Devlog

Character Controller: Feb 20th - Feb 26th

The first thing I had to do was to create a first-person character controller. I need to be able to walk, crouch, sprint, jump, lean and potentially lie prone.

After doing some searching online, I found an example Github Repo that mimicks the Source Engine movement code. This was the code used in CS:2, Half-Life 2 and other Valve games. I also found some short videos that explained this code in very simple terms, and helped me bugfix. I read through the movement code, and translated the parts I needed into my own project. I missed out on things such as ladder movement and swimming, as the map doesnt include either. The camera and player movement happens in the `Player.cs` file.

```
private void CalculateGroundMovement()
{
        //wishDir is direction you wish to move in, or the vector directly resulting from your key presses
    Vector3 wishDir = Vector3.Normalize(inputMovement.y * transform.forward + inputMovement.x * transform.right);

    float wishSpeed = maxSpeed / 100;

    //currentSpeed is not actually the current speed, but the dot product of the current velocity and the direction your arrow keys are pressing.
    //This is what enables air-strafing, allowing the player to bunnyhop.
    float currentSpeed = Vector3.Dot(velocity, wishDir);

    float addSpeed = wishSpeed - currentSpeed;
    float accelSpeed = Mathf.Min(accel * Time.deltaTime * wishSpeed, addSpeed);

    //Add speed in wish direction
    velocity.x += accelSpeed * wishDir.x;
    velocity.z += accelSpeed * wishDir.z;

        //Calculate the deceleration
    float speed = velocity.magnitude;
    float drop = speed * decel * Time.deltaTime;
    float newSpeed = Mathf.Max(speed - drop, 0);
    if (speed > 0) newSpeed /= speed;
    //Apply deceleration
    velocity.x *= newSpeed;
```

```
    velocity.z *= newSpeed;
}
```

This creates a much higher quality character movement controller than a lot of other solutions for Unity, as it has smooth acceleration and deceleration.

```
    private void CalculateView()
    {
            //Delta input for mouse movement
        inputView *= sensitivity / 100;

        cameraAngles += inputView;
        //Clamp the Y axis to +-90 so you can't do "backflips"
        cameraAngles.y = Mathf.Clamp(cameraAngles.y, -90, 90);
                //Rotate the entire player on the Y axis, and only the
camera holder on the X axis.This means the entire player object is always
looking forward, but the player does not glitch out when you look up or down
as vertical rotation is only done on the camera.
        Quaternion camRot = Quaternion.AngleAxis(-cameraAngles.y,
Vector3.right);
        Quaternion playerRot = Quaternion.AngleAxis(cameraAngles.x,
Vector3.up);

        camHolder.localRotation = camRot;
        transform.localRotation = playerRot;
    }
```

Weapon Sway: Feb 27th - Mar 3rd

Once I had completed the movement controller, it was time to work on the weapon controller. I searched online for some tutorials on first person weapons in Unity, and found this very helpful tutorial series.

First, I worked on weapon sway, which is the gun's movement when you look around quickly, are currently walking, etc.

```
void CalculateWeaponRot()
{
        //if you are aiming down sights, scale each component of the sway
down by individual amounts
    float _movementScaler = 1;
    float _swayScaler = 1;
    if (isAiming)
    {
        _movementScaler = movementRotScaler;
        _swayScaler = swayRotScaler;
    }
        //Most weapon sway is done using Vector3.SmoothDamp
```

```
        //This smoothly moves the object to the target position, with a
smoothness set by the user
    weaponRotation.x += GameManager.GM.player.accumulatedInputView.y *
swayAmount;
    weaponRotation.y += -GameManager.GM.player.accumulatedInputView.x *
swayAmount;
    weaponRotation = Vector3.SmoothDamp(weaponRotation, Vector3.zero, ref
weaponRotationVelocity, swaySmoothing);
    newWeaponRotation = Vector3.SmoothDamp(newWeaponRotation,
weaponRotation, ref newWeaponRotationVelocity, swayResetSmoothing);
    newWeaponRotation.z = newWeaponRotation.y * 0.75f;

    movementRotation.z = -movementSwayAmount *
GameManager.GM.player.inputMovement.x;
    movementRotation = Vector3.SmoothDamp(movementRotation, Vector3.zero,
ref movementRotationVelocity, movementSwaySmoothing);
    newMovementRotation = Vector3.SmoothDamp(newMovementRotation,
movementRotation, ref newMovementRotationVelocity, movementSwaySmoothing);
    //Apply the new positions multiplied by the aiming scaler
    wpnRot += newWeaponRotation * _swayScaler + newMovementRotation *
_movementScaler;
}
```

The tutorial series helped me with starting the weapon system, however it had many issues and I had to write quite a lot of my own code. For example, the youtuber used an animation clip for walking, but I wanted a more procedural system, where you can change the step speed and how far it steps side to side. It also allowed me to slightly randomise the step locations to add some realism.

```
void FixedUpdate(){
        //Simple iterator acts as a timer
        if (curWalkLifetime < walkLifetime * walkLifetimeScaler)
        {
            curWalkLifetime += 1;
        }
        else
        {
                //Swap foot
            curWalkLifetime = 0;
            rightFoot = !rightFoot;
        }
}
```

```
    void CalculateWalk()
    {
            //Scale down if aiming down sights
        float _walkScaler = 1;
```

```
        if (isAiming) _walkScaler = walkScaler;

                //if in the second half of the step, the target is the rest
position
        Vector3 target = Vector3.zero;
        if (GameManager.GM.player.velocity.magnitude > 0.01f)
        {
                //if you are in the first half of the step, target is moved
down and towards the foot that is currently stepping
            if (curWalkLifetime < (walkLifetime * walkLifetimeScaler) / 2)
            {
                float lateralVelocity = new
Vector2(GameManager.GM.player.velocity.x,
GameManager.GM.player.velocity.z).magnitude;
                target.y -= Random.Range(stepDownAmount.x, stepDownAmount.y)
* lateralVelocity;
                float sideAmount = Random.Range(stepSideAmount.x,
stepSideAmount.y);
                //Move to left or right depending on boolean set in
FixedUpdate()
                if (rightFoot) target.x += sideAmount * lateralVelocity;
                else target.x -= sideAmount * lateralVelocity;
            }
        }
        //Apply movement towards target
        walkMove = Vector3.SmoothDamp(walkMove, target, ref
walkMoveVelocity, walkMoveSmoothing);
        newWalkMove = Vector3.SmoothDamp(newWalkMove, walkMove, ref
newWalkMoveVelocity, walkMoveSmoothing);

        wpnPos += newWalkMove * _walkScaler;
        wpnRot += new Vector3(newWalkMove.y, newWalkMove.x, -newWalkMove.x *
1.5f) * stepRotScaling;
    }
```

I also wrote a similar method for breathing, with some randomness to increase realism. Once I was done with that, I also needed to allow the player to aim down the sights of their gun. This is where I started to create the actual gun logic (shooting, reloading, recoil, etc.) Each firearm has 2 Vector3s called `restPos` and `aimPos`. These are the 2 positions of the gun at rest and while you are aiming down sights. They are different for each gun, so is stored in the `firearmInfo` class along with stats such as recoil, fire rate, fire modes and magazine size.

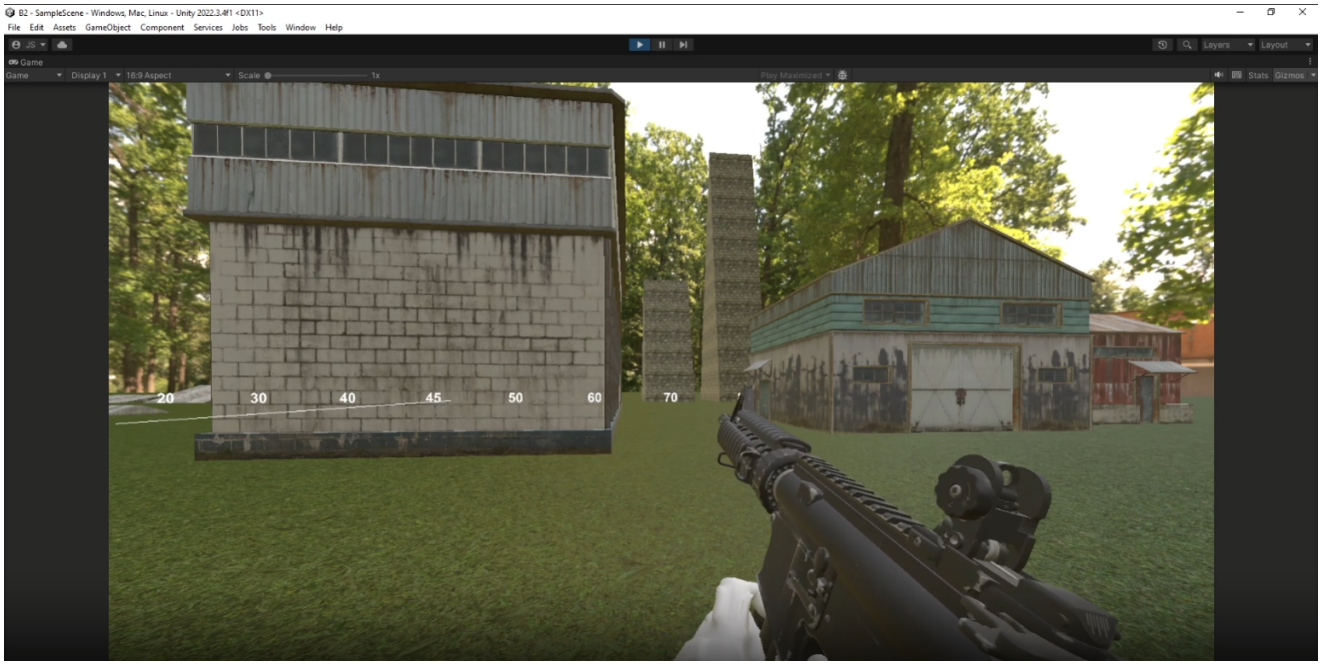
Weapons: Mar 4th - Mar 14th

I then added shooting mechanics.

```
void FixedUpdate(){
        //Single-Fire just calls the Shoot() method as soon as it is pressed
    if (canShoot && fullAutoHeld && curFireMode == FireMode.fullAuto &&
!isReloading && roundsInMag > 0)
    {
        Shoot();
    }
}
 void Shoot()
 {
         //Instantiate round at barrel point
     GameObject roundObj = Instantiate(roundPrefab, barrelPoint.position,
transform.rotation);
     roundObj.GetComponent<Round>().firearmFiredFrom = this;

     AddRecoil();
     canShoot = false;
     roundsInMag -= 1;
     sustainedRecoilAdd += info.sustainedRecoilAdd;

     if (muzzleFlash != null)
     {
         muzzleFlash.transform.localEulerAngles =
new(muzzleFlash.transform.localEulerAngles.x,
muzzleFlash.transform.localEulerAngles.y, UnityEngine.Random.Range(0, 360));
         muzzleFlash.SetActive(true);
         Invoke(nameof(ResetMuzzleFlash), Time.deltaTime * 2.5f);
     }

     if (roundsInMag >= 0)
     {
             //Invoke() calls a method after a set period of time
             //In this circumstance, ResetShot() sets canShoot back to true
after a single rpm interval (converted into seconds from rounds per minute)
         Invoke(nameof(ResetShot), 1/ (info.roundsPerMinute/60));
     }
```

This is the first time I have used the `Invoke()` method, and it is a very useful tool for timing different actions. It can be used to time a full-auto gun to fire at the right RPM, as well as handling reloading. I also added a button (V by default), to change the fire mode from single fire to full-auto. Once all of the shooting was finished, recoil was next. Recoil is done in a very similar way to weapon sway, by adding rotation to the gun using `Vector3.SmoothDamp()`. There are 4 ways that recoil affects you, vertical rotation, horizontal rotation, camera rotation and lateral movement. Each shot's recoil is slightly randomized as

well, to make spray patterns different every time.



Mar 20th

I found 6 weapon models on Sketchfab, and began putting them into blender in order to get them ready for Unity. I had to manually position the hands, and added some very simple reload animations (your character moves the gun off screen then back on screen).

As well as this, I used 2 old shaders from when I created mods for H3VR, a red dot sight and a magnified optic. They were fairly simple to implement and I particularly like the optic shader, as it uses a `RenderTexture`. This is an additional camera that renders to a texture in real-time, and you can use that texture to display in shaders. This means that even though the scope zooms in, your peripheral vision stays at the normal FOV and you do not sacrifice

spatial awareness when scoping in.

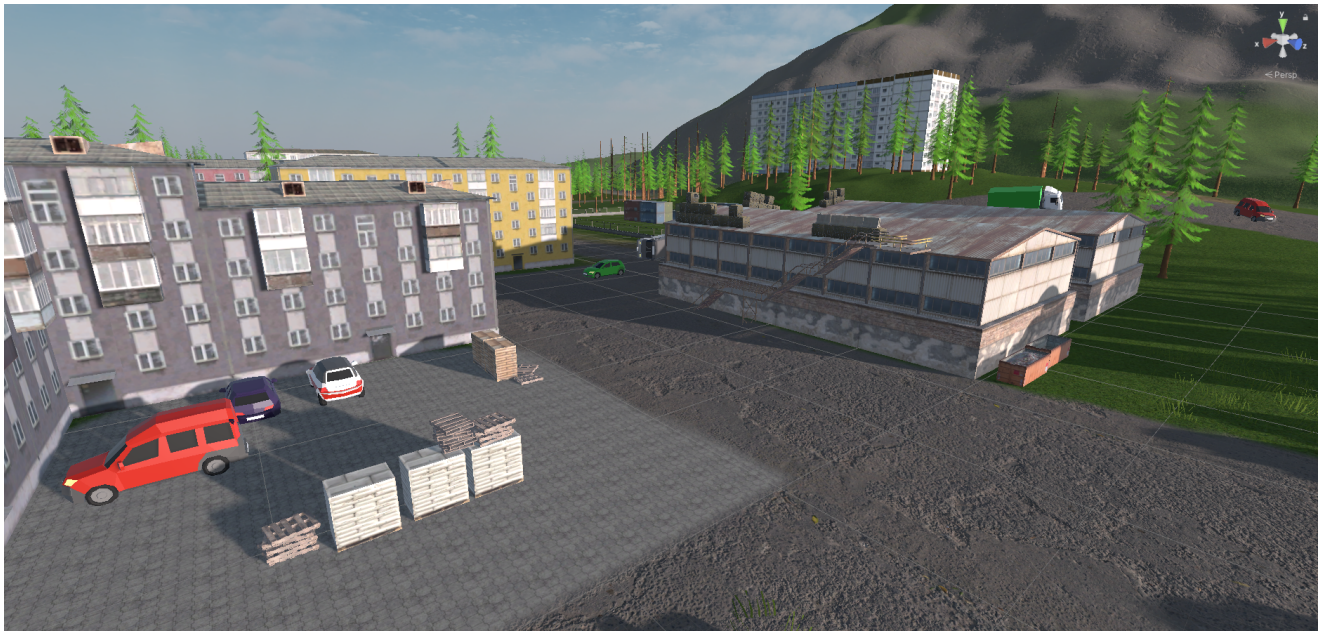
Map Design: Mar 14th - Apr 24th

At this point, I knew that the debug scene I had created in order to test movement needed an update to further iterate on my game concept. I started making a larger scene that would become the first area in the game: "Clifftown". Originally a small town in Sujusterea, it was taken over by military forces and turned into a stronghold defending important intelligence. With a large cliff overlooking the town, long range combat is inevitable, and you'll want to bring an optic.

Here is a very early photo of the map from the players perspective. I used many assets from the Unity Asset Store and the website Sketchfab, all of which were free and even though the

visual style of my game is very kitbashed (meaning there are lots of different parts from different people, and the aesthetics are not consistent), I ended up with a functional and quite large play area for the initial build. I begun creating the map with Unity's Terrain system. This is a plane that you can deform and paint on to create the ground for the world.

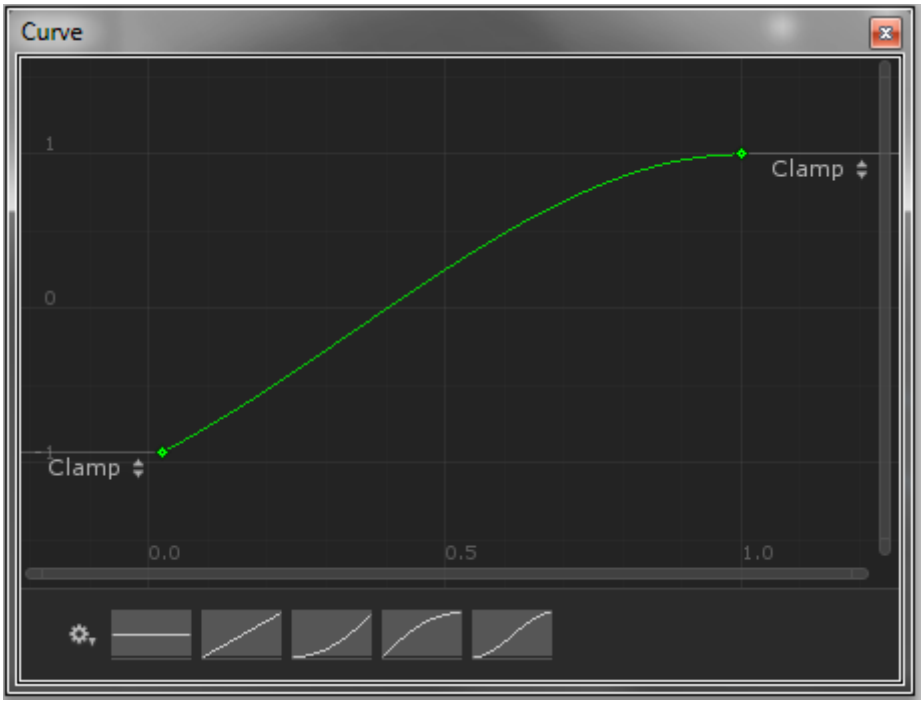
I then added pathways by painting gravel and dirt, added many assets, including buildings, military tents, tanks and watchtowers, as well as cover such as cars and trees. The grass and trees use Unity's terrain system as well, as there is functionality for painting trees and grass onto the ground.



Bullet Physics: Mar 12 - Apr 11th

I knew that I wanted to have realistic bullet physics, including travel time and bullet drop. This actually was one of the hardest features to implement, as I ran into many issues with collision. The way that many games handle bullets is using a raycast. Raycasts are invisible lines that travel at infinite speed that you can send out and get information about the object that it hits. It is used everywhere in games programming, for example checking if there is a wall in-between the Enemy AI and the Player Character. They are a very simple solution for

bullet physics, as they are very easy to implement and have very little performance impact. However, it has a few drawbacks, namely that they have to be in a straight line. This means that the bullet can not have drop-off at range, which is unrealistic. Another drawback is that all raycasts happen instantly, meaning the bullet travels at an infinite speed. This is not realistic and makes the gameplay shallower as you will not need to account for bullet drop and travel time when shooting at long distances. I had to use a new solution that treats the bullet as a physical object, instead of a raycast. Unity detects collision between objects using its collision system, and it checks physics updates every time a method called `FixedUpdate()` is called. This is typically 50 times a second, however it is possible to customise the time step inside Unity's Preferences. My first iteration of the bullet physics used a physical object that updated its position every `FixedUpdate()`. I also used a variable type called an `AnimationCurve`, that lets you put in an input value and receive a value from the curve. I used this to measure the bullet drop, and authored the curves to fit real world values for each caliber.



While testing however, many issues were apparent. Unity's collision system is notoriously bad at handling collisions between very fast moving objects. I have run into this problem before in some other game prototypes, where an object is moving so fast, it just phases through obstacles it should hit. Obviously, this is not good as bullets may go through enemies and walls.

```
    void FixedUpdate()
    {
        distFromOrigin = (transform.position - startPoint).magnitude;
        //Use distance from origin to evaluate animation curve
        float dropAmount = dropCurve.Evaluate(distFromOrigin);

        curPoint += (velocity.z * transform.forward)+(velocity.y *
transform.up);
        curPoint.y += dropAmount/1000;

        float velocityReduction = Mathf.InverseLerp(0, maxDist,
```

```
distFromOrigin);
        velocity.z *= velocityReduction / 100;

                //This should be done in the Start() method, however this is
old deprecated code
        Material newMat = new(tracerMat);
        newMat.color = tracerColor;
        newMat.SetColor("_EmissionColor", tracerColor);

        lineRenderer.material = newMat;


                //destroy for performance if too far away (2000m default)
        if (distFromOrigin > despawnDist) Destroy(gameObject);
        positions.Add(curPoint);
        transform.position = curPoint;
    }
```

Apr 8

As this method had many issues with collision, I went online to find other solutions to my problem. I found a very helpful youtube tutorial that uses a hybrid method.

Every `FixedUpdate()`, a new raycast is sent, going from the current point to the "next point." The next point is calculated in the `PointOnParabola()` method.

```
private void Start()
{

    startPosition = transform.position;
    startDir = transform.forward.normalized;

    lineRenderer.enabled = false;
    Invoke("Show", 0.05f);

        //Set up color and materal of line renderer for tracers
    Material newMat = new(tracerMat);
    newMat.color = tracerColor;
    newMat.SetColor("_EmissionColor", tracerColor);
    lineRenderer.material = newMat;
}
void Show()
{
    lineRenderer.enabled = true;
}

Vector3 PointOnParabola(float time)
{
```

```
        Vector3 pos = startPosition + (muzzleVelocity * time * startDir);
        Vector3 gravityVector = Vector3.down *(gravity * time * time);
        return pos + gravityVector;
    }
    bool RayBetweenPoints(Vector3 startPoint, Vector3 endPoint, out RaycastHit
hit)
    {
            //Do raycast from current point to next point
        return Physics.Raycast(startPoint, endPoint - startPoint, out hit,
(endPoint - startPoint).magnitude);
    }
    private void Update()
    {
        if (startTime < 0) return;

        float currentTime = Time.time - startTime;
        Vector3 currentPoint = PointOnParabola(currentTime);

        transform.position = currentPoint;
    }
    void FixedUpdate()
    {
        distFromOrigin = (transform.position - startPosition).magnitude;

        if (startTime < 0) startTime = Time.time;
        float currentTime = Time.time - startTime;
        //fixedDeltaTime is the amount of time that one FixedUpdate takes up,
so we are calculating what point the currentPoint will be next FixedUpdate
        float nextTime = currentTime + Time.fixedDeltaTime;
            //gets the points from time value
        Vector3 currentPoint = PointOnParabola(currentTime);
        Vector3 nextPoint = PointOnParabola(nextTime);

            //Line Renderer pointing in the right direction
        transform.LookAt(nextPoint);

        RaycastHit hit;
            //This is where all of the collision happens
            //This should be a switch statement for efficiency, however I
havent changed it yet
        if (RayBetweenPoints(currentPoint, nextPoint, out hit))
        {
            if (!hit.collider.CompareTag("Round"))
            {
                if (hit.collider.CompareTag("Ground"))
                {
                    GameObject g = Instantiate(concreteHit, hit.point,
Quaternion.Euler(nextPoint - currentPoint));
                    g.transform.parent = null;
```

```
            }

            if (hit.collider.CompareTag("Enemy"))
            {
                Instantiate(bloodHit, hit.point, Quaternion.Euler(nextPoint
- currentPoint));
                hit.collider.gameObject.GetComponent<Enemy>
().Hit(Random.Range(damage.x, damage.y), hit.point);
            }
            if (hit.collider.CompareTag("EnemyHead"))
            {
                hit.collider.gameObject.GetComponent<Head>().Hit();
            }
            if (hit.collider.CompareTag("Player"))
            {
                Transform player = GameManager.GM.player.transform;
                Vector3 hitPoint = player.position;
                hitPoint.y += 1.75f;
                hitPoint += 0.5f * player.forward;
                Instantiate(bloodHit, hitPoint, Quaternion.Euler(nextPoint
- currentPoint));

                hit.collider.gameObject.GetComponent<Player>
().Hit(Random.Range(damage.x, damage.y));
            }
            if (hit.collider.CompareTag("Target"))
            {
                AudioSource g =
hit.collider.gameObject.GetComponent<AudioSource>();
                g.PlayOneShot(g.clip);
            }
            Destroy(gameObject);

        }
    }
```

This solution for bullet physics works perfectly, and as it is raycast-based, there will be no issues with hit detection as raycasts are much more consistent. The only issue I have is that the bullet drop cannot be tuned to real-world values. This is because the calculation is `gravity * time * time,` and `gravity` is not based on any real world physics.

UI and Infiltration - Mar 21st - Apr 22

Now that I had the player character functional, I decided to create the Infiltration screen, which is where you choose your loadout and the point that you want to enter the map. I used a camera that is rendering the map from above, and some simple buttons on the side of the screen. Right now, the UI is quite unintuitive and many people glazed their eyes over the buttons to select the infiltration point while testing. I think this is because I need larger icons

that will contrast more with the background, as well as some text labels. I could additionally grey out the Infiltration button until both a weapon and an infiltration point is selected.

As well as this, I added some UI to the game including a crosshair, health indicator, ammo indicator and fire mode indicator. The crosshair uses a raycast directly from the barrel and displays the crosshair wherever it hits. This means that when you are moving around, the crosshair will accurately reflect the point where the bullet will hit, and it will move around with weapon sway as well.

AI - Apr 11th - Apr 26th

AI was the feature that I expected would be one of the hardest features to implement, however once I started it was more intuitive than I thought. Unity's Navmesh system is very

high quality, and I was able to bake a navmesh (a mesh with every area that an enemy is able to walk to), and then I am instantly able to give the enemy a point, and it will automatically pathfind to that location with smooth movement. Before I wrote the code to tell the enemies where to go, I decided they should be able to shoot back. To do this, I needed to check if the enemy could see the player, which was as simple as measuring the angle of the raycast between the enemy and the player, and if it was inside the "view cone" bounds, then the enemy could see you.

```
Vector3 playerDir = playerEyePos - eyePos.position;
        canSeePlayer = false;
        if (Physics.Raycast(eyePos.position, playerDir, out RaycastHit hit,
Mathf.Infinity) && hit.collider.CompareTag("Player")) // TODO: less chance
to spot if far away
        {
            if (Vector3.Angle(eyePos.forward, playerDir) < viewCone)
            {
                if (!canSeePlayer) SpotPlayer();
                canSeePlayer = true;
                lastPositionPlayerSeen =
GameManager.GM.player.transform.position;
                //player position is the feet position, so we need to add on
the Y axis to bring it to eye level
                lastPositionPlayerSeen.y += 1.75f;
            }
        }
```

Once that check was in place, the enemy will now rotate to face you if it spots you, then after a randomised "reaction time" (0.5s - 3s depending on distance from player), they will start firing in bursts at you. If they are a long distance away, they will fire in bursts of 1 or 2 shots, however if they are closer they may use full-auto fire. This was because I noticed enemies very far away would be able to shoot extremely accurate full-auto bursts at an unrealistically long range. There is also a short, randomised cooldown between burst shots.

```
void FixedUpdate(){
        if (canSeePlayer)
        {

            lookPos.forward = lastPositionPlayerSeen - lookPos.position;
            lookPos.localEulerAngles = new(0, lookPos.localEulerAngles.y,
0);
            firearmPos.forward = lastPositionPlayerSeen -
firearmPos.position;
            //randomise bullet trajectory for innacuracy
            firearmPos.localEulerAngles += Random.Range(-0.5f, 0.5f) *
transform.up;
            firearmPos.localEulerAngles += Random.Range(-0.5f, 0.5f) *
transform.right;
```

```
            if (timeSincePlayerSeen == 0)
            {
                curReactionTime = Random.Range(reactionTime.x,
reactionTime.y);
                onCooldown = true;
                CooldownShot();
                //roundsLeftInBurst = 1;
                ResetShot();
            }
            else if (!onCooldown && roundsLeftInBurst ≤ 0)
            {
                onCooldown = true;
                Invoke(nameof(CooldownShot), Random.Range(cooldown.x,
cooldown.y));
                curRecoilInaccuracy = 0;
            }
            if (!onCooldown && canShoot && roundsLeftInBurst > 0 &&
timeSincePlayerSeen > curReactionTime)
            {
                Instantiate(bulletPrefab, firearmPos.position,
firearmPos.rotation);

                canShoot = false;
                roundsLeftInBurst -= 1;
                roundsInMag -= 1;
                Invoke(nameof(ResetShot), 0.1f);

                anim.Play("Base Layer.demo_combat_shoot");
                source.PlayOneShot(shotSounds[Random.Range(0,
shotSounds.Count)]);
            }
        }
        else timeSincePlayerSeen = 0;
    }
}
void ResetShot() => canShoot = true;
void CooldownShot()
{
    float distFromPlayer = (lastPositionPlayerSeen -
transform.position).magnitude;
    distFromPlayer = Mathf.InverseLerp(0, 500, distFromPlayer);
    float maxShots = Mathf.Lerp(15, 1, distFromPlayer);
    Debug.Log(maxShots);
    roundsLeftInBurst = (int)Random.Range(1, maxShots);
    onCooldown = false;
}
```

Now that the enemies can shoot back in a realistic manner depending on the distance from the player, I added a simple "whizz" mechanic where the enemy can detect if a bullet has barely missed them, and turn in the direction of where the shot came from. Then it was time to add the movement to the AI enemies. This was one of the hardest parts, as I used weighted chance to decide what the AI should do. On the first frame that the player is spotted, it will decide whether to fall back, or to push forward after a random amount of time. This means that they may initially stand in place and shoot back, but then reposition after 10-20 seconds. This is the method for calculating weighted chance. It takes into account the current health, if any enemies are nearby, the current number of rounds in the enemies mag, and a variable called "surprise."

```
void SpotPlayer()
    {
        if(isMoving) return;
        Debug.Log("Spotted");
        float healthChance = Mathf.InverseLerp(0, 100, health);
        float magChance = Mathf.InverseLerp(0, magSize, roundsInMag);
        float surpriseChance = Mathf.InverseLerp(0, 180, surprise);
                float friendsChance = Mathf.InverseLerp(0, 4,
nearbyFriends);
                //generate number from 0 to 1 based on the chances
        float chanceToPush = Mathf.InverseLerp(0, 4, healthChance +
magChance + surpriseChance + friendsChance);

        float decision = Random.Range(0f, 1f);
        //chanceToPush is a float between 0 and 1 that defines the threshold
of running away vs puhing
        if (decision < chanceToPush)
        {
            Invoke(nameof(RunAway), Random.Range(timeToMove.x,
timeToMove.y));
            Debug.Log("Running away");
        }
        else
        {
            Invoke(nameof(RunTowards), Random.Range(timeToMove.x,
timeToMove.y));
            Debug.Log("Pushing");
        }

    }
```

```
    void RunTowards()
    {
    //direction to search is towards player
        Vector3 dir = (GameManager.GM.player.transform.position -
transform.position).normalized;
```

```
                FindCover(dir);
    }
    void RunAway()
    {
    //direction to search is away from player
        Vector3 dir = -(GameManager.GM.player.transform.position -
transform.position).normalized;
                FindCover(dir);
    }
    void FindCover(Vector3 direction)
    {
        List<Vector3> checkPositions = new();

        Vector3 searchDir = direction;

        Debug.DrawRay(transform.position, searchDir, Color.red);
                //loop through 10 different possible positions
        for (int i = 0; i < 10; i++)
        {
            searchDir = new(Random.Range(searchDir.x - 30, searchDir.x +
30), transform.position.y, Random.Range(searchDir.z - 5, searchDir.z + 5));
            Vector3 position = transform.position - searchDir;
            checkPositions.Add(position);
            Debug.DrawLine(transform.position, position);
        }

        Vector3 bestCover = Vector3.positiveInfinity;
        float bestDist = 0;
        foreach (Vector3 pos in checkPositions)
        {
                //check if you can see the player from these positions
            if (Physics.Raycast(pos, searchDir, out RaycastHit hit,
Mathf.Infinity))
            {
                if (hit.transform.CompareTag("Player")) return;
                else
                {
                        //if you cannot see the player, check how far away
that point is from the player.
                    float dist = (GameManager.GM.player.transform.position -
pos).magnitude;
                    //save the point that is closest to the player
                    if (dist < bestDist) return;
                    else
                    {
                        bestDist = dist;
                        bestCover = pos;
                    }
                }
```

```
            }
        }
        tempTgt = bestCover;
        //Start movement after random time between 10 and 20 seconds
                Invoke(nameof(StartMovement), Random.Range(10,20));
    }
        void StartMovement(Vector3 tgt){
                finalTgt = tempTgt;
                isMoving = true;
        }
```

This makes the AI feel much more reactive and like they are actually thinking, which helps immersion greatly. If you catch the enemy by surprise, they will be more scared and run away, but if they are expecting you and have many friends nearby, they may be more aggressive and push you. I am very happy with how this AI has come out and it feels like a thinking agent that can execute different tactics very well. I added blood effects to the enemy, so it was obvious when you hit them, as well as a seperate head hitbox that would instantly kill them if you hit a headshot. Finally, I also added ragdolls when they die, so they fall to the ground in a realistic manner.