngine; //Provides functionality for working with Unity game engine.
using TMPro; //Provides TextMeshPro assets for rendering high-quality text in Unity projects.

public class HigherConsoleItemSemiCore : MonoBehaviour
{
    // This class manages the UI elements of the console items in the higher console.

    // Text element that displays the instruction for the console item.
    [SerializeField] private TextMeshProUGUI consoleInstructionText;
    // Property to get or set the text of the instruction.
    public string ConsoleInstructionText { get { return consoleInstructionText.text; } set { consoleInstructionText.text =

    // Input field for console items that take input.
    [SerializeField] private TMP_InputField consoleInputField;
    // Property to get or set the input field.
    public TMP_InputField ConsoleInputField { get { return consoleInputField; } set { consoleInputField = value; } }

    // Dropdown for console items that take a selection from a list.
    [SerializeField] private TMP_Dropdown consoleDropdown;
    // Property to get or set the dropdown.
    public TMP_Dropdown ConsoleDropdown { get { return consoleDropdown; } set { consoleDropdown = value; } }

    // Method to switch between input field and dropdown based on parameter.
    public void InputOrDropdown(bool inputOpen)
    {
        consoleDropdown.gameObject.SetActive(!inputOpen);
        consoleInputField.gameObject.SetActive(inputOpen);
    }
}
```

▼ Units Base

```
using UnityEngine; // Required to access Unity's GameObject, MonoBehaviour and other important classes.
using Cinemachine; // Required to access Cinemachine camera classes for Unity.

public class AircraftCamera : MonoBehaviour
{
    #region Variables that are assigned
    [Header("References")]
    [SerializeField] private AircraftController airPlaneController; // Reference to the AircraftController script
    [SerializeField] private CinemachineFreeLook freeLook; // Reference to the CinemachineFreeLook component
    [Header("Camera values")]
    [SerializeField] private float cameraDefaultFov = 60f; // The default field of view for the camera
    [SerializeField] private float cameraTurboFov = 40f; // The field of view for the camera when the aircraft is in turbo mod
    #endregion

    #region Basic
    private void Update()
    {
        CameraFovUpdate();
    }

    // Updates the field of view of the camera based on the input from the player
    private void CameraFovUpdate()
    {
        // Turbo
        if (!airPlaneController.PlaneIsDead()) // Check if the aircraft is not dead
        {
            if (Input.GetKey(KeyCode.LeftShift)) // Check if the player is holding down the left shift key
            {
                ChangeCameraFov(cameraTurboFov); // Change the camera's field of view to the turbo mode value
            }
            else // If the player is not holding down the left shift key
            {
                ChangeCameraFov(cameraDefaultFov); // Change the camera's field of view to the default value
            }
        }
    }

    // Changes the field of view of the camera
    public void ChangeCameraFov(float _fov)
    {
        float _deltatime = Time.deltaTime * 100f; // Get the delta time multiplied by 100
        freeLook.m_Lens.FieldOfView = Mathf.Lerp(freeLook.m_Lens.FieldOfView, _fov, 0.05f * _deltatime); // Interpolates the f
    }
    #endregion
  }
```

```
using UnityEngine;

public class AircraftCollider : MonoBehaviour
{
    public bool collideSometing; // Flag to indicate if the aircraft has collided with something

    // This method is called when another collider enters this collider's trigger
    private void OnTriggerEnter(Collider other)
    {
        // If the other collider doesn't belong to the same AircraftCollider component, then it means the aircraft has collide
        if (other.gameObject.GetComponent<AircraftCollider>() == null)
        {
            collideSometing = true; // Set the flag to true
        }
    }
}
```

```
using UnityEngine; // This namespace contains all the core functionality for Unity, including components, game objects, and tr
using System.Collections.Generic; // This namespace contains commonly used data structures such as lists, dictionaries and que

[RequireComponent(typeof(Rigidbody))]
public class AircraftController : MonoBehaviour
{
    #region Variables that are protected
    // List of all colliders of the aircraft
    protected List<AircraftCollider> airCraftColldiers = new List<AircraftCollider>();

    // Maximum speed of the aircraft
    protected float maxSpeed = 0.6f;

    // Current speed of the aircraft for yaw, pitch, and roll
    protected float currentYawSpeed;
    protected float currentPitchSpeed;
    protected float currentRollSpeed;
    protected float currentSpeed;

    // Current intensity and pitch of the engine sound
    protected float currentEngineLightIntensity;
    protected float currentEngineSoundPitch;

    // Boolean to check if the plane is dead
    protected bool planeIsDead;

    // Rigidbody component of the aircraft
    protected Rigidbody rb;
    #endregion

    #region Variables to be assigned
    // Wing trail effects that will be assigned in the inspector
    [SerializeField] protected TrailRenderer[] wingTrailEffects;
```

```
// Engine sound source that will be assigned in the inspector
[SerializeField] protected AudioSource engineSoundSource;

// Array of propellers that will be assigned in the inspector
[SerializeField] protected GameObject[] propellers;

// Array of turbine lights that will be assigned in the inspector
[SerializeField] protected Light[] turbineLights;

// Root of the crash colliders that will be assigned in the inspector
[SerializeField] protected Transform crashCollidersRoot;

// Root of the camera that will be assigned in the inspector
[SerializeField] Transform cameraRoot;

// Aircraft core component that will be assigned in the inspector
[SerializeField] protected AircraftCore myAircraftCore;
#endregion

#region Variables that are accessed
// Getter for the camera root transform
public Transform CameraRoot { get { return cameraRoot; } }

public AircraftCore MyAicraftCore { get { return myAircraftCore; } }
#endregion

#region Audio
// Update engine sound volume and pitch
protected void AudioSystem()
{
    // Smoothly change engine pitch
    engineSoundSource.pitch = Mathf.Lerp(engineSoundSource.pitch, currentEngineSoundPitch, 10f * Time.deltaTime);

    // Fade out engine sound when plane is dead
    if (planeIsDead)
    {
        engineSoundSource.volume = Mathf.Lerp(engineSoundSource.volume, 0f, 0.1f);
    }
}
#endregion

#region Private methods
//List of colliders for the plane
protected List<Collider> crashColliders;

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

    //Loop through colliders and add components to them
    for (int i = 0; i < colliders.Length; i++)
    {
        //Change collider to trigger
        colliders[i].isTrigger = true;

        //Get current gameobject
        GameObject _currentObject = colliders[i].gameObject;

        //Add airplane collider to the current object and add it to the list
        AircraftCollider _airplaneCollider = _currentObject.GetComponent<AircraftCollider>();
        airCraftColldiers.Add(_airplaneCollider);

        //Add rigidbody to the current object
        Rigidbody _rb = _currentObject.GetComponent<Rigidbody>();
        _rb.useGravity = false;
        _rb.isKinematic = true;
        _rb.collisionDetectionMode = CollisionDetectionMode.ContinuousSpeculative;
    }

    //Get all colliders from the crash collider root transform
    crashColliders = new List<Collider>(crashCollidersRoot.GetComponentsInChildren<Collider>());
}

//Rotate the propellers
protected void RotatePropellers(GameObject[] _rotateThese)
{
    //Calculate the propel speed
    float _propelSpeed = currentSpeed * myAircraftCore.PropelSpeedMultiplier;
```

```
        //Loop through the gameobjects to rotate and rotate them
        for (int i = 0; i < _rotateThese.Length; i++)
        {
            _rotateThese[i].transform.Rotate(Vector3.forward * -_propelSpeed * Time.deltaTime);
        }
    }

    //Control the engine lights
    protected void ControlEngineLights(Light[] _lights, float _intensity)
    {
        //Calculate the propel speed
        float _propelSpeed = currentSpeed * myAircraftCore.PropelSpeedMultiplier;

        //Loop through the lights and control their intensity
        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);
            }
        }
    }

    //Change the wing trail effect thickness
    protected void ChangeWingTrailEffectThickness(float _thickness)
    {
        //Loop through the wing trail effects and change their thickness
        for (int i = 0; i < wingTrailEffects.Length; i++)
        {
            wingTrailEffects[i].startWidth = Mathf.Lerp(wingTrailEffects[i].startWidth, _thickness, Time.deltaTime * 10f);
        }
    }

    //Check if the plane has hit something
    protected bool HitSometing()
    {
        //Loop through the airplane colliders and check if any of them collide with something
        for (int i = 0; i < airCraftColldiers.Count; i++)
        {
            if (airCraftColldiers[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 < airCraftColldiers.Count; i++)
        {
            airCraftColldiers[i].GetComponent<Collider>().isTrigger = false;
            Destroy(airCraftColldiers[i].GetComponent<Rigidbody>());
        }

        //Kill player
        planeIsDead = true;

        //What happens next?
        var controller = GameObject.FindGameObjectWithTag("GameController");
        if (controller)
        {
            controller.GetComponent<MasterController>().TerminationRun();
        }

        print("Crash");
    }
    #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 / myAircraftCore.TurboSpeed;

        return _percentToMax;
    }

    public bool PlaneIsDead()
    {
        return planeIsDead;
    }

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

        return false;
    }

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

▼ Player

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

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

    public void CoreUpdate(AircraftCore newAircraftCore)
    {
        myAircraftCore = newAircraftCore;
    }
    #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 += myAircraftCore.Accelerating * Time.deltaTime;
        }
        else
        {
            currentSpeed -= myAircraftCore.Deaccelerating * Time.deltaTime;
        }

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

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

            //Engine lights
            currentEngineLightIntensity = myAircraftCore.TurbineLightTurbo;

            //Effects
            ChangeWingTrailEffectThickness(myAircraftCore.TrailThickness);

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

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

            //Engine lights
            currentEngineLightIntensity = myAircraftCore.TurbineLightDefault;
            //Effects
            ChangeWingTrailEffectThickness(0f);

            //Audio
            currentEngineSoundPitch = myAircraftCore.DefaultSoundPitch;
        }

    }
    #endregion


}
```

▼ Cores

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

[CreateAssetMenu(fileName = "Situation", menuName = "ScriptableObjects/Situation", order = 2)]
public class SituationCore : 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
}
```

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

[CreateAssetMenu(fileName = "EnemyName", menuName = "ScriptableObjects/Enemy", order = 1)]
public class EnemyCore : ScriptableObject
{
    public string enemyID { get; set; }

    #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 { strategyTacticalAwarene
    public MasterAttribute StrategyPositionalAwareness { get { return strategyPositionalAwareness; } set { strategyPositionalA
    public MasterAttribute StrategyStrategemAgility { get { return strategyStrategemAgility; } set { strategyStrategemAgility
    public MasterAttribute StrategyCalculativeAgility { get { return strategyCalculativeAgility; } set { strategyCalculativeAg
    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
    public MasterAttribute PersonalityObedience { get { return personalityObedience; } set { personalityObedience = value; } }
    public MasterAttribute PersonalityImagination { get { return personalityImagination; } set { personalityImagination = valu
    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 { behaviorManeuverPrefer
    public MasterAttribute BehaviorSpeedPreference { get { return behaviorSpeedPreference; } set { behaviorSpeedPreference = v
    public MasterAttribute BehaviorAltitudePreference { get { return behaviorAltitudePreference; } set { behaviorAltitudePrefe
    #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;
        }
    }

    //While Reading Core
    public override string ToString()
    {
        return $"({StatBase}, {StatCons}, {StatMarg}";
    }
}
#endregion
```

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

[CreateAssetMenu(fileName = "Aircraft", menuName = "ScriptableObjects/Aircraft", order = 0)]
public class AircraftCore : ScriptableObject
{
    #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

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

    #endregion

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

    #endregion
}
```

▼ Enemy

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

public class EnemyBrain : MonoBehaviour
{
    #region Base Data
    private EnemyCore eBase;
    private SituationCore sBase;
    private AircraftCore aBase;

    public void Init(SituationCore sb, EnemyCore eb, AircraftCore ab)
    {
        eBase = eb;
        sBase = sb;
        aBase = ab;

        FixingBaseData();
        GenerateWayPoints();

    }

    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;
        var zin = eBase.BehaviorAltitudePreference.StatBase;

        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,
            }
            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 /
            }
            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
}
```

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

[RequireComponent(typeof(EnemyBrain))]
public class EnemyController : AircraftController
{
    #region Base
    private bool isReady = false;

    private void Setup()
    {
        //Setup speeds
        maxSpeed = myAircraftCore.DefaultSpeed;
        currentSpeed = myAircraftCore.DefaultSpeed;

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

        SetupColliders(crashCollidersRoot);
    }
    public void Init(AircraftCore mAC, SituationCore mSC, EnemyCore mEC)
    {
        myAircraftCore = mAC;
        mySituationCore = mSC;
        myEnemyCore = mEC;


        if (myEnemyCore == null || myEnemyCore.name == "Empty")
        {
            int coreSequenceOrder = CoreSequence.CreateRandomEnemyCore();
            myEnemyCore = CoreSequence.randomEnemyCores[coreSequenceOrder];
        }

        Setup();
        isReady = true;
        StartMovement();
    }

    private bool showMe = true;
    private void Update()
    {
        //AudioSystem();


        //Airplane move only if not dead
        if (!planeIsDead && isReady)
        {
            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();
        }
    }

    public void CoreUpdateCopy(EnemyController copyWhat)
    {
        CoreUpdate(copyWhat.myEnemyCore);
        CoreUpdate(copyWhat.mySituationCore);
        CoreUpdate(copyWhat.myAircraftCore);

    }

    public void CoreUpdate(AircraftCore newAircraftCore)
    {
        myAircraftCore = newAircraftCore;
    }

    public void CoreUpdate(EnemyCore newEnemyCore)
    {
        myEnemyCore = newEnemyCore;
    }

    public void CoreUpdate(SituationCore newSituationCore)
    {
        mySituationCore = newSituationCore;
    }
    #endregion

    #region Movement
    [SerializeField] SituationCore mySituationCore;
    [SerializeField] EnemyCore myEnemyCore;
    public SituationCore MySituationCore { get { return mySituationCore; } }
    public EnemyCore MyEnemyCore { get { return myEnemyCore; } }

    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(mySituationCore, myEnemyCore, myAircraftCore);
    }

    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 += myAircraftCore.Accelerating * Time.deltaTime;
        }
        else
        {
            currentSpeed -= myAircraftCore.Deaccelerating * Time.deltaTime;
        }

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

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

            //Engine lights
            currentEngineLightIntensity = myAircraftCore.TurbineLightTurbo;

            //Effects
            ChangeWingTrailEffectThickness(myAircraftCore.TrailThickness);

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

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

            //Engine lights
            currentEngineLightIntensity = myAircraftCore.TurbineLightDefault;

            //Effects
            ChangeWingTrailEffectThickness(0f);

            //Audio
            currentEngineSoundPitch = myAircraftCore.DefaultSoundPitch;
        }
    }
    #endregion

}
```