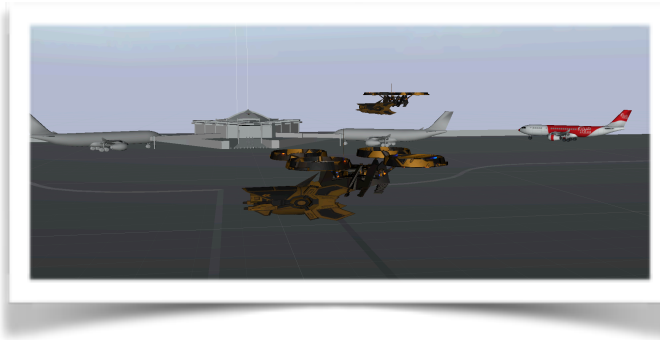
@GREGOIRE COUSIN : 18204188 : GREGOIRE.COUSIN@UCDCONNECT.IE

UNITY GAME DESTROYER

@INTRODUCTION

For this assignment, I was tasked with building a 3D game complete with a variety of complex requirements. The requirements were both designed to test the ability to solve problems successfully, and prove to have a solid understanding in relation to managing a 3-Dimensional Game through the use of Unity. Building an entire game, quite clearly, requires a lot of time, dedication and the ability to manage objects on a 3D spectrum to then manipulate their variables in the desired sequence of events. The game's storyline is based on a drone controlled by a user, who is in charge of protecting an airport from two other drones who are roaming around the landing strip in which an airplane takes off. The enemies are designed to chase you when you are within close proximity, and in turn, shoot at you until you inevitably crash and lose all of your rotors. While you are fighting the enemy drones, you are given the ability to repair yourself at a station in order to augment your health if required.

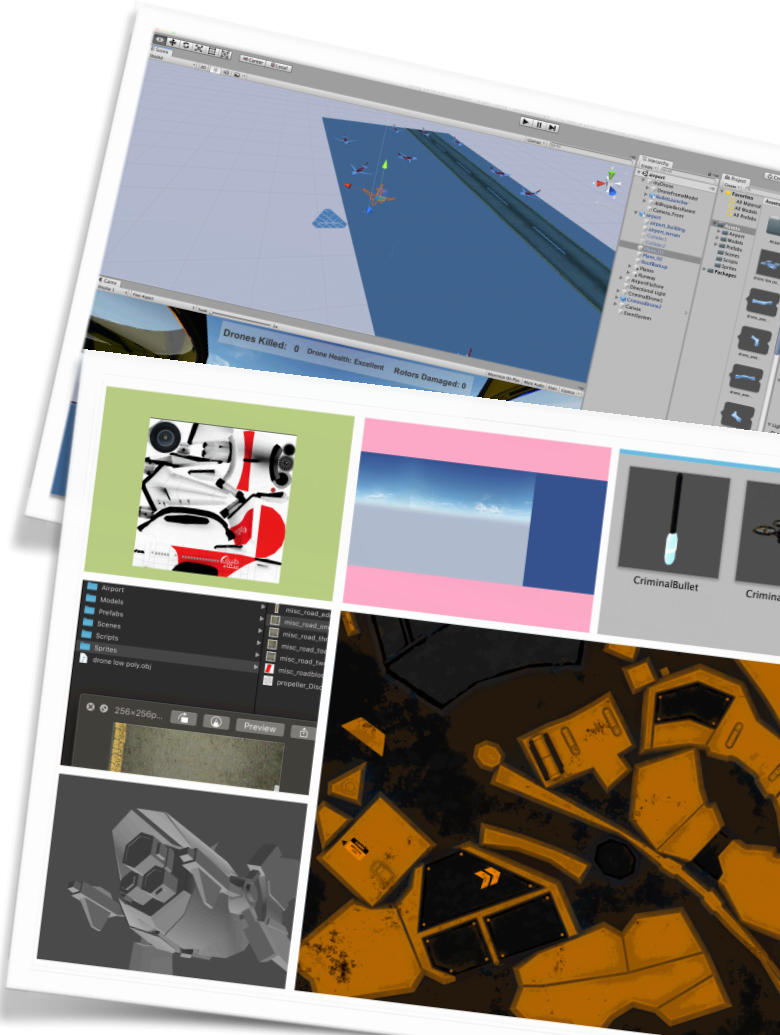
@Design



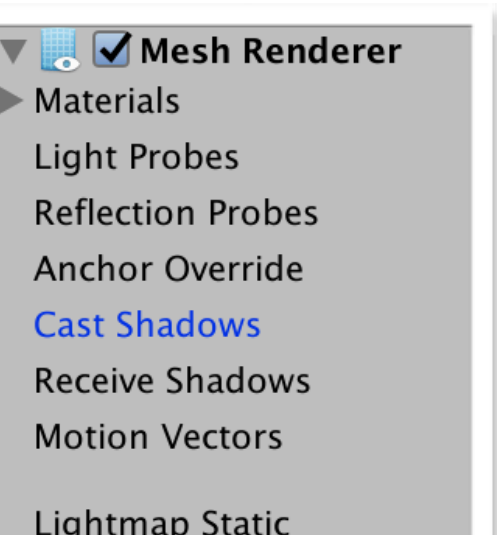
ASSETS

The game has been composed with the use of a series of meshes, each complete with their proper skins and attributes. The material attributes consist of features such as preset lighting and texture materials in order to give the game an increased atmosphere of realism. Game Destroyer encompasses the following Assets, which can be found throughout the map:

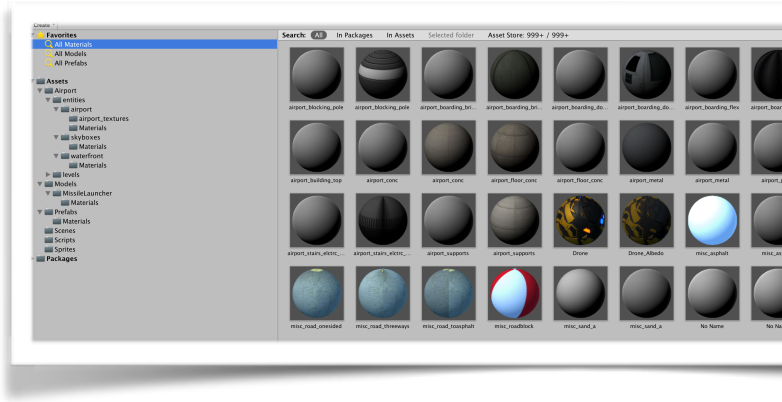
- *Main Character Drone : Hero Character*
- *Two Criminal Drones : Antagonist Characters*
- *Duplicated stationary Airplanes scattered across The Map: Scenery*
- *Airport Homebase : Scenery*
- *Repair Station : Fix Drone upon Damaged Rotors*
- *Flight Lane: Where the Plane Takes Off at Random Intervals*
- *Moving Airplane: Flies off Into the Sky and Comes Back*



By using the Unity Store, I was able to find Sprites free of charge. Each Sprite encompassed in the Mesh are attached together, given proper attributes and are rendered through Unity's Mesh Renderer. The Sprites came with a series of preset materials attached to them, and further, each section of these specific Sprites was given a proper and fleshed out texture.



The airport consists of both bright lighting and metallic surfaces, whilst the airplane's landing strip fits with complex materials such as graduated flooring. The game is interactive, and therefore, will allow the player to be readily aware of their health status. Moreover, the gameplay is constricted to an Airport's Fly zone expertly, complete with specific boundaries that can only filter the airborne plane outside of its boundaries.



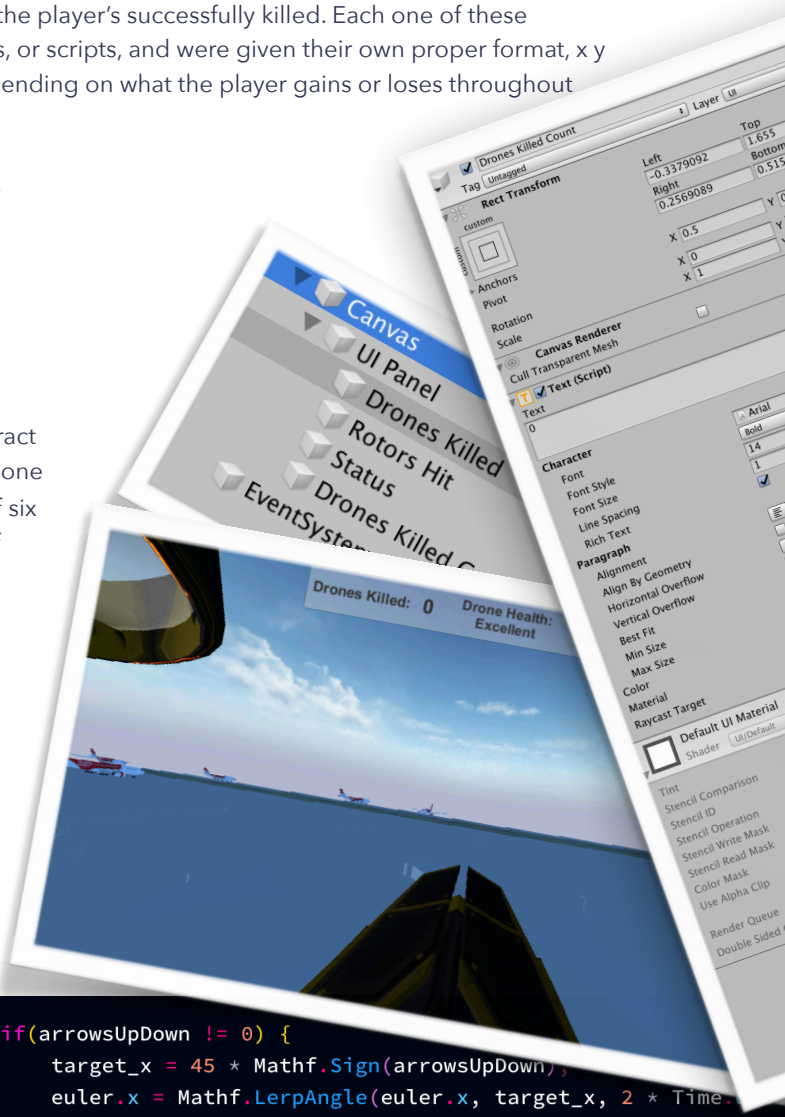
VIEW - UI PANEL

The general view encompassed within our game is designed around the first player's position in relation to two of the rotors which are visible on both side of the canvas, along with the bullet launcher's top section which is viewable at the bottom of the screen. As such, this gives the user a first-person perspective, and allows them to feel as though they're piloting the drone as though they were in the front seat. The top of the screen was designed in a way in which the user can easily keep track of important in-game factors such as remaining health, the damage status of the rotors, how many enemy drones remain and, similarly, the number of enemy drones that the player's successfully killed. Each one of these aforementioned attributes were made with the help of Unity's Text Tools, or scripts, and were given their own proper format, x y and z positions, and were implemented within the script to change depending on what the player gains or loses throughout their session.

The placement of a players position on game start is set in the transform component of the drone.

@Scripts

It was necessary to ensure that some components within the game interact with each other seamlessly throughout the duration of gameplay; each one based on their proper given scripts. In Game Destroyer, I have a total of six specific scripts with each containing different behaviors for each part of the game.



MY DRONE CONTROLLER

Drone Movement: At first, the drone is able to be moved by checking if the specific keys, which are both up and down, have not been triggered yet. After this, it then calculates how far off the specific player moves while the key is being pressed. Once the script has accurately checked as to whether the player has pressed the key, it will then handle the acceleration and, in the same vein, dampen the deceleration of the drone; in other words, the player can determine the velocity and gravity. This can then be set on the main inspector.

```
if(arrowsUpDown != 0) {
    target_x = 45 * Mathf.Sign(arrowsUpDown);
    euler.x = Mathf.LerpAngle(euler.x, target_x, 2 * Time.
    _transform.eulerAngles = euler;
}

// Handle acceleration
if(upDown != 0 && Mathf.Abs(vel_x) < 50 && Mathf.Abs(vel_y) <
    _rigidbody.AddForce(_transform.forward * upDown * accelera
```

To both check and update the position of the drone in an accurate manner, I was made to set specific variables that are tasked with gathering data that is present within the game. Then updating these coordinates correctly in a loop that checks if the drone has not ye crashed. The coordinates include the x and y axis of the drone, along with their unique velocity and Euler Angles.

```
while(!isCrashed) {

    float upDown = Input.GetAxis("Vertical"); //  w/s controls
    float arrowsUpDown = Input.GetAxis("ArrowsVertical"); //up arrow/down arrow tilt
    float leftRight = Input.GetAxis("Horizontal"); // a/d controls
    float vel_x = _rigidbody.velocity.x;
    float vel_y = _rigidbody.velocity.y;
    vel_z = _rigidbody.velocity.z;
    Vector3 euler = _transform.eulerAngles;
    float target_z = 0, target_y = 0, target_x = 0;
```

Furthermore, the drone was also given explicit abilities that work to allow it to stabilize itself if it was not in motion. This was successfully implemented within the game by thoroughly checking whether or not the up and down keys are pressed, and in turn, sending the divided position by half to the AngleAxis method to ensure that it stays in the air but also has the capability to move.

```
if(leftRight == 0 && arrowsUpDown == 0) {
    Vector3 predictedUp = Quaternion.AngleAxis(
        _rigidbody.angularVelocity.magnitude * Mathf.Rad2Deg * stability / stability_speed,
        _rigidbody.angularVelocity) * _transform.up;

    Vector3 torqueVector = Vector3.Cross(predictedUp, Vector3.up);
    _rigidbody.AddTorque(torqueVector * stability_speed * stability_speed);
}

yield return new WaitForSeconds(.01f);
```

The *Drone Controller Script* holds the methods to determine if a drone collides with an object such as an airplane. The Collision method first attempts to generate the instance of the outer-most parent which collided with the drone, and then checks as to whether the object has not collided with it previously to then avoid jitter, along with similar issues, which can result in the potential freezing of the game. After this, it checks whether our playable drone has collided with the tagged Criminal Bullet, to ascertain whether it could have been hit by a criminal drone.Once both of these variables are determined to be true, I can then alter the variables that have been crashed into to true, and perform the crash behavior coupled with the desired gravity that's needed. Therefore, I provide the transformed variable with the string entitled "crashed" within the crash behavior script, and in turn, alter the main "UI views" display to let the player know that they have indeed crashed. What's more, the script also encompasses many other paramount initializations which work to assist the drone in moving towards a desired location. When capturing the gameObject.GetComponent, I am able to gather the game's physics in relation to a particular object, to then change it. In this script, this is used to move one's velocity and force whilst moving in relation to the surrounding gravity

```
private void OnCollisionEnter(Collision collision) {

    // get id of the root object(outer most parent) collided with drone
    int other_id = collision.transform.root.GetInstanceID();

    // to avoid multiple detection of same object collision
    if(LastHitGameObject != other_id && isCrashed == false) {
        LastHitGameObject = other_id;

        if(collision.transform.root.CompareTag("CriminalBullet")) {
            // do following if drone is hit by criminal's bullet
            RotorsDamaged += 1;
            RotorsDamagedUI.text = "Rotors Damaged: " + RotorsDamaged.ToString();
            DroneStatusUI.text = "Drone Health: " + HealthStatus[(int)(RotorsDamaged

            // crashed, if all 12 rotors are damaged
            if(RotorsDamaged >= 12) {
                isCrashed = true;
                _rigidbody.useGravity = true;
                _transform.name += "_crashed";
                DroneStatusUI.text = "Drone Crashed  Game Over! ";
            }
        }
    }
```

COOL PLUS: Along with capturing the drone's behavior including movement, collision and crashes, I also need to capture the behavior of each of the rotors; The capturing of the rotors are placed in a loop within the script in order for the player to be able to rotate each of the rotor's wings whilst simultaneously being in motion.

```
// make a list for 4 propellers discs
List<Transform> discs_transform = new List<Transform> ();
int total_children = _propellers.transform.childCount;
for(int i=0; i < total_children/2; i++) {
    discs_transform.Add(_propellers.transform.GetChild(i)); //discs
}
```

- Not having an Axis-Alligned Launcher, which moves in accordance to the angles of the player's specific positions relating to the keys used.

LAUNCH CONTROLLER

The Launch Controller consists of a simple algorithm that handles the tilting and rotation of the launch controller on the bottom of your view.

First capturing the Arrow Horizontal Input, Initializing targets and getting the local Euler Angle which is relative to the parent transform's rotation. This method is responsible for accurately adjusting the launcher's controls, and furthermore, begins by checking if any other keys have been pressed. Then, it sets the targets of both the left and right deviation limit to 30 degrees. After this, the script then captures the direction that I desire based on Unity' SIGN Scripting method.

```
//handle tilting and rotation
if(leftRight != 0) {
    //set the target left/right deviation limit to 30 degrees.
    //sign provides the direction based on the key pressed.
    //if left is pressed, value < 0; if right is pressed, value > 0, if no
    target_y = 30 * Mathf.Sign(leftRight);
    euler.y = Mathf.LerpAngle(euler.y, target_y, 2 * Time.deltaTime);

    //set the target up/down tilt  limit to 45 degrees. sign provides the
    target_z = -45 * Mathf.Sign(leftRight);
    euler.z = Mathf.LerpAngle(euler.z, target_z, 2 * Time.deltaTime);

} else {
    //this code gradually resets the rotation as target is now 0 degrees
    euler.z = Mathf.LerpAngle(euler.z, 0, 1 * Time.deltaTime);
    euler.y = Mathf.LerpAngle(euler.y, 0, 1 * Time.deltaTime);
}
_transform.localEulerAngles = euler;
yield return new WaitForSeconds(.01f);
```

This calculates whether or not the left key has been pressed, and what the directional value must be, and whether it needs to be less than zero. If it is 'right', I determined that it needs to be greater, thus charging the direction. If it is 0, the value must stay at 0.

After repeating this process several times in relation to the up and down view of the game, sending the Euler back to the 'transform' is crucial. I accomplish this in the following way: transform.localEulerAngles = euler

CRIMINAL DRONE CONTROLLER

This script begins with the main objects and declarations in mind. I imported the required GameObjects such as the propellers, the playable drones, Boolean values that switch if an enemy drone is required to chase the player, the overall speed and the amount of times it has crashed along with the number of times that an enemy has been hit.

```
while(true) {
    // if criminal drone is not crashed and player drone is not crashed
    if(!isCrashed && !PlayerDrone.name.Contains("crashed")) {
        float chase_distance_x = Mathf.Abs(PlayerDrone.position.x - _transform.position.x);
        float chase_distance_z = Mathf.Abs(PlayerDrone.position.z - _transform.position.z);



        // CHASE RADIUS DEFINES THE DETECTION range where player drone is detected by criminal drone
        // -530 and -300, are the upper and lower bounds to make drone only follow player drone near runway.
        if(chase_distance_x <= chase_radius && _transform.position.x >= -530f && _transform.position.x <= -300) {
            //rotate to look at the player

            Vector3 target_look_position = PlayerDrone.position + PlayerDrone.forward * Mathf.Abs(PlayerDrone_rigidbody.velocity.z) * Mathf.Sign(Input.GetAxis("Vertical")) * 0.1f;
            _transform.LookAt(target_look_position);

            // follow the player drone position
            Vector3 target_position = PlayerDrone.position + PlayerDrone.forward * 2f * distanceFromPlayerDrone + side * PlayerDrone.right * distanceFromPlayerDrone;

            // if criminal drone is too far, it will chase with 2x speed to catchup with the player drone
            if(chase_distance_z < 30f)
                _transform.position = Vector3.Lerp(_transform.position, target_position, chaseSpeed * 2 * Time.deltaTime);
            else
                _transform.position = Vector3.Lerp(_transform.position, target_position, chaseSpeed * Time.deltaTime);

        } else {
            // stabilize the criminal drone
            Vector3 predictedUp = Quaternion.AngleAxis(
                _rigidbody.angularVelocity.magnitude * Mathf.Rad2Deg * stability / stability_speed,
                _rigidbody.angularVelocity) * _transform.up;

            Vector3 torqueVector = Vector3.Cross(predictedUp, Vector3.up);
            _rigidbody.AddTorque(torqueVector * stability_speed * stability_speed);

            //go back to original position
            _transform.position = Vector3.MoveTowards(_transform.position, OriginalPosition, Time.deltaTime * 10);
        }
    } else {
        // this sticks the rope to drone rotor and while drone crashes
        if(PlayerBullet != null)
            PlayerBullet.position = _transform.position;
    }
    yield return new WaitForSeconds(.01f);
}
```

Furthermore, the Criminal Drone controller successfully implements the A.I. mechanisms which help to follow the main character, as well as chase him. The A.I. Chase of the Criminal Drone Controller initially captures the position of the player's character, and then minuses its coordinates with its current position. This is needed to work out the relative calculated distance between the two characters.

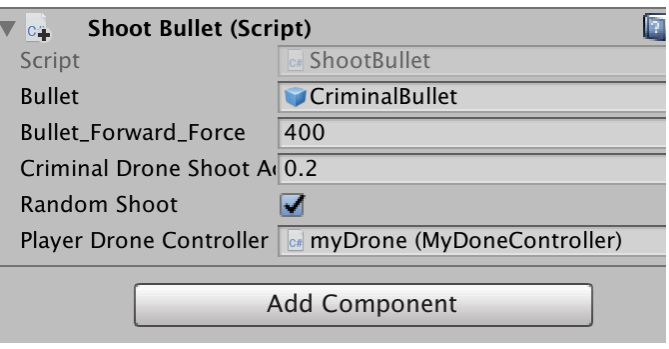
After this, I then check if this distance is less than the "chase-radius" variable. The chase radius is what's needed to define the detection range of the player's drone, when it is detected by a criminal drone. Due to the fact that the specifications of this assignment asked us to ensure that each drone did not cross the runway's boundaries, it was incredibly important to detect where these boundaries were; evidently, -530 and -300 were the upper and lower bounds needed to ensure that an enemy drone only followed the player drone up to the point of this runway.

To further embellish upon the aforementioned point, the condition method works to further determine whether or not the playable character has crashed, and if they have, they will go back to their original position. Using this method, I then proceed to follow the main character's position, simultaneously checking whether or not the chase radius is less than the chase distance. Put simply, if the criminal drone has travelled too far, it will chase the player with two times it's normal speed to successfully catch up with the playable drone. Because these are all constantly moving coordinates for the criminal drone to adhere to, re-stabilizing them back to their original positions was a paramount feature to ensure the game's success. In this script, each rotor is accounted for with a proper mechanism to then spin their axels, giving the drone an added realistic effect throughout the duration of the game.

SHOOT BULLET

The role responsibility of the shoot bullet script is to successfully capture the bullet object, fit each drone with random shooting mechanisms and then calculate a shooting accuracy measurement to ensure that the game is neither too easy or too hard; as such, this manipulates both the force and velocity of the bullets after they've been fired.

The script first uses the "*shootingbullet*" method; after first creating a GameObject Temporary Bullet Handler and then giving it proper moving values with a *rigidbody*, it was important to calculate the relative force which needed to be applied depending on the drone's movement.



Once this bullet is shot, it's destroyed after three seconds. On occasion, bullets might appear to rotate incorrectly due to the way in which its pivot is set in accordance to the original modeling package, and suffice to say, changing position values in relation to gravity can be quite messy. However, This is easily corrected by explicitly giving the right coordinates to follow the mesh's rigid and accurate axis. Since the Enemy Drones needed to be able to shoot at random, I implemented randomized shooting.

After checking both the enemy's parent instance, and whether the player's drone has recently crashed, a random value is given to the input each and every time a user presses the spacebar and then sends this variable to the *shootingbullet* method; this is an accurate way to calculate the force in which a bullet needs to travel to then generate the desired effect. When the keys 'w' or 's' are pressed, a method that checks if *randomshot* has been used is triggered; the relative velocity is then calculated and adjusted as a means of ensuring that the shooting is always balanced. This gives our criminal drone the ability to miss, but still offers the player the challenge that they'd expect from a competitive game.

PLANE TAKE OFF

The Plane Take Off Script makes use of a horizontal acceleration float variable, which will move the airplane along the runway until the variable velocity is smaller than 600. At each cycle, this looped velocity adds +10 to itself whilst also attaching this number to the AddForce method of the plane object's rigid body. Once the velocity reaches 600, the plane will take off with an AddForce; this will both transform the position and acceleration of the plane. Similarly, it will cause AddTorque to rotate the plane so as to ensure that it looks up towards the sky. It is certainly necessary for the plane to have multiple take-offs throughout the duration of a player's session. Thus, both a reset rotation and position were implemented.

LEVEL SELECTOR

Throughout this Script, I made it possible to alter the difficulty of the game based on a criterion made up of two points.

1. *The chaseradius being raised at each game mode, these modes being easy, medium and hard. In this way, the criminal drone cannot catch up to the player at a faster rate.*
2. *The shooting accuracy of each of the criminal drones.*

This means, Upon choosing the level's difficulty, the drones are initially made inactive and both their transform position and rotation are reset. Once they're reset, the chase_radius variable from the CriminalDroneController, as well as the ShootAccuracy variable from the ShootScripts, are changed upon starting the game. Menu Game Object is then deactivated, and the player begins the game.

@Conclusion

The primary objective of this project was to both design and program an A.I. game with the use of unity, and to fully understand the core principles involved in building a three-dimensional game. Throughout the process of conceptualization, design and the eventual programming of the game, I was able to develop a deeper understanding of the core principles which underpin Unity, as well as the differing object-oriented scripting functions that I had never used previously. The most difficult aspect of this project was the act of conceptualizing a fully-functional A.I. drone, and then putting it into motion with the use of various functions to then bring the concept alive. I've learned that taking smaller steps, and organizing certain behaviors within the game into separate files, sections and objects has helped me tremendously. In conclusion, this project has proven to be an extremely important step in my learning exactly how to deal with the complexities involved in game development.