

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

UNITY GAME DESTROYER



@INTRODUCTION

For this assignment, I was given the task of building a 3D game involving a variety of complex requirements. The requirements were designed to both test our ability to solve problems successfully, and to prove we have developed a solid understanding of the creation of features such as gravity, difficulty level, controls etc. in 3D development through Unity. Building an entire game quite clearly requires a lot of time, dedication and an ability to manage objects on a 3D spectrum while subsequently being able to manipulate their variables according to the desired sequence of events. The storyline of the game is based on the recent disruption caused by two enemy drones at an airport. The objective of this assignment is to create a drone which is in charge of protecting the airport runway from two other drones which have the potential to cause serious damage to the aircraft. The enemies are designed to chase you when you are in close proximity to them, and to shoot you until you inevitably crash and lose all of your rotors. While you are fighting the enemy drones, the player should be given the ability to repair the drone at a station in order to augment their health if required. The following sections will explain the design and scripting components of the game before concluding with the challenges faced upon creation of this 3D game.

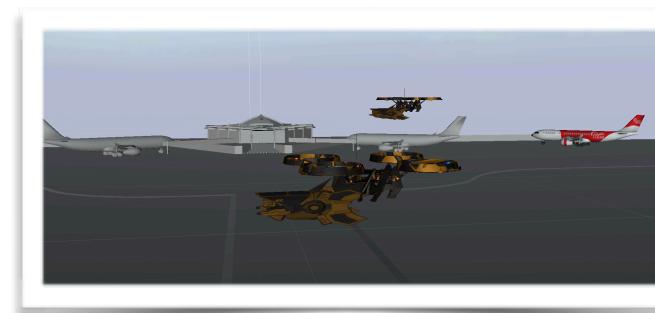
@Design

ASSETS

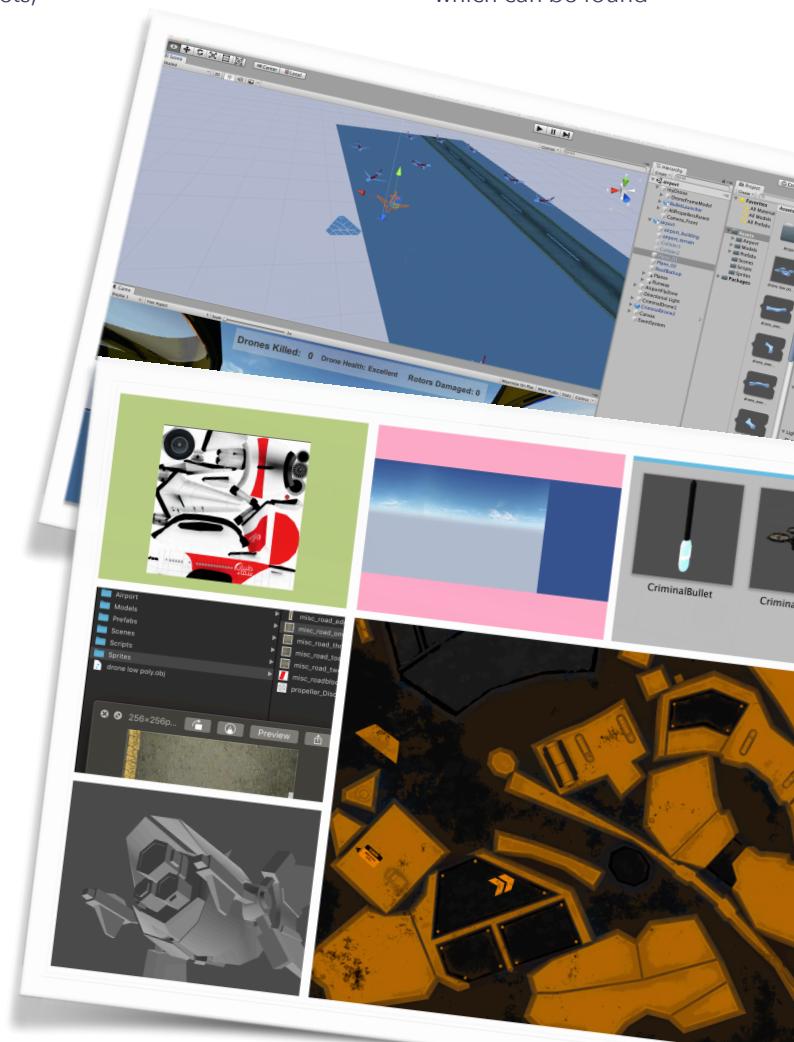
The game has been developed using a series of meshes, each complete with their own skins and attributes. The material attributes consist of features such as preset lighting and texture materials in order to give the game a more realistic atmosphere. Game Destroyer encompasses the following Assets, throughout the map:

- ◆ Main Character Drone : Hero Character
- ◆ Two Criminal Drones : Antagonist Characters
- ◆ Duplicated stationary Airplanes scattered across The Map: Scenery
- ◆ Airport Home-base : 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 and meshes 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.



which can be found

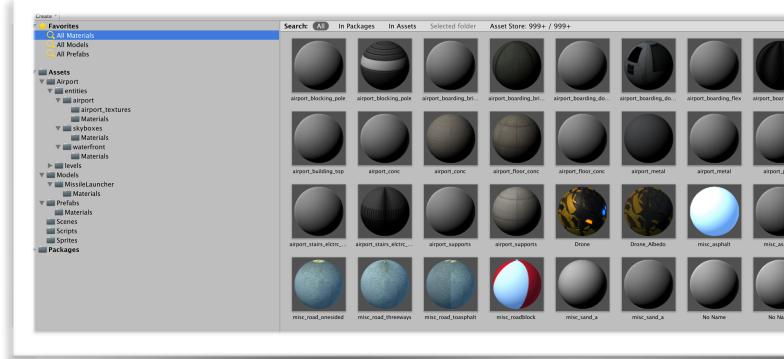


The airport consists of both bright lighting and metallic surfaces, whilst the airplane's landing strip fits with complex materials such as graduated flooring. The game is interactive, and therefore, will allow the player to be readily aware of their health status. Moreover, the gameplay is constricted 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 the 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.

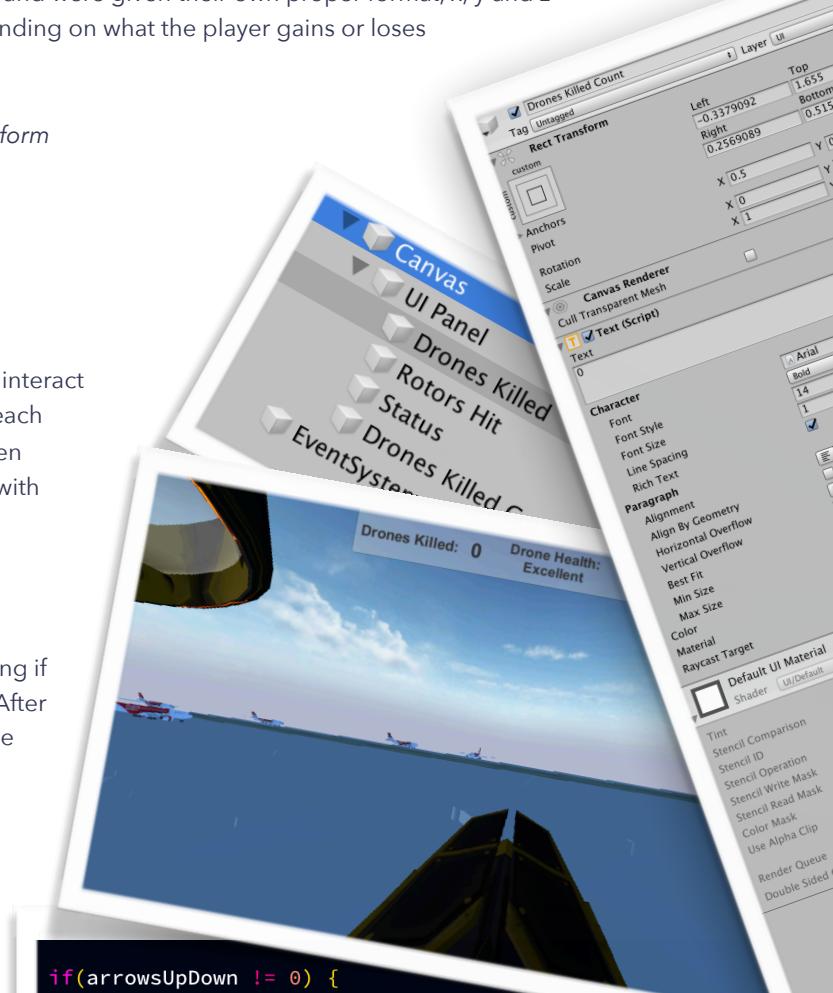


@Scripts

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

MY DRONE CONTROLLER

Drone Movement: At first, the drone is able to be moved by checking if the specific keys, 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.deltaTime);
    _transform.eulerAngles = euler;
}

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

```

PAGE #4 | GAMEDESTROYER : UCD

To both check and update the position of the drone in an accurate manner, specific variables were set with the intention of gathering data within the game. Subsequently, these coordinates are correctly updated within a loop that checks whether or not the drone has crashed yet. The coordinates include the x and y axis of the drone, along with their unique velocity and Euler Angles.

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

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 outermost parent that collided with the drone, and then checks whether the object has not collided with it previously to avoid jittering, along with other similar issues, which can result in the potential freezing of the game (memory overload). After this, it checks whether the 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 (to true) the variables that have been crashed into, 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 View" 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<Rigidbody>` I am able to gather the game's objects physics in relation to another particular object, to then change it. In this script, this is used to move one's velocity, force whilst moving in relation to the surrounding gravity. Where as with `gameObject.GetComponent<Transform>` I am gathering the position of the object.

COOL PLUS: Along with capturing the drone's behavior such as movement, collision and crashes, I also needed to capture the behavior of each of the rotors; The capturing of the rotors is 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.

```
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;
    float vel_z = _rigidbody.velocity.z;
    Vector3 euler = _transform.eulerAngles;
    float target_z = 0, target_y = 0, target_x = 0;
```

```
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);
```

```
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
            RotorDamaged += 1;
            RotorDamagedUI.text = "Rotors Damaged: " + RotorDamaged.ToString();
            DroneStatusUI.text = "Drone Health: " + HealthStatus[(int)(RotorDamaged % 12)];
        }

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

```
// 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-Aligned 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 at the bottom of the players 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's SIGN() Scripting method.

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 changing 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 accomplished 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.

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 and therefore sets the distances between the drones.

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. Given that the assignment specification asked us to ensure that each drone did not cross the runway's boundaries, it was 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 players drone up to the point of this runway.

```
//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 none, value is 0.
    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 direction based on the key pressed.
    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);
//breaks out of the loop when it reaches the target
while(_currentY < target_y || _currentZ < target_z) {
    _currentY +=Mathf.LerpAngle(_currentY, target_y, Time.deltaTime);
    _currentZ +=Mathf.LerpAngle(_currentZ, target_z, Time.deltaTime);
}
```

```
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
        // 330 degrees are the boundaries of runway so to make drone not follow player drone near runway
        if(chase_distance_x < chase_radius && _transform.position.x >=-530 && _transform.position.x <= -300) {
            //rotate to look at the player
            Vector3 targetLookPosition = PlayerDrone.position + PlayerDrone.forward + Mathf.Abs(PlayerDrone_rigidbody.velocity.z) * Mathf.Sign(Input.GetAxis("Vertical")) * 0.1f;
            _transform.LookAt(targetLookPosition);

            // follow the player drone position
            Vector3 targetPosition = 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, targetPosition, chaseSpeed * 2 * Time.deltaTime);
            else
                _transform.position = Vector3.Lerp(_transform.position, targetPosition, chaseSpeed * Time.deltaTime);

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

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

            //move 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);
}

//end of loop after 10 seconds

```

PAGE #6 | GAMEDESTROYER : UCD

To further expand upon the point above, 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 its 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 nor 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 creating a GameObject Temporary Bullet Handler and then giving it proper moving values with a *rigidbody*. It was important

to calculate the relative force that needed to be applied depending on the drone's movement.

This bullet is destroyed three seconds after it is shot. 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. Needless to say, changing position values in relation to gravity can be quite messy, but this is easily corrected by 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 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 would expect from a competitive game.



```

while(true) {
    //this script is applied on launcher child, to check if the criminal drone has crashed, we need to know its parent is
    //we append "crashed" in the gameObject name if it is crashed
    parentCrashed = _transform.root.name.Contains("crashed");
    PlayerCrashed = PlayerDroneController.gameObject.name.Contains("crashed");
    if(!parentCrashed && !PlayerCrashed) {
        //this is to allow player drone to shoot on pressing spacebar
        float multiplier = Random.Range(.5f, 1f);
        if(Input.GetKeyDown(KeyCode.Space) && !randomShot) {
            multiplier = Input.GetAxis("Vertical");
            shootBullet(multiplier);
        }
        //to allow criminal drones to shoot randomly
        else if(randomShot) {
            yield return new WaitForSeconds(multiplier * .5f);
            //if we press w the relative velocity is calculated by adding the velocity of bullet else it is shot with velocity
            float rel_vel = Input.GetAxis("Vertical") == 0.7f ? Mathf.Abs(PlayerDroneController.vel.z) * ShootAccuracy /
            shootBullet(rel_vel);
        }
    }
}

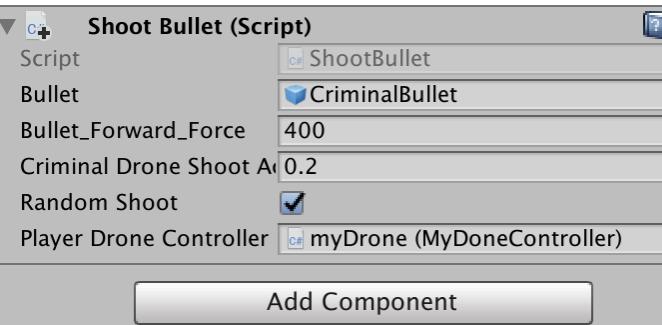
//this is to prevent the bullet from being destroyed immediately after it is created
void shootBullet(float multiplier) {
    //the bullet instantiation happens here.
    GameObject Temporary_Bullet_Handler = Instantiate(Bullet, _transform.position, _transform.rotation)as GameObject;
    Temporary_Bullet_Handler.transform.Rotate(Vector3.left * 90);

    //sometimes bullets may appear rotated incorrectly due to the way its pivot was set from the original modeling package.
    //this is mostly corrected here, you might have to rotate it from a different axis and/or angle based on your particular
    Temporary_Bullet_Handler.transform.Rotate(Vector3.left * 90);

    //retrieve the rigidbody component from the instantiated bullet and control it.
    Rigidbody Temporary_Rigidbody;
    Temporary_Rigidbody = Temporary_Bullet_Handler.GetComponent();
    float rel_force = Bullet_Forward_Force;

    //decide on the relative force to be applied depending on drones movement
    if(multiplier > 0)
        rel_force = Bullet_Forward_Force * 4 * Mathf.Abs(multiplier);
    else if(multiplier < 0)
        rel_force = Bullet_Forward_Force * Mathf.Abs(multiplier) * 0.09f;
    Temporary_Rigidbody.AddForce(_transform.forward * rel_force);
    //destroy in 3 seconds
    Destroy(Temporary_Bullet_Handler, 3f);
}

```



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 less 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 due to the combination of an AddForce and AddTorque method; 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 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 criteria 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 signifies that 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. The *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 gain a new level of understanding of the core techniques required for building a three-dimensional game. Throughout the process of conceptualization, design and eventual programming of the game, I was able to develop a deeper understanding of the core principles and constraints which underpin Unity itself, as well as the differing object-oriented scripting functions that I had not previously used.

The most difficult aspect of this project was the conceptualization of a fully-functional A.I. drone and bringing this concept alive by putting it into motion with the use of various functions. I also found it hard to set the boundaries of the runway so that the criminal drones wouldn't go beyond their boundaries. Additionally, I found the mathematical calculations such as finding the angular accelerations and position changes of certain object particularly challenging. I also found it hard to implement the change in direction and speed of an enemy in relation to a players drone. However, slowing down the criminal drone acceleration in contrast to the players current speed made it an enjoyable experience to detail the chase. Graphics are a crucial aspect to game presentation given their visual nature. I found that both assignment specification and time constraints resulted in the game not possessing a number of visual features I would have liked to add (*grass, lasers, buildings, etc.*). I felt as though this diminished the game's immersive quality significantly but were also understandably unnecessary for the functionality of the game. Given more time, I would be interested in improving the design aspect of the game.

Finally, I have learned that taking smaller steps, and organizing certain behaviors within the game into separate files, sections and objects has helped me tremendously moving forward. In conclusion, this project has proven to be an extremely important step in my learning of how to deal with a number of the complexities involved in game development.

```

public void ResetDrones() {
    //make all criminal drone inactive and reset transform position and rotation
    for(int i = 0; i < 3; i++) {
        CriminalDrones[i].gameObject.SetActive(false);
        CriminalDrones[i] = InitialTransforms[i];
    }
    PlayerDrone.SetActive(true);
    GameStatus.SetActive(true);
}

public void MediumMode() {
    ResetDrones();
    //make first and second criminal drones active and set detection zone to 50
    for(int i = 0; i < 2; i++) {
        CriminalDrones[i].gameObject.SetActive(true);
        CriminalDrones[i].gameObject.GetComponent<CriminalDroneController>().chase_radius = 50f;
    }
    //shoot accuracy range is from 0 to 1 where 1 stands for 80%
    ShootScripts[1].ShootAccuracy = 0.8f;
    //turn off menu screen
    Menu.SetActive(false);
}

```

```

while(true) {
    velocity = 0;
    r = Random.Range(5, 10);
    yield return new WaitForSeconds(r);
    //horizontal acceleration on runway and makes plane run on runway until its velocity is smaller than 600
    while(velocity < 600) {
        _rigidbody.AddForce(_transform.forward * velocity, ForceMode.Acceleration);
        velocity += 10f;
        yield return new WaitForSeconds(.1f);
    }
    //takeoff from runway and fly when velocity is greater than 600
    while(velocity < 900) {
        _rigidbody.AddForce((_transform.forward + _transform.up) * velocity, ForceMode.Acceleration);
        //add rotation to look upwards
        _rigidbody.AddTorque(_transform.right * -300f);
        velocity += 10f;
        yield return new WaitForSeconds(.1f);
    }
    //Dampen momentum of the drone
    _rigidbody.velocity = Vector3.zero;
}

```