Progress Report 2

Introduction

CHIP – Counter Human Independence Protocol is a post-apocalyptic first-person shooter survival game in which the player needs to fight off swarming enemies and survive for as long as possible whilst collecting resources to progress to the next level. I am one of two programmers working on the game's development and this document serves as a progress report outlining the development process followed to create the player movement system and weapon system for the game. It also contains a reflection on time management and a plan for time allocation moving forward.

1. Player Movement Overview

Preproduction research for the project indicated that the player movement would consist of several movement states; idle, walking, crouching, jumping, sliding, jet-packing and grappling. The first four of these states were created as part of a pre-production experiment to determine a basic indication of how long the game's systems would take to develop. This allowed for rough time allocation to be done for the rest of the milestones which had to be reached for the project's development (explored further in time management).

The extension of this basic movement system into a more advanced one required a system which tracked which state the player was in and would change their state depending on a specific input. Some input would force the player from one state to another, for instance if the player was crouching but then started sprinting, the input acquired for the sprint to happen would force the player out of the crouching state and into the sprinting one. In order to track these state transitions effectively, a simple state machine system using booleans was set up.

Some state transitions depended on external factors other than input as well, for instance the player would not be able to jump if they were not on the floor, so an empty GameObject was added to the player object in the scene which would detect if the player was touching the ground.

```
//Controls
bool sprint = Input.GetKey(KeyCode.LeftShift) || Input.GetKey(KeyCode.RightShift);
bool jump = Input.GetKeyDown(KeyCode.Space);
bool slide = Input.GetKeyDown(KeyCode.C);
bool crouch = Input.GetKeyDown(KeyCode.C);


//States
bool isGrounded = Physics.Raycast(groundDetector.position, Vector3.down, 0.1f, ground);
bool isJumping = jump && isGrounded;
bool isSprinting = sprint && t_vmove > 0 && !isJumping && isGrounded;
bool isSliding = isSprinting && slide && !sliding;
bool isCrouching = crouch && !isSprinting && !isJumping && isGrounded;
```

1.1. Basic Movement

The basic movement for the game was done by capturing the player's input using Input.GetAxis and then using the results as the x and z components of a direction vector along which the player object was moved using the player GameObject's rigidbody velocity. When testing this movement system however, the movement did not feel right and often felt delayed. Further research indicated that this was due to the input being captured using Input.GetAxis as opposed to Input.GetAxisRaw (which snaps to the value 0 or +/-1 as opposed to gradually changing it[1]).

Jumping was achieved by adding an upwards force to the rigidbody attached to the game object. However, this jumping mechanic also felt rather slow and jagged. Up until this point, all of the code was being executed in the Update() method of the PlayerMovement script and applying a force here

was a mistake I had made before, instead this code should have been made in the FixedUpdate() method as then the force is applied every fixed frame as opposed to just every frame inside Update()[2].

Developing crouching provided its own unique challenges. Firstly, the camera had to be adjusted accordingly as the player crouched and stood up again. By storing the camera's original position in a variable called cameraOrigin and creating a variable to store the amount by which the camera's height needed to be adjusted when crouching, I was able to overcome this challenge as follows:

```
if (crouched) {
    normalCam.transform.localPosition = Vector3.Lerp(normalCam.transform.localPosition, cameraOrigin + Vector3.down * crouchAmount, Time.deltaTime * 8f);
}else{
    normalCam.transform.localPosition = Vector3.Lerp(normalCam.transform.localPosition, cameraOrigin, Time.deltaTime * 8f);}
```

The second issue was the colliders, as when crouched the player would have to be able to fit through smaller areas, as per a request from the game designer. To achieve this, I added two colliders to the player object and activated the respective collider depending on the player's state. Along with the collider and camera changes, crouching also needed to adjust the player's speed, which was done by altering the player GameObject's rigidbody velocity.

```
void SetCrouch(bool p_state)
{
    if (crouched == p_state) { return; }

    crouched = p_state;

    if (crouched)
    {
        standingCollider.SetActive(false);
        crouchingCollider.SetActive(true);
    }
    else
    {
        standingCollider.SetActive(true);
        crouchingCollider.SetActive(false);
    }
}
```

1.2. Sprinting and Sliding

Sprinting simply involved the addition of a method which would increase the player GameObject's rigidbody velocity by a certain amount. To further create the illusion of movement when sprinting, I decided to adjust the camera's field of view. This was done by first storing the camera's original field of view and then changing it by a modifier amount which could be set in the inspector. Snapping from one view to another, however, felt unrealistic so instead I used Mathf.Lerp which allowed for the camera's field of view to be interpolated from one value to another at a certain rate[3].

```
if (isSprinting) { normalCam.fieldOfView = Mathf.Lerp(normalCam.fieldOfView, baseFOV * sprintFOVModifier, Time.deltaTime * 8f); }
else { normalCam.fieldOfView = Mathf.Lerp(normalCam.fieldOfView, baseFOV, Time.deltaTime * 8f); }
```

By combing this field of view effect whilst using the camera changes done during crouching and further increasing the speed, I was able to create the movement mechanics for sliding.

1.3. Jetpacking

Using the isGrounded state and a variable tracking how much fuel the player has I was able to incorporate a jet pack system similar to the one described by the pseudo-code in my pre-production research. The refueling of the jetpack was done using two variables, currentRecovery which tracked how long it had been since the player last used the jet pack and jetWait, which determined the cool down time before the jetpack started refueling and the player was able to use it again. These variables could be set in the inspector allowing the game designer to tweak them to the achieve the desired effect.

```
if( grounded ){
        increase fuel
}else if( jumpButtonPressed AND haveFuel){
        add force to player
        decrease fuel
}
```

Pseudo Code

```
if (!isGrounded)
{
    canJet = true;
}
if (isGrounded)
{
    canJet = false;
}

if (canJet && jet && current_fuel >0)
{

    rig.AddForce(Vector3.up * jetForce * Time.fixedDeltaTime, ForceMode.Acceleration);
    current_fuel = Mathf.Max(0, current_fuel - Time.deltaTime*1.5f);

}
```

Implementation Code

```
if (isGrounded)
{
    if (current_recovery < jetWait)
    {
        current_recovery = Mathf.Max(jetWait, current_recovery + Time.fixedDeltaTime);
    }
    else
    {
        current_fuel = Mathf.Min(max_fuel, current_fuel + Time.fixedDeltaTime * jetRecovery);
    }
}
```

Jet Pack Cooldown System

1.4. Grappling Hook Shot

Although the implementation of the grappling hook system is similar to the one I planned for in the pre-production research, it turned out to be the most challenging aspect of the movement to develop as it did not simply contain one state, as I had previously anticipated it would. It actually had three separate states which each had to be tracked with its own boolean and depended on one another. The first of these states occurred when the player attempted to throw the grappling hook, at which point a ray had to be cast to determine if the direction the player was throwing it resulted in it colliding with something, this states was called grapStart. In the event that the ray collided with a surface the player was able to grapple towards, the next state the player would enter would be a waiting period in which the grappling hook would extend from their current position to reach the point of collision, this state was called thrownState.

Once the grappling hook had reached the destination, the player's transform would then be lerped towards that position in a state called hitGrap. Code was then written to apply the necessary adjustments depending on the state. The gravity on the rigidbody was switched off when the player was grappling and switched back on when the player reached the grappling hook collision position.

```
if( grapplingHookButtonPressed ){
        cast a ray
            if(rayCollidesWithSurface){
                store position of collision
                move player towards that location
                    if(playerPosition = collisionLocation){
                        stop moving player
                    }
            }
}
```

Pseudo Code

Because the Vector3.lerp used the change in position since the last frame relative to the total distance needed to be travelled to move the player, I needed to clamp the speed as the player would move very quickly in the beginning and then as they got closer to the destination would move incredibly slowly.

```
if (thrownState)
{
    rig.useGravity = true;
    hookshotTransform.LookAt(hookshotPosition);
    float hookShotThrowSpeed = 70f;
    hookShotSize += hookShotThrowSpeed * Time.fixedDeltaTime;
    hookshotTransform.localScale = new Vector3(1, 1, hookShotSize);
}
else if (hitGrap)
{
    rig.useGravity = false;
    float hookshotSpeedMin = 2f;
    float hookshotSpeedMax = 4f;
    hookShotSpeed = Mathf.Clamp(Vector3.Distance(transform.position, hookshotPosition), hookshotSpeedMin, hookshotSpeedMax);

    transform.position = Vector3.Lerp(transform.position, hookshotPosition, Time.fixedDeltaTime * hookShotSpeed);
}
```

Implementation Code

2. Weapon System Overview

For the weapon system, the game designer outlined that there would be a variety of weapons so in my pre-production research I determined that the easiest way to do this would be using a scriptable object for the guns, allowing different settings to be applied to each one and then to have a weapon script which would execute actions such as shooting and aiming by applying the settings of the weapon object currently equipped to the player.

2.1. Set Up

As the game is a first person shooter, the camera's perspective is that of the main character and in order for the gun's placement to seem realistic, I needed to create an empty game object that was positioned correctly on the player GameObject which would allow me to Instantiate a weapon as a child of this GameObject to ensure its positioning was correct. This GameObject was called Weapon and its transform was stored in the weapon script as weaponParent. Two more empty game objects were also added to this transform, one to store the position the gun would be at when hip-firing, and another to store the position the gun would move to when aiming. The guns default position would be the hip position.

2.2. Equipping

The weapon script was attached to the player and stored an array of Gun objects in a variable called loadout. This allowed each gun to be accessed and its respective attributes to be applied to each of the methods in the weapon script. The first of these methods was the Equip Method, which took in an index parameter, the method would check if the player currently had a gun equipped, and if not would set the gun object in the loadout script at the index provided as the current weapon. The corresponding Gun object's prefab would then be instantiated at the weaponParent transform's position (outlined in Set Up above). If the player already had a weapon equipped, this weapon would be destroyed, and replaced with the new one.

```
public void Equip(int p_ind)
{

    if (currentWeapon != null){
        Destroy(currentWeapon);
    }

    currentIndex = p_ind;

    GameObject t_newEquipment = Instantiate(loadout[p_ind].prefab, weaponParent.position, weaponParent.rotation, weaponParent) as GameObject;
    t_newEquipment.transform.localPosition = Vector3.zero;
    t_newEquipment.transform.localEulerAngles = Vector3.zero;

    currentWeapon = t_newEquipment;
}
```

2.3. Aiming

The method for aiming involved lerping the current weapon from its hip position to its aim-down sight position and setting a Boolean which stored whether the player was aiming or not.

```
void Aim(bool p_isAiming)
{
    isAiming = p_isAiming;
    Transform t_anchor = currentWeapon.transform.Find("Anchor");
    Transform t_state_ads = currentWeapon.transform.Find("States/ADS");
    Transform t_state_hip = currentWeapon.transform.Find("States/Hip");

    if (p_isAiming)
    {
        t_anchor.position = Vector3.Lerp(t_anchor.position, t_state_ads.position, Time.deltaTime * loadout[currentIndex].aimSpeed);
    }
    else
    {
        t_anchor.position = Vector3.Lerp(t_anchor.position, t_state_hip.position, Time.deltaTime * loadout[currentIndex].aimSpeed);
    }


}
```

2.4. Shooting

The shooting method's complexity had to be changed as the development of the weapon's system progressed. Initially the implemented system consisted of code similar to the pseudo-code presented in the pre-production research but this shooting system proved to not be adequate for the game's scope.

```
if( fireButtonPressed AND haveAmmo ){
        cast a ray from tip of gun
        decrease ammo
        if(rayCollidesWithEnemy){
                deal corresponding damage to enemy
        }
```

Pseudo Code

```
void Shoot()
{
    Transform t_spawn = transform.Find("Cameras/Player Camera");

    //raycast
    RaycastHit t_hit = new RaycastHit();
    if (Physics.Raycast(t_spawn.position, t_spawn.forward, out t_hit, 1000f, canBeShot))
    {
        if (t_hit.collider.gameObject.layer == 12)
        {
            t_hit.collider.gameObject.GetComponent<EnemyController>().TakeDamage(loadout[currentIndex].damage);
        }
    }
}
```

Initial Implementation

The first extension that needed to be made to the system was some way of incorporating accuracy, as currently every “shot” would hit the exact same position, as the ray was being cast straight from the center of the screen every single time. To add an accuracy component to this system, I used the bloom attribute stored on each gun object. To randomize the position from which the ray would be cast I multiplied the Up and Right directional vectors of the camera’s position by a random amount and then added that to a variable called t_bloom. I then normalized this vector and used it in the Raycasting, the normalization was done to keep its direction but reduce it magnitude to 1 as the Raycasting would determine how far out the ray should be cast.

```
Vector3 t_bloom = t_spawn.position + t_spawn.forward * 1000f;
t_bloom += Random.Range(-loadout[currentIndex].bloom, loadout[currentIndex].bloom) * t_spawn.up;
t_bloom += Random.Range(-loadout[currentIndex].bloom, loadout[currentIndex].bloom) * t_spawn.right;
t_bloom -= t_spawn.position;
t_bloom.Normalize();

//raycast
RaycastHit t_hit = new RaycastHit();
if (Physics.Raycast(t_spawn.position, t_bloom, out t_hit, 1000f, canBeShot))
```

Accuracy Adjustment

The second major adjustment came when the game designer sent me the list of guns the game would have to include. Up until this point I had been using a simple Pistol gun to test the methods, but the list included an assault rifle, SMG, shotgun and RPG. Whilst the current shooting method would work for the Pistol, Assault Rifle and SMG by simply adjusting the damage, fire-rate, bloom and kick back attributes on the respective Gun Objects, it would not work the shotgun, as the shot gun would need to shoot multiple bullets simultaneously. To achieve this, I added a new attribute to the Gun scriptable object called “Pellets” which would store how many bullets the gun would shoot at once, then by enclosing the current shoot method in a for loop which would repeat for the number of pellets the current weapon’s Gun Object had, I was able to create the shotgun effect.

```
for (int i = 0; i < Mathf.Max(1,loadout[currentIndex].pellets); i++)
```

Pellet Adjustment

The final adjustment that needed to be made was to cater for the RPG. Currently the weapon system used Ray Casting to deal damage to enemies. The RPG however would need a projectile-base system to function correctly. First, I created a projectile that the RPG could shoot and added an Explode script to it which would destroy the game object and deal a certain amount of damage to anything within a predefined radius and on the correct layer.

```
Collider[] enemies = Physics.OverlapSphere(transform.position, explosionRange, whatIsEnemies);
for (int i = 0; i < enemies.Length; i++)
{
    enemies[i].GetComponent<EnemyController>().TakeDamage(explosionDamage);
}
```

I then added three new attributes to the Gun Object; a boolean isProjectile which would be used in the shoot method to determine which shooting mechanic should be used for the gun, projectilePrefab which stored the projectile prefab containing the explode script and projectile force which determined how much force would be added to the projectile once it had been instantiated. I also needed to add an empty game object to the RPG gun prefab and position it correctly as point where the projectiles could be instantiated.

```
if (loadout[currentIndex].isProjectile)
{

    Transform shootPoint = currentWeapon.transform.Find("ShootPoint");

    GameObject currentBullet = Instantiate(loadout[currentIndex].projectilePrefab, shootPoint.position, Quaternion.identity);
    currentBullet.transform.forward = t_bloom;
    currentBullet.GetComponent<Rigidbody>().AddForce(t_bloom * loadout[currentIndex].projectileForce, ForceMode.Impulse);
}
```

Time management

Communicating with the game designer I was able to layout each of the systems I would need to develop for the game:

1. Movement System
2. Weapon System
3. Enemy Movement System
4. Powerup and Equipment System
5. Currency System

The total time frame for the game's development is 12 weeks, but the first two weeks were focused predominantly on preproduction research, so overall there are approximately 10 weeks. Currently, integration has been limited as each developer has been working independently on their respective component of the game, but I predict that as we begin to integrate, we will face new challenges. Also, as we approach the final submission, we will be approaching exam season which comes with a lot of stress from other subjects as well. To prevent a situation in which I run out of time and am not able to develop exactly what is expected of me, I am aiming to complete one system a week, allowing for the base mechanics to be done by week 7 ensuring there is enough time for integration, external playtesting and polishing.

As mentioned earlier my preproduction planning involved an experiment which allowed me to roughly estimate how long certain system would take to develop. I concluded that each milestone would take roughly 3 hours to complete. Our project manager set up a Trello Board which is a collaborative tool that allows teams to work together more effectively[4]. One of Trello's features is that it allows you to track the time you spend on each task using a built-in timer. In order to complete a system a week , I would need to break it down into its milestones and assign time for each of those milestones. Below is a comparison of the time predicted for each milestone outlined in this document and the actual amount of time it took to develop.

Week 1 - Movement System

Milestone	**Estimated Time**	**Recorded Time**
Sprinting and Sliding	3 hours	2 hours 43 minutes
Jetpacking	3 hours	1 hour 35 minutes
Grappling Hookshot	3 hours	3 hours 18 minutes
Total:	**9 hours**	**7 hours 36 minutes**

Week 2 – Weapon System

Milestone	**Estimated Time**	**Recorded Time**
Equipping	3 hours	1 hour 57 minutes
Aiming	3 hours	54 minutes
Shooting	3 hours	5 hours 22 minutes
Total:	**9 hours**	**8 hours and 13 minutes**

Whilst the prediction for individual components proved ineffective, as some milestones took a lot longer or a lot less time than estimated, the overall estimated time for each week was close to the recorded time and in both cases I had time to spare. Moving forward I will continue to use an estimation of 9 hours per system, meaning I need to allocate that amount of time each week.

We also have two team meetings during the week, on Mondays and Thursdays, I think these also need to be included in my time management going forward. The 4 meetings we had over the passed two weeks average just over an hour, so to be safe, I will allocate an hour and a half for each of them, three hours total. This means that I can use an estimation of 12 hours a week to develop the necessary system and have team meetings.

References

1- Answers.unity.com. 2014. Difference Between Input.Getaxis And Getaxisraw? - Unity Answers. [online] Available at: <https://answers.unity.com/questions/728949/difference-between-inputgetaxis-and-getaxisraw.html> [Accessed 27 August 2020].
2- Sahota, R., 2017. Fixedupdate Vs Update | Treehouse Community. [online] Treehouse. Available at: <https://teamtreehouse.com/community/fixedupdate-vs-update> [Accessed 26 August 2020].
3- Unity - Scripting API: Mathf.Lerp. [online] Docs.unity3d.com. Available at: <https://docs.unity3d.com/ScriptReference/Mathf.Lerp.html> [Accessed 26 August 2020].
4- Trello.com. 2020. Trello. [online] Available at: <https://trello.com/en> [Accessed 27 August 2020].