FPS GAME DEVELOPMENT USING UNITY

Submitted in partial fulfillment of the requirements of the degree

(S.E) BACHELOR OF ENGINEERING IN

COMPUTER ENGINEERING

By

Tanmay Khule
Tanmay Kulkarni
Nishan Matre
Riddhi More

Supervisor

Prof. Ranjana Singh

Department of Computer Engineering
Watumull Institute of Electronics Engineering and Computer Technology
Ulhasnagar (W) – 421003

University of Mumbai

(AY 2020-21)

CERTIFICATE

This is to certify that the Mini Project entitled **"FPS GAME DEVELOPMENT USING UNITY"** is a bona fide work of following students submitted to the University of Mumbai in partial fulfillment of the requirement for the award of the degree of **"(S.E) Bachelor of Engineering"** in **"Computer Engineering".**

Tanmay Khule
Tanmay Kulkarni
Nishan Matre
Riddhi More

(Prof. Ranjana Singh)
Supervisor

(Prof.__________________)
Head of Department

(Prof.__________________)
Principal

Mini Project Approval

This Mini Project entitled "***FPS GAME DEVELOPMENT USING UNITY*"** by following students is approved for the degree of **(S.E) Bachelor of Engineering** in **Computer Engineering.**

Tanmay Khule
Tanmay Kulkarni
Nishan Matre
Riddhi More

Abstract

The paper title "FPS game using unity" is a game development project for developing a first-person shooter survival game. In this paper, we develop the game using the popular game engine Unity and scripting in the language C#.

We were amazed by the immersive graphics, realistic controls and thrilling gameplay that many modern-day games provide us. Which left us wondering how they were developed, and so we started exploring.
This is when we decided to actually develop a game ourselves.

"Elpida - The Last Hope" is a 64-bit windows application developed using Unity and C#, it uses Unity latest HDRP render pipeline for developing realistic graphics.

The game starts with our main player being spawned in a map filled with zombies. He has to travel to the other side of the map without being killed by these zombies. The theme of the game is therefore post-apocalyptic gloomy weather with fog in it.

The game can be broken down into various modules as follows: 1) The Player 2) Gun and Shooting 3) Zombie and its animations 4) UI canvas 5) Map and post-processing. Later we will look into all those in detail. All these modules interact with each other to complete the game and provide an immersive experience to the user.

Acknowledgement

We Would like to add a few words of Appreciation for the People who have been a Part of this Project right from its Inception. The writing of this Project has been one of the significant Academic Challenges we have Faced and without the support, Patience & guidance of the People involved, this task would not have been completed. It is to them We owe Our deepest gratitude.

It gives us Immense pleasure in Presenting this Project Report on ***"FPS Game Development Using Unity"****. The Success of this project is a result of sheer hard work, and determination put in by us with the help of our Project guide. We hereby take this opportunity to add a special note of thanks for* ***Prof. Ranjana Singh*** *who undertook to act as our mentor despite her many other academic & professional commitments. Her wisdom, Knowledge & commitment to highest standards inspired and motivated us. Without her insight, support and energy, this project wouldn't have kick-started & neither would have reach fruitfulness.*

We also feel heartiest sense of obligation for Our H.O.D ***Prof. Nilesh Mehta*** *for guiding us & giving us Proper direction to make our Project One of the Best. Without his kind & keen co-operation we wouldn't be able to learn & apply core aspects of engineering in our Project easily. Last but not the least we express our Sincere thanks to all of our friends, seniors and our parents who Patiently extended all sorts of help for accomplishing this Undertaking.*

Contents

Table of figures

1. Introduction

Elpida is a first-person shooter game developed in unity and scripted in C#.

It is a survival game; the player has to cruise through the map, player has various movements like walking running crouching and jumping.

The map will be dynamic and will transition from day to night, it has various realistic elements like fog and waterfall. Various props like bridges and houses are placed throughout the map.

Map is filled with blood thirsty zombies, who will not spare your life if you get close to them. They will be dormant unless player enters a specific radius, once the player enters the attacking radius, zombies will attack and player will take damage.

The player can use health packs to heal the damage, he can kill the zombies using pistol, rifle or knife.

Health packs and ammo boxes will be spawned at fixed locations throughout the map. Health packs heal the player by specified health points.

1.1 Motivation

The reason for choosing this project is our sheer interest and curiosity.

We see immersive graphics, realistic controls and thrilling gameplay. Which left us wondering how they were developed, and so we started exploring. We came across different game engines like Unreal Engine, Amazon Lumberyard, CryEngine, Unity. We chose Unity.

Though we decided to develop a game we still had to decide on what type of game we should make and how to make it more interesting and fun to play. Different ideas and discussing them developed our creative thinking and gaming logic.

We wanted to make a game that would provide an adrenaline rush to our user. Also, we wanted the game to have a post-apocalyptic feel to it.

We were amazed by the idea of FPS games how one could see the world we created in our game from player view which would provide him an immersive experience.

So, after a lot of thinking we decided to make Elpida a first-person shooter survival game.

1.2 Problem Statement & Objectives

Our problem statement revolves around game development using UNITY and Learning a new language C# and implementing these learning to create a first-person survival thriller game.

The map must be realistic, it should have a post-apocalyptic feel and must transition from day to night.

The player should have basic movements and must be able to take damage and heal.

Zombies should spawn at random locations and must be able to move randomly and attack the player.

Weapons must be able to perform basic functions like shooting, reloading scope out and scope in mechanisms.

1.3 Report Organization

The report is broken into various parts firstly the introduction itself, why we choose the project, what was the motivation behind it, then we go on to explaining the various modules of our game and how they work together to make our game. In the end there are some references.

2. Literature Survey

What is a First-person shooter (FPS) game?
First-person shooter (FPS) is a video game genre centered on gun and other weapon-based combat in a first-person perspective; that is, the player experiences the action through the eyes of the protagonist. The genre shares common traits with other shooter games, which in turn makes it fall under the heading action game. Since the genre's inception, advanced 3D and pseudo-3D graphics have challenged hardware development, and multiplayer gaming has been integral.

The first-person shooter genre has been traced as far back as Wolfenstein 3D, which has been credited with creating the genre's basic archetype upon which subsequent titles were based. One such title, and the progenitor of the genre's wider mainstream acceptance and popularity, was Doom, one of the most influential games in this genre; for some years, the term Doom clone was used to designate this genre due to Doom's influence. Corridor shooter was another common name for the genre in its early years, since processing limitations of the era's hardware meant that most of the action in the games had to take place in enclosed areas, such as in cramped spaces like corridors and tunnels.

Gameplay and Features of FPS Games First-person shooters often focus on action gameplay, with fast-paced and bloody firefights, though some place a greater emphasis on narrative, problem-solving and logic puzzles. In addition to shooting, melee combat may also be used extensively. In some games, melee weapons are especially powerful, a reward for the risk the player must take in maneuvering his character into close proximity to the enemy. In other situations, a melee weapon may be less effective, but necessary as a last resort.

First-person shooters typically give players a choice of weapons, which have a large impact on how the player will approach the game. Some game designs have realistic models of actual existing or historical weapons, incorporating their rate of fire, magazine size, ammunition amount, recoil and accuracy. Other first-person shooter games may incorporate imaginative variations of weapons, including future prototypes, "alien technology" scenario defined weaponry, and/or utilizing a wide array of projectiles, from industrial labor tools to laser, energy, plasma, rocket and grenade launchers or crossbows. These many variations may also be applied to the tossing animations of grenades, rocks, spears and the like. Also, more unconventional modes of destruction may be employed from the viewable users' hands such as flames, electricity, telekinesis or other supernatural abilities, and traps. However, designers often allow characters to carry varying multiples of weapons with little to no reduction in speed or mobility, or perhaps more realistically, a pistol or smaller device and a long rifle or even limiting the player to only one weapon at a time.

First-person shooters may be structurally composed of levels, or use the technique of a continuous narrative in which the game never leaves the first-person perspective. Others feature large sandbox environments, which are not divided into levels and can be explored freely. In first-person shooters, protagonists interact with the environment to varying degrees, from basics such as using doors, to problem solving puzzles based on a variety of interactive objects. In some games, the player can damage the environment, also to varying degrees: one common device is the use of barrels containing explosive material which the player can shoot, destroying himself and harming nearby enemies. Other games feature environments which are extensively destructible, allowing for additional visual effects. The game world will often make use of science fiction, historic (particularly World War II) or modern military themes, with such antagonists as aliens, monsters, terrorists and soldiers of various types. Games feature multiple difficulty settings; in

harder modes, enemies are tougher, more aggressive and do more damage, and power-ups are limited. In easier modes, the player can succeed through reaction times alone; on more difficult settings.
Some popular FPS games: P.U.B.G, Call of Duty, Raven Holm etc.
Game Engines:
A **game engine**, also known as a **game architecture**, **game framework** or **game frame**, is a software-development environment designed for people to build video games. Developers use game engines to construct games for consoles, mobile devices, and personal computers. The core functionality typically provided by a game engine includes a rendering engine ("renderer") for 2D or 3D graphics, a physics engine or collision detection (and collision response), sound, scripting, animation, artificial intelligence, networking, streaming, memory management, threading, localization support, scene graph, and may include video support for cinematics. Implementers often economize on the process of game development by reusing/adapting, in large part, the same game engine to produce different games or to aid in porting games to multiple platforms. The actual game logic has to be implemented by some algorithms. It is distinct from any rendering.

A subset of game engines are 3D first-person shooter (FPS) game engines. Ground-breaking development in terms of visual quality is done in order to get FPS games to its current standard. The level of visual details emphasized in these games have become increasingly precise, something that engines focused on flight and driving simulators and real-time strategy (RTS) games don't contain.

The development of the FPS graphic engines that appear in games can be characterized by a steady increase in technologies, with some breakthroughs. Attempts at defining distinct generations lead to arbitrary choices of what constitutes a highly modified version of an "old engine" and what is a brand-new engine.

The classification is complicated as game engines blend old and new technologies. Features that were considered advanced in a new game one year become the expected standard the next year. Games with a mix of older generation and newer feature are the norm.

Some Popular Game Engines:

Unreal Engine:

One of the most popular and widely used game engines is the Unreal Engine by Epic Games.
The original version was released in 1998 and 17 years later it continues being used for some of the biggest games every year.
Notable titles made with Unreal Engine include the **Gears of War** series, **Mass Effect** series, **Bioshock** series, and the **Batman: Ark ham** series.
The strength of the Unreal Engine is its ability to be modified enough that games can be made into very unique experiences.
The latest version, Unreal Engine 4, is said to be the easiest one to use when in the hand tools of a professional.

Unity:

One of them is Unity, a multi-platform game engine that allows you to **create interactive 3D content with ease**.
A lot of indie developers use Unity for its excellent functionality, high-quality content, and ability to be used for pretty much any type of game.
Recent notable titles made with Unity include **Lara Croft Go**, **Her Story**, **Pillars of Eternity**,

and **Kerbal Space Program**.
One of the best things about Unity 5 is the Personal Edition, which is free for everyone to download. It includes the engine with all features and can (for the most part) be used to make games on every platform.

GameMaker:
Unlike most other game engines, **GameMaker: Studio has become widely used** because **it doesn't require programming knowledge** to use.
Instead, users can "point-and-click" to create games much easier and faster than coding with native languages.
Some of the best titles made with GameMaker include **Spelunky**, **Hotline Miami**, **Super Crate Box**, and the upcoming **Hyper Light Drifter**.
GameMaker is popular because you can make a game without having to learn a programming language first, and those that do have coding experience can use it to make their game better.
The problem with GameMaker and other point-and-click engines is that **developers are much more limited than with other engines**.

3. Requirements

3.1 Hardware

1. Unity Software
2. Unity Teams
3. C-# (for scripting various features in the game)
4. Visual Studio Code
5. Windows Operating – System (7 SP1+, 8, 10, 64-bit versions only)

3.2 Software

Graphics card with DX10 (shader model 4.0) capabilities.

4. Design and Implementation

4.1 Character Controller

To start with the game first of all we need a character that moves, to do this we create a character controller.

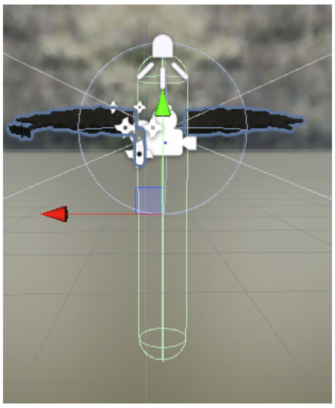


Fig 1: Character controller capsule

Character controller is a capsule which is responsible for character movements. It helps to perform movements of character without having to deal with rigid body dynamics. It has a capsule collider hence has movement constraints. As it is not a rigid body no forces affect it only move function can move the character controller.

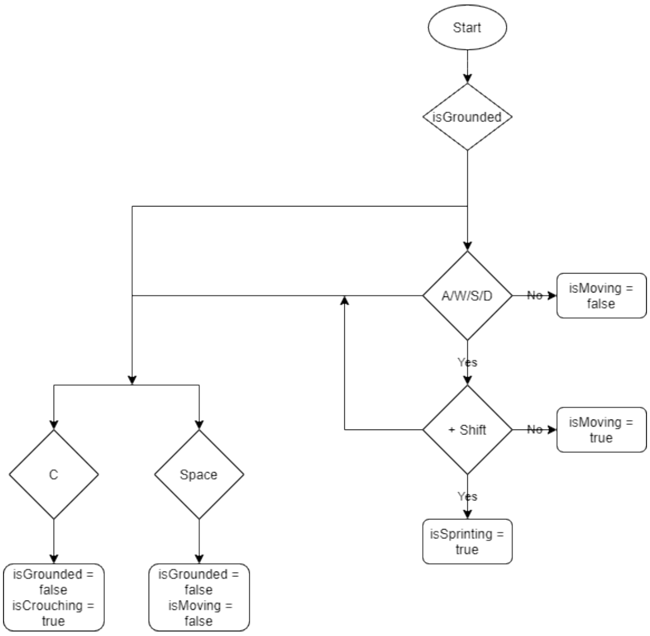


Fig 2: Character Flow control

4.1.1 isGrounded

As we can see from above for any movement to be performed character must be grounded, we check that by a Boolean isGrounded.

```
public bool isGrounded;
```

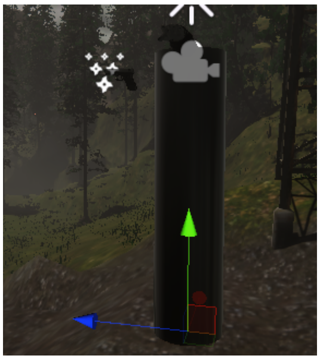


Fig 3:to check if grounded

An empty game object **Ground Check** is assigned to the bottom of the player capsule. To check whether the player is grounded or not, the **CheckSphere** function creates an imaginary sphere of given radius and Returns true if there are any colliders overlapping the sphere defined by position and radius in world coordinates.

```
isGrounded = Physics.CheckSphere(groundCheck.position, groundDistance, groundMask);
```

4.1.2 Movements

Linear movements of the objects are controlled by A\W\S\D keys. We calculate the change in two directions of the player using **GetAxis** function

```
float x = Input.GetAxis ("Horizontal");
float z = Input.GetAxis ("Vertical");
```

The **GetAxis** function gets change in horizontal and vertical axis since the last update and stores it in float x and y respectively.

```
Vector3 move = transform.right * x + transform.forward * z;
```

Vector3 is a datatype in unity. This structure is used throughout Unity to pass 3D positions and directions around. It also contains functions for doing common vector operations.

```
controller.Move (move * characterSpeed * Time.deltaTime);
```

Move is a function of character controller which moves the controller in the direction of specified motion Vector3. The motion vector is multiplied by **Time.deltaTime** to make it update time independent.

4.1.3 Sprinting

The character is sprinting when the Boolean is running is true.

```
        public bool isRunning;
```

If ‘Shift + A\W\S\D’ keys are held then the character will sprint and the Boolean isRunning to True.

```
if (Input.GetKeyDown (KeyCode.LeftShift) && isGrounded){
characterSpeed = runSpeed;
isRunning = true;
}
```

4.1.4 Jumping

For Jumping if the character is grounded and space is pressed, the character will get a velocity in positive y direction, and the character will move up.

```
        if (Input.GetButtonDown ("Jump") && isGrounded){
                velocity.y = Mathf.Sqrt (jumpHeight * -2f * gravity);
        }
```

The character will not translate motion will be only in y direction while jumping

4.1.5 Gravity

For gravity the character controller needs velocity in negative y direction.

```
        velocity.y += gravity * Time.deltaTime * 2f;
```

4.1.6 Crouching

The character will crouch if the Boolean isCrouching is true.

```
        public bool isCrouching;
```

For crouching if Key ‘C’ is held down then the character will crouch and his speed will be reduced to crouching speed and the height of character will be reduced to 0.9f.

```
        if (Input.GetKeyDown (KeyCode.C) && isGrounded){
                isCrouching = true;
                controller.height = 0.9f;
        }
```

4.2 Raycast

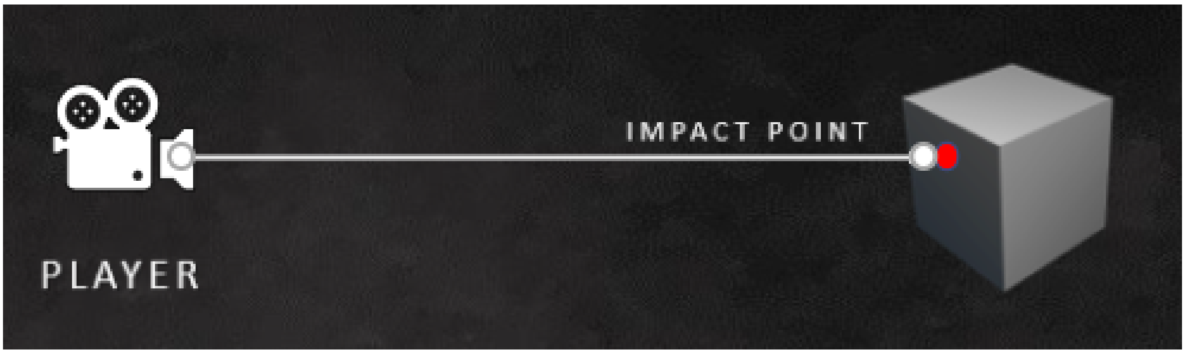


Fig 4: Raycast

In Our Project in Unity, we’ve used the Raycast Concept for Shooting as well as Picking up Objects. Here, as per this diagram the thin Invisible Ray is Starting from the Position of our Player Camera in World Co-ordinates & will be going in Forward Direction we’re facing & when it will hit Something, we’ll gather Information about that object Like Colour, name, Position, etc. of that Object. We can apply this Raycast, Ray hitting Concept to any Specific Object. Also, we can get the Angle by which a Ray is Incident on the Object.

4.3 The weapons

4.3.1 Shooting & Melee Attack

First here declared our Shooting Class as Public & in that we’ve some public variables for further use & pubic variables in Ammo part Section are also going displayed in Unity Inspector as they’re under Header Tag.

```
public class Shooting : MonoBehaviour
{
    public Camera Camera;
    public WeapoSwitcher weapontype;
    public float Range = 100f;
    public float fireRate = 15f;
    private float nextTimetoFire = 0f;

    // Ammo Part
    [Header("Ammo Management")]
    public int pistolCapacity = 6;
    public int rifleCapacity = 20;
    public int pistolAmmoCount = 6;
    public int rifleAmmoCount = 20;
    public float reloadTime = 3f;
    public bool isReloading = false;
    public int availableAmmo = 50;
    public int maxavailableAmmo = 50;
    public bool isFiring = false;
    public bool isFull = true;
```

Now, here in Update Function we decide that when we've to Actually Shoot From Gun. If we press Left Click & with that Time.time is greater than equals nextTimetoFire then it checks if we've Ammo in our Gun by second "if" condition, after that we'll check the selected Gun like if it's Pistol having Index 0 or Rifle having Index 1 And then Particularly Checking all these Conditions we'll Decrease Ammo in our Gun & recalculate nextTimetoFire. Greater the Fire rate Less time Between the Shots.

```
    // Update is called once per frame
    void Update()
    {
        if (Input.GetButton("Fire1") && Time.time >= nextTimetoFire )
        {
            isFiring = true;
            if (rifleAmmoCount > 0 && pistolAmmoCount > 0)
            {
                if (weapontype.SelectedWeapon == 0)
                {
                    pistolAmmoCount -= 1;
                }
                if (weapontype.SelectedWeapon == 1)
                {
                    rifleAmmoCount -= 1;
                }
                nextTimetoFire = Time.time + 1f / fireRate;
                weapon();
            }
        }
```

Here also we Use Same Logic as above when we Press Right Click. This Code is specifically For the Zoom/Scope In Firing.

```
        else if (Input.GetButton("Fire2"))
        {
            isFiring = true;
            if (rifleAmmoCount > 0 && pistolAmmoCount > 0 && Time.time >= nextTimetoFire)
            {
                if (weapontype.SelectedWeapon == 0)
                {
                    pistolAmmoCount -= 1;
                    pistolFlash.Play();
                }
                if (weapontype.SelectedWeapon == 1)
                {
                    rifleAmmoCount -= 1;
                    rifleFlash.Play();
                }
                nextTimetoFire = Time.time + 1f / fireRate;
                weapon();
        }
      }
```

Here We do Knife Attack if AlphaKeyword 3 is pressed & we’re not Firing at same time.

```
      if (Input.GetKeyDown(KeyCode.Alpha3) && isFiring == false){
          meleeAttack();
      }
```

If any Of Above Firing Conditions are not Satisfied then Bool isFiring Sets To False.

```
      if (Input.GetButtonUp("Fire1") || Input.GetButtonUp("Fire2")){
          isFiring = false;
      }
    }
```

Now Here, we actually implement Our Raycast Concept that we saw Previously to shoot & Damage Our Enemy. In Weapon Function if firing Condition is true and Ammo is there in Gun, then we create this hit variable which will store information about what we’re hitting. Now as per this Condition we Access our Camera Position to start casting ray then the Ray will be going in Forward Direction w.r.t Camera & hit variable will store the Object information that we hit automatically. Hence, if we Simply hit any Solid Colliding Object as per this Condition then we’ll check if the it do have Tag of Enemy to it. If Yes then we’ll Simply Destroy Our Enemy here Zombies by accessing Damage Component of it & Passing the Value Of Damage we want to give to Enemy(here 20f) to Damage Function of another Script & Print the message of “Bullet Hit”.

```
    void weapon()
    {
      isFiring = true;
      isFull = false;
      RaycastHit hit;
      if (Physics.Raycast(Camera.transform.position, Camera.transform.forward, out hit))
      {
          if (hit.transform.tag == "Enemy")
          {
              damage enemy = hit.transform.GetComponent<damage>();
              enemy.Damage1(20f);
              Debug.Log("Bullet Hit");
          }
      }
    }
```

The Same above Logic we do apply for the Knife/Melle Attack & Print “Melle” On Console.

```
void meleeAttack () {
      RaycastHit hit;
      if (Physics.Raycast(Camera.transform.position, Camera.transform.forward, out hit))
      {
          if (hit.transform.tag == "Enemy" && hit.distance <= 5f)
          {
              damage enemy = hit.transform.GetComponent<damage>();
              enemy.Damage1(20f);
              Debug.Log("Melee");
```

```
            }
        }
    }
}
```

4.3.2 Reloading

Reloading Concept with the help of our Code is Explained Below.
First of all here, We Check if we've to Reload Gun or not by Simply Checking if Ammo Of anyone of our Gun is Finished or not as well as User has Pressed Key 'R' or Not. If anyone of this condition satisfies then we'll Check Gun we Currently holding & will Call Reload Function respective to it.

```
void Update()
  {
      if (pistolAmmoCount == 0 || rifleAmmoCount == 0 || Input.GetKeyDown(KeyCode.R))
      {
         if (weapontype.SelectedWeapon == 0)
         {
            isReloading = true;
            StartCoroutine(Reload());
            return;
         }
         if (weapontype.SelectedWeapon == 1)
         {
            isReloading = true;
            StartCoroutine(Reload());
            return;
         }

      }
```

When Any of Above Condition arises we'll call Reload Function in which this main logic will Execute Reloading Operation. Reloading can only happen if availableAmmo means total ammo in our pocket is greater than 0, if pocket is empty then we can't reload. & also currentAmmo of our selected weapon should be less than or equals to 0; as here its divided for pistol as well as rifle weapon. After that we'll check the Condition of respective weapon & will select the one we'll be holding Currently. Then, we'll decrease the available i.e. ammo in our pocket & increase the CurrentammoCount means the ammo in the gun till it reaches to the condition that ammo capacity of gun equal equal to CurrentammoCount of gun. When this condition will satisfy we'll break the While Loop & hence our Reload of Gun has been done Succesfully!!

```
      IEnumerator Reload()
      {
         isReloading = true;
         yield return new WaitForSeconds(reloadTime);
         while (availableAmmo > 0 && (pistolAmmoCount < pistolCapacity || rifleAmmoCount <
rifleCapacity))
         {
            if (weapontype.SelectedWeapon == 0)
            {
               pistolAmmoCount += 1;
            }
            if (weapontype.SelectedWeapon == 1)
            {
                rifleAmmoCount += 1;
            }
            availableAmmo -= 1;
        }
        isReloading = false;
    }
```

4.3.3 Damaging

To give Damage & Kill Zombies the Damage Script is explained below Shortly.
Now when the Damage1 function is called from the Shoot script we get that parameter value of damage to be caused to enemy here 20f in damage script. As all functions & variables are public we can access them from other scripts. Now, when the parameter value is passes in Damage1 Function it decreases the health of player which is only 20f here. So, Logically Zombie will Die in One Shot & Once his health becomes to 0 then We'll Print that Specific message. Bool 'isDead' will be True now as Enemy has died & we'll Call the Function Invoke() where gameObject having Enemy tag means Our Zombie will get Destroyed after Delay of 2f(2 Seconds).

```
    public Animator anim;
    public float health = 20f;
    public bool isDead = false;

    public void Damage1(float amount)
    {
        health -= amount;
        if (health <= 0f)
        {
            Debug.Log("dead");
            isDead = true;
            Invoke("Destroyobj",2f);
        }
    }
    public void Destroyobj ()
{
        Destroy (gameObject);
    }
```

4.3.4 Weapon Switching

To Switch between Weapons in GamePlay, we Use Below Logic & Code Given.
After Creating the Class WeapoSwitcher & initializing SelectedWeapon to '0' with declaration of pistol and rifle Objects in the Class; In Start Function we call SelectWeapon Function & so, first default Pistol Gun is Selected while Game Starts as it have index 0.

```
void Start()
    {
        SelectWeapon();
        pistol.SetActive(false);
        rifle.SetActive(false);
    }
```

Here, in SelectWeapon Function we use foreach loop in that our each Transform weapon here pistol & Rifle will be checked from our current transform WepoSwitcher. As we don’t have any facility to initialize i value in loop we simply initialize it to 0 here. Then we check if I value matches to SelectedWeapon value if yes then we enable that gameobject here Pistol has i=0 as it’s first child transform & others will be made disable.

```
//To Switch between Weapons.
//To Switch between Weapons.
    void SelectWeapon()
    {
        int i = 0;
        foreach (Transform weapon in transform)
        {
            if (i == SelectedWeapon)
            {
                weapon.gameObject.SetActive(true);
            }
            else
            {
                weapon.gameObject.SetActive(false);
            }
            i++;
        }
    }
```

Now, the condition here is that if we press Alpha 2 key & if we actually have that Gun Rifle in Our Gun Hierarchy then we’ll get SelectedWeapon as Rifle as per it’s index value from foreach loop but it’ll also check if our PreviousSelectedWeapon should not match with current SelectedWeapon as we want to switch weapons from one to another not from same weapon to again same one. And, then Lastly Checking the Index Value of SelectedWeapon we Set the Respective Weapon according to its Assigned Index value To SetActive & hence we Succesfully change the Weapon.

```
void Update()
    {
        int previousSelectedWeapon = SelectedWeapon;
        if (Input.GetKeyDown(KeyCode.Alpha1))
        {
            SelectedWeapon = 0;
        }
        if (Input.GetKeyDown(KeyCode.Alpha2) && transform.childCount >= 1)
        {
            SelectedWeapon = 1;
        }
        if (previousSelectedWeapon != SelectedWeapon)
        {
            SelectWeapon();
        }
        if (SelectedWeapon == 0){
            pistol.SetActive(true);
            rifle.SetActive(false);
        }
        if (SelectedWeapon == 1){
            rifle.SetActive(true);
            pistol.SetActive(false);
        }
    }
```

4.4 Pickups

4.4.1 Ammo Picking

The Short Explaination With help of Code about Picking up any Objects Specifically Ammo here is there below.
All the Variables & Instances of Object that we're gonna make Use are first Declared by Creating PickUpObj Class as Public.

```
using UnityEngine;
public class PickUpObj : MonoBehaviour
{
    Ray ray;
    RaycastHit hit;
    [SerializeField]
    float pickupDistance = 5f;
    public Camera mainCam;
    public Shooting shootscript;
    public LayerMask layer;
```

Now, To pick up Ammo we use Similar Raycast Concept that we saw till now but in a bit different way. As per this Code Snippet we can see that we're starting our Ray from our Player Camera as mainCam named & setting it to the middle of the screen as x &y Co-ordinates 0.5 & 0.5 & Z is 0 w.r.t ViewportPoint Co-ordinates in 3D View & Storing this in ray Variable. Then, we do check if Ray hits Some Colliding Object within the distance of 5F which is in pickUpDistance Variable here & checks the layer which is named PickUpLayer here; Layer is almost similar to tag use to Identify the gameObject we're looking for; if this satisfies then we check the tag given to our Object here it's 'AmmoPick' given to Ammo bag Object. & Finally we Check Crucial Condition that If Our availableAmmo(PocketAmmo) Should not equals Ammo that's present in AmmoBox. If we've any Less amount of ammo that of our availableAmmo, then we're Ready To Pick Up Ammo from AmmoBox. If all these Conditions are True then only we Call the PickupAmmo Function to actually Pick Ammo from AmmoBox.

```
public void Update()
    {
        ray = mainCam.ViewportPointToRay(new Vector3(0.5f, 0.5f, 0));
        if(Physics.Raycast(ray, out hit, pickupDistance, layer))
        {
            if(hit.transform.tag == "AmmoPick")
            {
                if (shootscript.availableAmmo != shootscript.maxavailableAmmo)
                {
                    PickupAmmo();
                }
            }
        }
    }
```

Now here in PickupAmmo Function, we check if user has pressed Key ‘E’ if yes, then we dismisses that Object from Scene using Destroy KeyWord & then Ammo in our Pocket is Again reseted to the maximum capacity of our Pocket Ammo, here it’s 50(Initialized in Shoot Script).

```
// To Pick Ammo Bag.
    void PickupAmmo()
    {
        if (Input.GetKeyDown(KeyCode.E))
        {
            Destroy(hit.transform.gameObject);
            shootscript.availableAmmo = shootscript.maxavailableAmmo;
        }
    }
}
```

4.5 Gun Animations

We saw in Making Of Animations that How we Can make Different Animations for Different Objects. But, To Execute these Animations which are made we’ve to take help of Animator Transistor Window in Unity & some Sort Of Code.

4.5.1 Animator Transistor Window

After giving Animations to any Weapon then we’ve to decide also that when a particular Animation will be played & from which State. Here, Default Animation(from Entry/Start State) we set is Breathe. From that we can go to Fire as weel as Reload Animation depending on how much Transition Time we give to them to change from Current Animation State to Another One. We also have to set the Parameters here of type Bool, int, etc. which will be useful while writing Conditions for them in Code. These all things can be done by Animator Transistor Window.

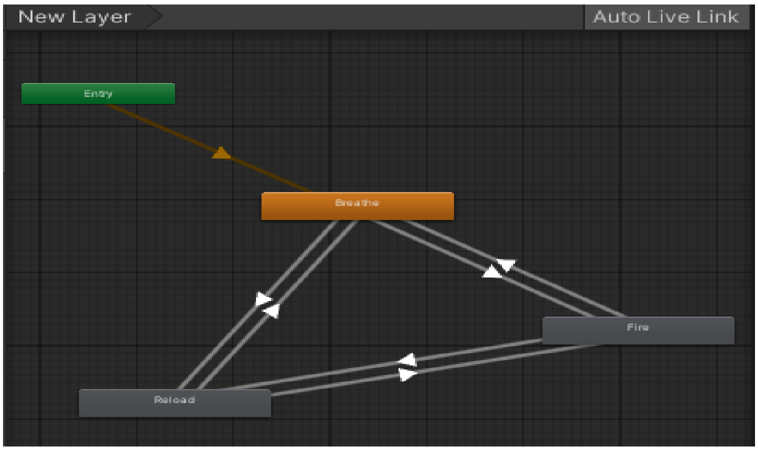


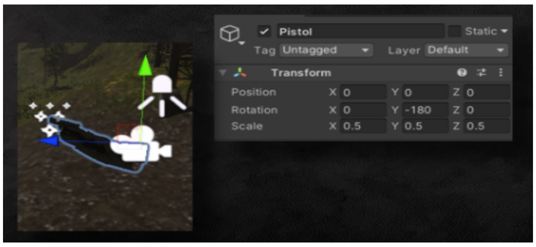
Fig 5: Animation of gun

4.5.2 Applying Conditions Using Code

After Doing Connections between all States & Setting Transition Time & Parameters we decide When Particular Animation is going to play in Game by applying Specific Conditions in Code. Like here, After Creating wepoman Class and anim Object of Animator we've Used If else Conditions that if R Key is Pressed Reload Animation will be played, for left click Fire Animation & Breathe if Player is Stable by accessing his position values on horizontal & Vertical Axis. While playing one Animation at a time all others are set to False as bool value here.

```
public class wepoman : MonoBehaviour
{
    public Animator anim;
    // Update is called once per frame
    void Update()
    {
        float x = Input.GetAxis("Horizontal");
        float z = Input.GetAxis("Vertical");
        if (x == 0 && z == 0)
        {
            anim.SetBool("Breathe", true);
            anim.SetBool("Fire", false);
            anim.SetBool("Reload", false);
        }
        if(Input.GetButton("Fire1"))
        {
            anim.SetBool("Breathe", false);
            anim.SetBool("Fire", true);
            anim.SetBool("Reload", false);
        }
        if(Input.GetKeyDown(KeyCode.R))
        {
            anim.SetBool("Breathe", false);
            anim.SetBool("Fire", false);
            anim.SetBool("Reload", true);
        }
          }
}
```

4.6 Positioning of Weapon



☑ Pistol □ Static

Tag Untagged Layer Default

Transform

	X	Y	Z
Position	0	0	0
Rotation	0	-180	0
Scale	0.5	0.5	0.5

Fig 6: Gun transformations

In unity, While Positioning any game object like here a Weapon, we've to use its World Coordinates & Transform them Accordingly. As this is a 3D View using 3D Transformations, we can Position object in any direction on plane by simply Changing its respective X, Y, Z Co-ordinate values. To Rotate it we just have to set Specific Degree for that Object respective to Specific Axis & Similarly we can Scale the Object as we want by Adjusting its Co-ordinate Values.

4.7 Making Gun Animations

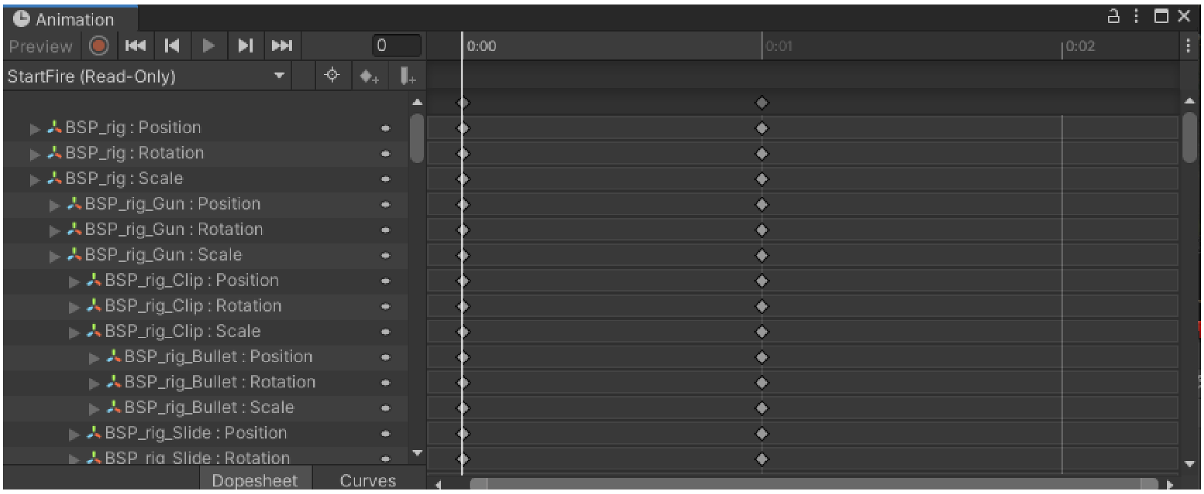


Fig 6.1: Animation window

We need to give Movements to many Objects in GamePlay Like Player, Gun, Zombie Should change their States, Positions While Running, Walking, Shooting, etc. So, here Animation Part Comes in picture.

Here in Unity, we simply have to select the Object to one which we want to Animate (here weapon) & have to go in Record Mode in Animation Window as shown above. Then, we've to give Movement Values Of that Object like Position Values, Rotation Values depending on how we want to move it. These are stored in KeyFrames, which are like diamond shaped shown in figure. KeyFrames are shot that defines the starting and ending points of any smooth transition.

5. Enemy Controller

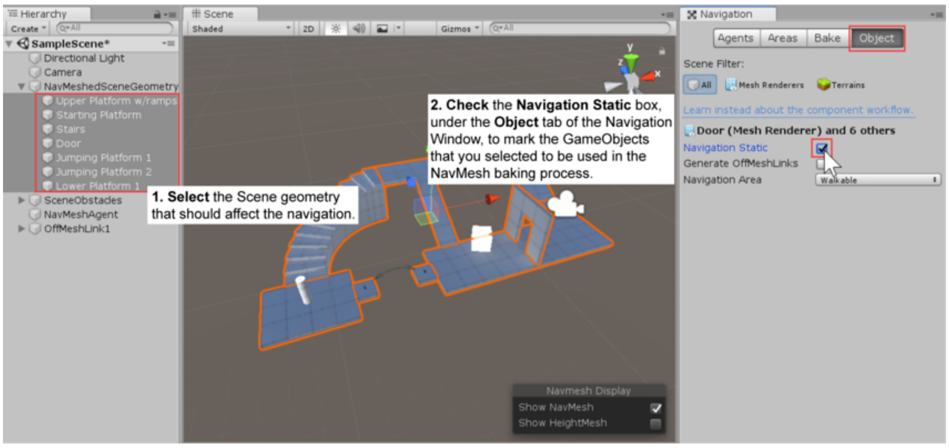
5.1 Building a NavMesh

The process of creating a **NavMesh** from the level geometry is called **NavMesh** Baking. The process collects the Render Meshes and **Terrains** of all Game Objects which are marked as Navigation Static, and then processes them to create a navigation **mesh** that approximates the walkable surfaces of the level.

In Unity, NavMesh generation is handled from the Navigation window
(menu: **Window** > **AI** > **Navigation**).

Building a NavMesh for your **scene**
can be done in 4 quick steps:

1. **Select** scene geometry that should affect the navigation – walkable surfaces and obstacles.
2. **Check Navigation Static** on to include selected objects in the NavMesh baking process.
3. **Adjust** the bake settings to match your agent size.
   - *Agent Radius* defines how close the agent center can get to a wall or a ledge.
   - *Agent Height* defines how low the spaces are that the agent can reach.
   - *Max Slope* defines how steep the ramps are that the agent walk up.
   - *Step Height* defines how high obstructions are that the agent can step on.
4. **Click bake** to build the NavMesh.

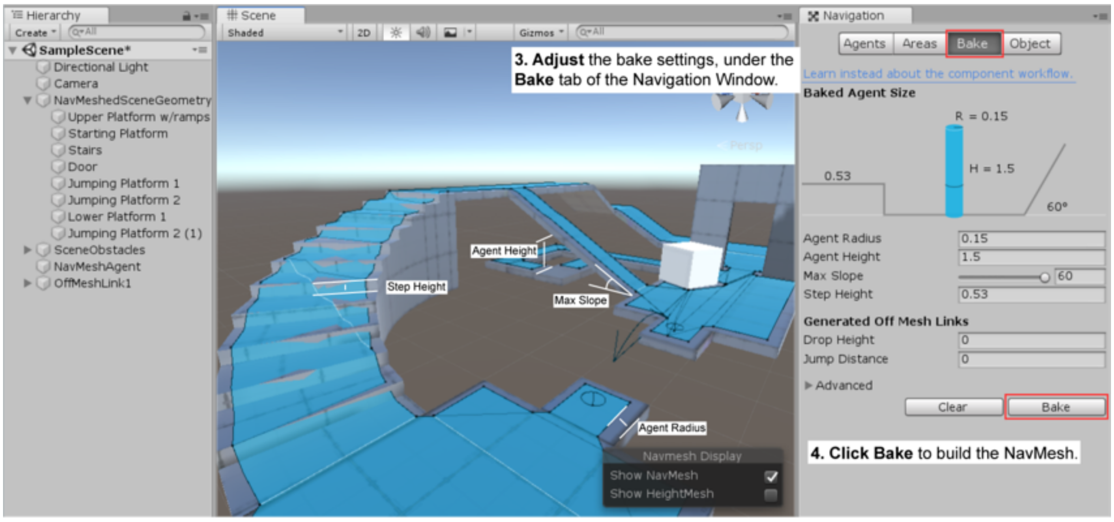


1. Select the Scene geometry that should affect the navigation.

2. Check the **Navigation Static** box, under the **Object** tab of the Navigation Window, to mark the GameObjects that you selected to be used in the NavMesh baking process.

Door (Mesh Renderer) and 6 others
Navigation Static ☑
Generate OffMeshLinks □
Navigation Area Walkable

Fig 7: Navigation window



3. Adjust the bake settings, under the **Bake** tab of the Navigation Window.

Baked Agent Size
Agent Radius 0.15
Agent Height 1.5
Max Slope 60
Step Height 0.53
Generated Off Mesh Links
Drop Height 0
Jump Distance 0

4. Click Bake to build the NavMesh.

Fig 8: Nav Mesh baking

The resulting NavMesh will be shown in the scene as a blue overlay on the underlying level geometry whenever the Navigation Window is open and visible.

As you may have noticed in the above pictures, the walkable area in the generated NavMesh appears shrunk. The NavMesh represents the area where the center of the agent can move. Conceptually, it doesn’t matter whether you regard the agent as a point on a shrunken NavMesh or a circle on a full-size NavMesh since the two are equivalent. However, the point interpretation allows for better runtime efficiency and also allows the designer to see immediately whether an agent can squeeze through gaps without worrying about its radius.

Another thing to keep in mind is that the NavMesh is an approximation of the walkable surface. This can be seen for example in the stairs which are represented as a flat surface, while the source surface has steps. This is done in order to keep the NavMesh data size small. The side effect of the approximation is that sometimes you will need to have a little extra space in your level geometry to allows the agent to pass through a tight spot.

When baking is complete, you will find a NavMesh asset file inside a folder with the same name as the scene the NavMesh belongs to. For example, if you have a scene called *First Level* in the Assets folder, the NavMesh will be at *Assets > First Level > NavMesh.asset*.

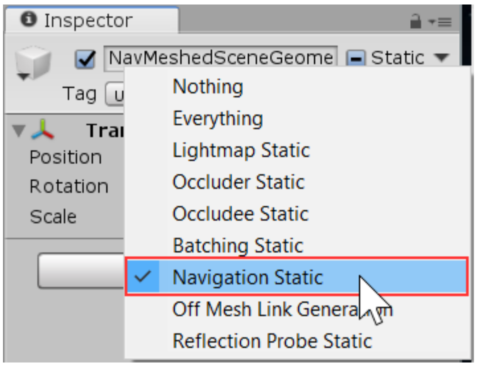
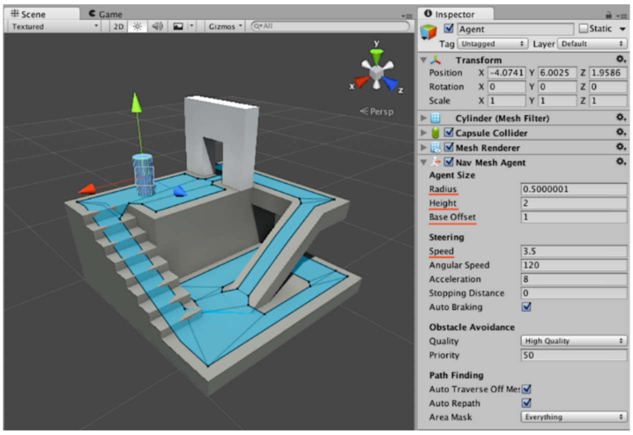


Fig 9: Navigation Inspector

5.2 Creating a NavMesh Agent

Once you have a **NavMesh** baked for your level it is time to create a character which can navigate the **scene**. We're going to build our prototype agent from a cylinder and set it in motion. This is done using a **NavMesh** Agent component and a simple script.



☑ Agent ☐ Static
Tag Untagged Layer Default

Transform
Position X -4.0741 Y 6.0025 Z 1.9586
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Cylinder (Mesh Filter)
☑ Capsule Collider
☑ Mesh Renderer
☑ Nav Mesh Agent

Agent Size
Radius 0.5000001
Height 2
Base Offset 1

Steering
Speed 3.5
Angular Speed 120
Acceleration 8
Stopping Distance 0
Auto Braking ☑

Obstacle Avoidance
Quality High Quality
Priority 50

Path Finding
Auto Traverse Off Me: ☑
Auto Repath ☑
Area Mask Everything

Fig 10: Navigation Mesh Agent

First let's create the character:

1. Create a **cylinder**: **GameObject > 3D Object > Cylinder**.

2. The default cylinder dimensions (height 2 and radius 0.5) are good for a humanoid shaped agent, so we will leave them as they are.
3. Add a **NavMesh Agent** component: **Component > Navigation > NavMesh Agent**.

Now you have simple NavMesh Agent set up ready to receive commands!

When you start to experiment with a NavMesh Agent, you most likely are going to adjust its dimensions for your character size and speed.

The **NavMesh Agent** component handles both the pathfinding and the movement control of a character. In your **scripts**
, navigation can be as simple as setting the desired destination point – the NavMesh Agent can handle everything from there on.

```
Using UnityEngine;
   using UnityEngine.AI;

   public class MoveTo : MonoBehaviour {

      public Transform goal;

      void Start () {
         NavMeshAgent agent = GetComponent<NavMeshAgent>();
         agent.destination = goal.position;
      }
   }
```

Next we need to build a simple script which allows you to send your character to the destination specified by another Game Object, and a Sphere which will be the destination to move to:

1. Create a new **C# script** (MoveTo.cs) and replace its contents with the above script.
2. Assign the MoveTo script to the character you've just created.
3. Create a **sphere**, this will be the destination the agent will move to.
4. Move the sphere away from the character to a location that is close to the NavMesh surface.
5. Select the character, locate the MoveTo script, and assign the Sphere to the **Goal** property.
6. **Press Play**; you should see the agent navigating to the location of the sphere.

To sum it up, in your script, you will need to get a reference to the NavMesh Agent component and then to set the agent in motion, you just need to assign a position to its destination property.
The Navigation How Tos will give you further examples on how to solve common gameplay scenarios with the NavMesh Agent.

5.3 Zombie Animations

Similar to gun animation, To Execute these Animations which are made we've to take help of Animator Transistor Window in Unity & some Sort Of Code.

5.3.1 Animator Transistor Window

After giving Animations to a zombie then we've to decide also that when a particular Animation will be played & from which State. Here, Default Animation(from Entry/Start State) we set is Z_Idle (idle animation). From that we can go to dormant_walk (walk animation) and form there to Z_Run_InPlace (run animation) as well as to Z_Attack (attack animation) and Z_FallingBack (fall animation) depending on how much Transition Time we give to them to change from Current Animation State to Another One. We also have to set the Parameters here of type Bool, int, etc. which will be useful while writing Conditions for them in Code. Parameter to play dormant_walk animation is d_walk, Z_Run_InPlace animation is run, Z_Attack animation is attack and Z FallingBack animation is dead. These parameters are of bool type so in a parameter is set true its respective animation is played. These all things can be done by Animator Transistor Window.

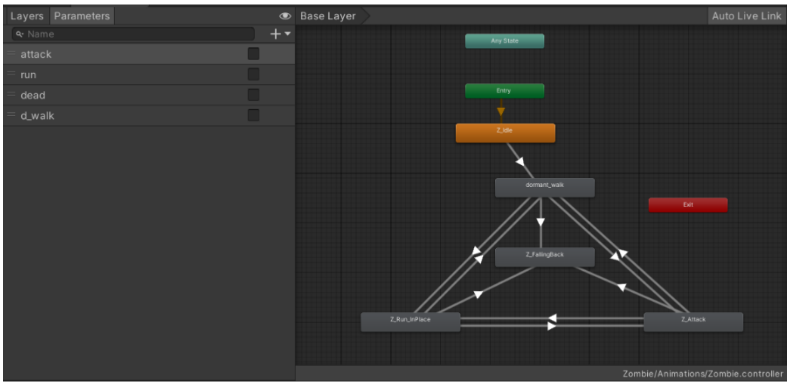


Fig 11: Zombie animator

5.3.2 Applying Conditions Using Code

After Doing Connections between all States & Setting Transition Time & Parameters we decide When Particular Animation is going to play in Game by applying Specific Conditions in Code. Like here, After Creating enemycontroller Class and anim Object of Animator then in Update method we've Used If Conditions that dist<= attackingDistance (in distance between zombie and played is less than equal to attackingDistance(50m)) run is set false, attack is set true and d_walk is set false and Z_Attack (attack animation) Animation will be played. Parameter is set using object_of_animator.SetBool("parameter",true/false) function, for example anim.SetBool("attack",true) sets attack parameter to true.

```
public class enemycontroller : MonoBehaviour
{
public HealthBar healthbar;
    damage dmg;
    NavMeshAgent agent;
    public float attackingDistance = 4f;
    public float idleDistance = 10f;
    GameObject target;
    public bool spottedTarget;
    public Animator anim;
    public float maxDistance;
private void Update()
{
        float dist = Vector3.Distance(transform.position, target.transform.position);
        if (dist <= attackingDistance)
        {
            anim.SetBool ("run", false);
            stopEnemy();
            anim.SetBool ("attack", true );
        anim.SetBool ("d_walk", false);
        healthbar.damageTaken (0.0125f);
        spottedTarget = true;

    }
```

6. UI elements

Unity UI is a UI toolkit for developing user interfaces for games and applications. It is a Game Object-based UI system that uses Components and the Game View to arrange, position, and style user interfaces. You cannot use Unity UI to create or change user interfaces in the Unity Editor.

6.1 Visual Tree

The visual tree holds all the visual elements in a window. It is an object graph made of lightweight nodes refered to as *visual elements*.

These nodes are allocated on the C# heap, either manually or by loading UXML assets from a UXML template file.

Each node contains the layout information, its drawing and redrawing options, and how the node responds to events.

The root object of the visual tree is referred to as the panel. A new element is ignored until it is connected to the panel. You can add elements to an existing element to attach your user interface to the panel.

6.2 Position, transforms, and coordinate systems

The different coordinate systems are defined as follow:
World: Coordinates are relative to the panel space. The panel is the highest element in the visual tree.
Local: Coordinates are relative to the element itself.

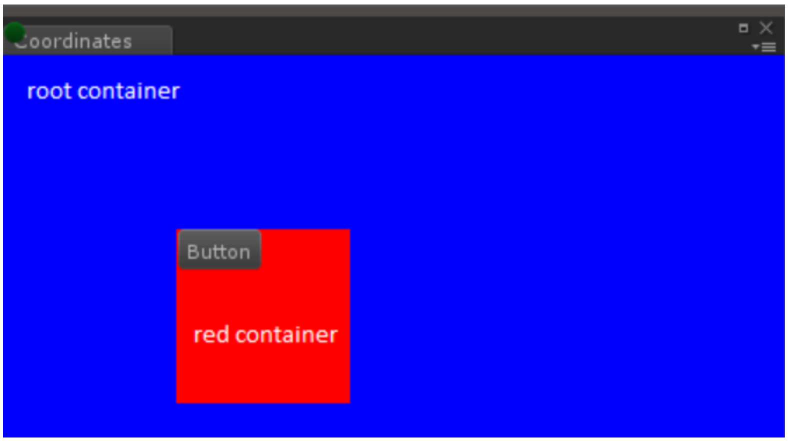


Fig 12: Panel hierarchy

6.3 Panel

- Tab section (refered to as DockArea and labelled “Coordinates”)
- Blue VisualElement acts as the root (refered to as “rootVisualContainer”)
    - Red VisualElement acts as a parent of the button (“red container”)
        - Button

6.4 The layout Engine

IElements includes a layout engine that positions visual elements based on layout and styling properties. The layout engine is the Yoga open source project that implements a subset of *Flexbox*: a HTML/CSS layout system.

To get started with Yoga and Flexbox, consult the following external resources:

- Yoga official documentation: the mapping of properties is almost 1-to–1
- CSS-Tricks guide to Flexbox: most properties are supported with some minor differences

6.5 The UXML format

UXML files are text files that define the logical structure of the user interface. The format used in UXML files is inspired by HTML (HyperText Markup Language), XAML (eXtensible Application Markup Language), and XML (eXtensible Markup Language). If you are familiar with these recognized formats, you should notice lots of similarities in UXML. However, the UXML format includes small differences to provide an efficient way to work with Unity.

6.6 UQuery

UQuery provides a set of extension methods for retrieving elements from any UIElements visual tree. UQuery is based on JQuery or Linq, but UQuery is designed to limit dynamic memory allocation as much as possible. This allows for optimal performance on mobile platforms.

6.7 Styles and Unity style sheets

Each VisualElement includes style properties that set the dimensions of the element and how the element is drawn on screen, such as backgroundColor or borderColor.

Style properties are either set in C# or from a style sheet. Style properties are regrouped in their own data structure (IStyle interface).

UIElements supports style sheets written in USS (Unity style sheet). USS files are text files inspired by Cascading Style Sheets (CSS) from HTML. The USS format is similar to CSS, but USS includes overrides and customizations to work better with Unity.

This section includes details on USS, its syntax, and its differences when compared to CSS.

6.8 The event system

UIElements includes an event system
that communicates user interactions to visual elements. Inspired by HTML events, the UIElements events system shares many of the same terminology and event naming. The **UlElement event system is comprised of the following:**

1. **Event dispatcher:** UIElements listens to events, coming from the operating system or scripts
, and dispatches these events with the Event dispatcher. The Event dispatcher also determines the dispatching strategy used to send events to visual elements and other supporting classes.
2. **Event handler:** When an event occurs inside a panel, the event is sent to the VisualElement tree within the panel. You can add event handlers to visual

elements to respond to certain event types when they occur. See Responding to events.

3. **Event synthesizer:** The operating system is not the only source of events. Scripts can also create and dispatch events through the dispatcher. See Synthesizing Events for more on creating and dispatching events.
4. **Event types:** The different event types are organized into a hierarchy based on EventBase and grouped into families. Each family of events implements an interface that defines the common characteristics for all events of the same family. For example, MouseUpEvent, MouseDownEvent and other mouse events implement the IMouseEvent inteface. This interface specifies that each mouse event has a position, a pressed button, a set of modifers, and other mouse-related event types. See Event type reference for a description of each event family and their UIElement event types.

You can also use events to communicate other types of messages to visual elements. For example, the ContextualMenuManager uses the ContextualMenuPopulateEvent to add items to a contextual menu. See built-in controls.

6.9 Health Bar & Stamina Bar

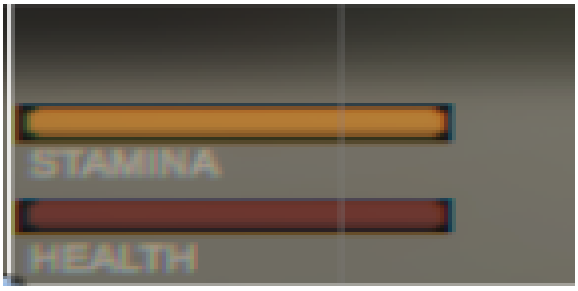


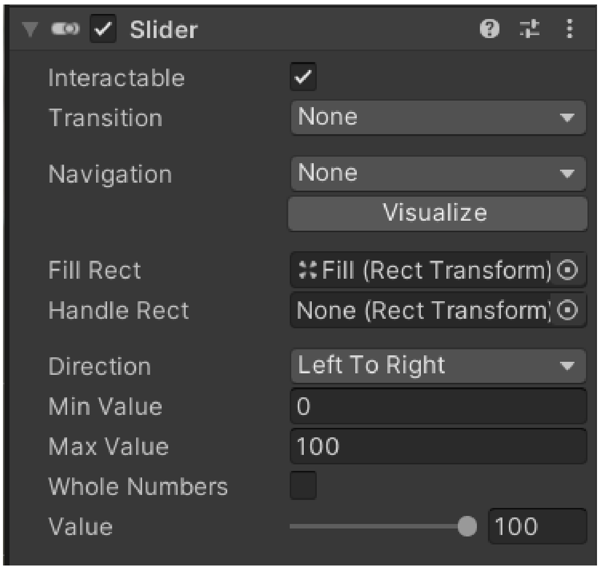
Fig 13: Stamina Bar and health bar

This is how the staminabar and healthbar look on the canvas.

These are sliders. The value of a Slider is determined by the position of the handle along its length.

The value increases from the *Min Value* up to the *Max Value* in proportion to the distance the handle is dragged.

The default behaviour is for the slider to increase from left to right but it is also possible to reverse this behaviour using the *Direction* property. You can also set the slider to increase vertically by selecting *Bottom to Top* or *Top to Bottom* for the *Direction* property.



Slider ☑

Interactable ☑
Transition: None
Navigation: None
Visualize
Fill Rect: Fill (Rect Transform)
Handle Rect: None (Rect Transform)
Direction: Left To Right
Min Value: 0
Max Value: 100
Whole Numbers ☐
Value: 100

Fig 14: Slider Inspector

The slider has a single event called On Value Changed that responds as the user drags the handle. The current numeric value of the slider is passed to the function as a float parameter

slider.maxValue sets the max value of the slider to given value.

```
slider.maxValue = maxHealth;
```

slider.value sets the current value of slider to the given value.

```
slider.value = health;
```

6.10 Updating UI Elements

To update the ammo count and health pick up count on canvas we use text boxes of type **rich text**.

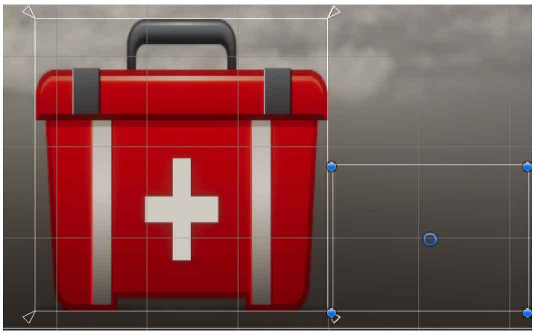


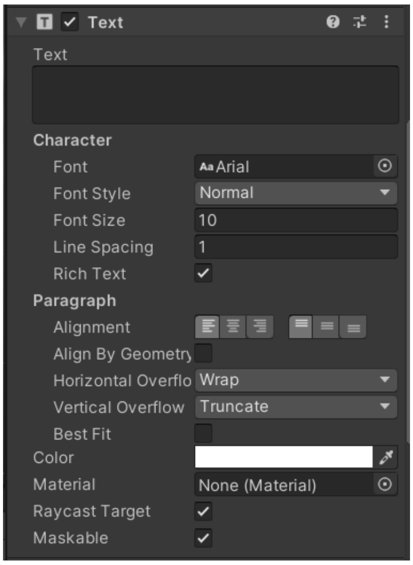
Fig 15: medical pack

As u can see in the above image the box besides the medkit icon is a text box of type **rich text**.

The text for UI elements and text meshes can incorporate multiple font styles and sizes.

Rich text is supported both for the UI System and the legacy GUI system.

The Text, GUIStyle, GUIText and TextMesh classes have a **Rich Text** setting which instructs Unity to look for markup tags within the text



☑ Text

Text

Character

Font: Arial

Font Style: Normal

Font Size: 10

Line Spacing: 1

Rich Text: ☑

Paragraph

Alignment

Align By Geometry: □

Horizontal Overflo: Wrap

Vertical Overflow: Truncate

Best Fit: □

Color

Material: None (Material)

Raycast Target: ☑

Maskable: ☑

Fig 16: Text Box Inspector

Firstly, we create objects of type text box

```
public Text PistolAmmoDisplay;
public Text RifleAmmoDisplay;
public Text AmmoCount;
```

Now to set text in these text boxes we use **TextBox.text** function which is a function of Rich Text Box sets the text in the box to the string assigned.

```
PistolAmmoDisplay.text = "x" + pistolAmmoCount;
RifleAmmoDisplay.text = "x" + rifleAmmoCount;
AmmoCount.text = "x" + availableAmmo;
```

6.11 Mini Map

Now for easy navigation through the map an indicator is required, and hence we require min map.

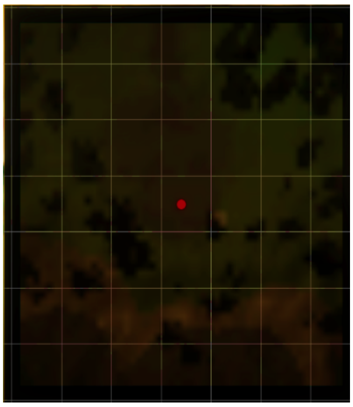


Fig 17: Mini map

For creating Mini Map we created a camera that is at the top of the player.

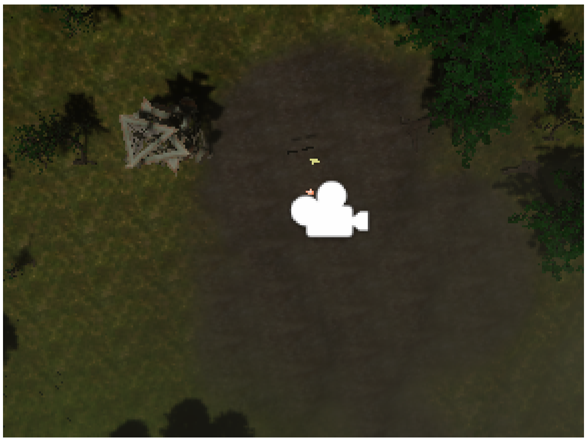
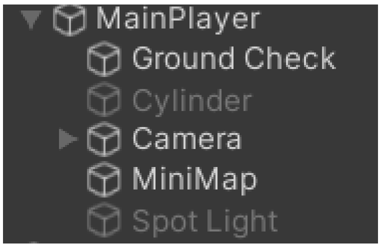


Fig 18: Camera from top view to follow player and navigate on mini map

As u can see in the above image there is camera icon that is the minimap camera, and the bright shiny dot below it is our player.

To make it foll0w the player we make it the child of the main player.

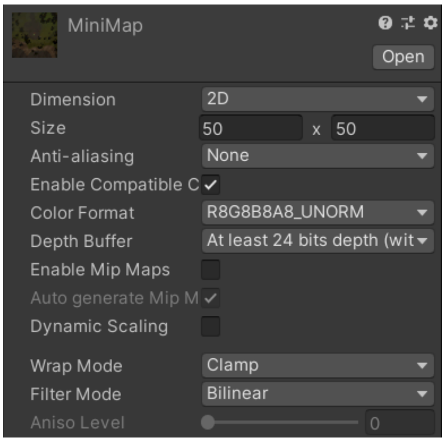


MainPlayer
Ground Check
Cylinder
Camera
MiniMap
Spot Light

Fig 19: hierarchy of mini map in camera

Then we want the output of the camera as 2D and not 3D so we set it to orthographic mode.

Now on the canvas we create a render texture box. A Render Texture is a type of Texture that Unity creates and updates at run time.



MiniMap

Dimension 2D
Size 50 x 50
Anti-aliasing None
Enable Compatible C ✓
Color Format R8G8B8A8_UNORM
Depth Buffer At least 24 bits depth (wit
Enable Mip Maps □
Auto generate Mip M ✓
Dynamic Scaling □
Wrap Mode Clamp
Filter Mode Bilinear
Aniso Level 0

Fig 20: camera view as texture to text box

Now we set the output texture of camera to that of render texture.

7. IEnumerator

When you iterate through a collection or access a large file, waiting for the whole action would stop all others, IEnumerator allows to stop the process at a specific moment, return that part of object (or nothing) and gets back to that point whenever you need it.

This until the MoveNext method from IEnumerator returns false indicating we reached the end of the collection/file.

The coroutine is the object within unity that allows to start parallel action (or almost). When using StartCoroutine, unity creates a new object of type Coroutine, this object performs some action and then returns a IEnumerator object (or nothing)

```
IEnumerator Reload()
{
        isFiring = false;
        isReloading = true;
        yield return new WaitForSeconds(reloadTime);
        while (availableAmmo > 0 && ( pistolAmmoCount <= pistolCapacity || rifleAmmoCount
        <= rifleCapacity))
        {
                // code for reloading
}
```

It returns a null object or a **IEnumerator**. A coroutine needs a MonoBehaviour to be attached to so it will use it as a parent process (if familiar with Unix). That parent process checks on each frame if it has a coroutine pending, it does at various time in the frame since you can control when you want the coroutine to get back on.

8. Atmosphere

8.1 SkyBoxes are a wrapper around your entire scene that shows what the world looks like beyond your geometry

Skyboxes are rendered around the whole scene in order to give the impression of complex scenery at the horizon. Internally skyboxes are rendered after all opaque objects; and the mesh used to render them is either a box with six textures, or a tessellated sphere.

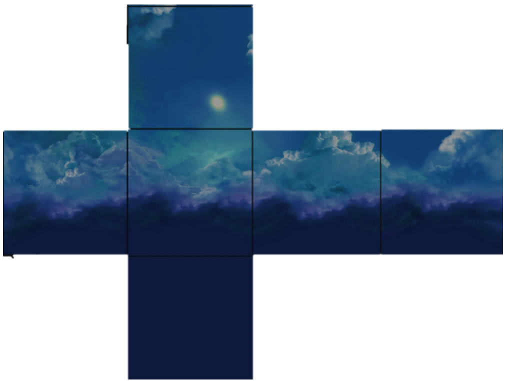


Fig 21: sky box opened

8.2 Fog is the effect of overlaying a color onto objects dependant on the distance from the camera. This is used to simulate fog or mist in outdoor environments and is also typically used to hide clipping of objects when a camera's far clip plane has been moved forward for performance.

The Fog effect creates a screen-space fog based on the camera's depth texture. It supports Linear, Exponential and Exponential Squared fog types. Fog settings should be set in the **Scene** tab of the **Lighting** window.

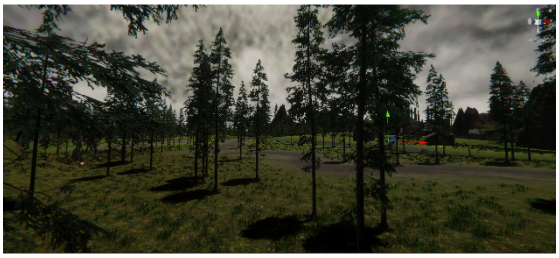


Fig 22.1: Prior post processed fog

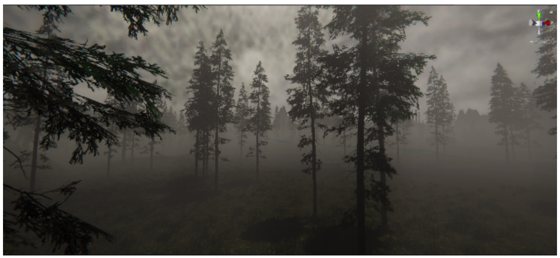


Fig 22.2: After post processed fog

8.3 Post processing: Ambient Occlusion, Anti-aliasing, Auto Exposure, Bloom, Color Adjustments, Motion Blur, Panini Projection, Shadows Midtones Highlights, Vignette

Unity provides a number of **post-processing** effects and full-screen effects that can greatly improve the appearance of your application with little set-up time. You can use these effects to simulate physical **camera** and film properties, or to create stylised visuals.

Effect Name	Description
Ambient Occlusion	The Ambient Occlusion effect darkens the [illegible] that
Anti-aliasing	The Anti-aliasing effect softens the appearance of edges in your scene
Auto Exposure	The Auto Exposure effect dynamically adjusts the exposure of
Bloom	The Bloom effect makes bright areas in your image glow. Note that Bloom
Channel Mixer	The Channel mixer lets you adjust the balance of each input color
Chromatic Aberration	The Chromatic Aberration effect disperses colors along the
Color Adjustments	The Color Adjustments effect lets you change the overall tone
Color Curves	The Color Curves effect lets you adjust specific ranges in hue
Fog	The Fog effect simulates the look of fog or mist in outdoor environments
Depth of Field	The Depth of Field effect blurs the background of your image while the
Grain	The Grain effect overlays film noise onto your image
Lens Distortion	The Lens Distortion effect simulates distortion caused by the shape of a
Lift, Gamma, Gain	The Lift, Gamma, Gain effect allows you to perform three-way color
Motion Blur	The Motion Blur effect blurs the image in the direction of
Panini Projection	The Panini Projection effect corrects distortion at the edge of an image
Screen Space Reflection	The Screen Space Reflection effect creates subtle reflections that
Shadows Midtones Highlights	The Shadows Midtones Highlights effect separately controls
Split Toning	The Split Toning effect maps two different tones in your image to two
Tonemapping	The Tonemapping effect remaps the values of an image to high dynamic
Vignette	The Vignette effect darkens the edges of an image
White Balance	The White Balance effect preserves the white areas in your image and

Table 1: Post processing properties

9. Light

9.1 Point light A point light is located at a point in space and sends light out in all directions equally. The direction of light hitting a surface is the line from the point of contact back to the center of the light object. The intensity diminishes with distance from the light, reaching zero at a specified range. Light intensity is inversely proportional to the square of the distance from the source. This is known as ‘inverse square law’ and is similar to how light behaves in the real world.

We have used it for the purpose of indoor lightings to simulate bulbs.

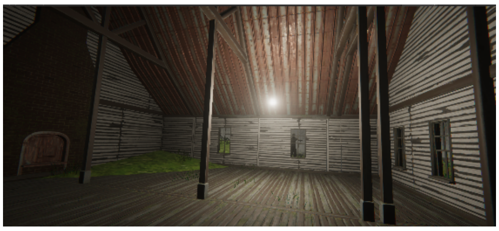


Fig 23: bulb simulation with point light

9.2 Spotlight Like a point light, a spot light has a specified location and range over which the light falls off. However, the spot light is constrained to an angle, resulting in a cone-shaped region of illumination. The center of the cone points in the forward (Z) direction of the light object. Light also diminishes at the edges of the spot light's cone. Widening the angle increases the width of the cone and with it increases the size of this fade, known as the 'penumbra'.

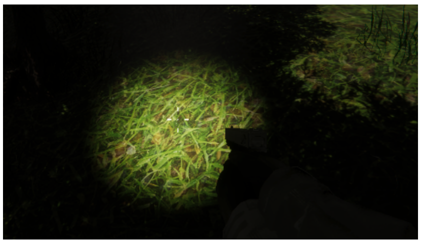


Fig 24: Torch simulation with spotlight

9.3 Directional lights are very useful for creating effects such as sunlight in your scenes. Behaving in many ways like the sun, directional lights can be thought of as distant light sources which exist infinitely far away. A directional light does not have any identifiable source position and so the light object can be placed anywhere in the scene. All objects in the scene are illuminated as if the light is always from the same direction. The distance of the light from the target object is not defined and so the light does not diminish.

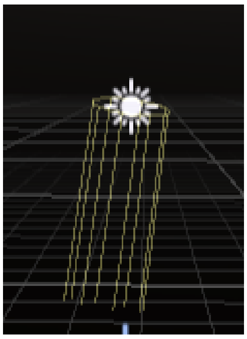


Fig 25: Directional light

10. Particle system

A **particle system** simulates and renders many small images or Meshes, called particles, to produce a visual effect. Each particle in a system represents an individual graphical element in the effect. The system simulates every particle collectively to create the impression of the complete effect. When a GameObject with a Particle System is selected, the **Scene view** contains a small **Particle Effect** panel, with some simple controls that are useful for visualising changes you make to the system's settings.
The Particle System Curves editor has the following buttons:

Optimize: Fits the curve into four or fewer keys to build a fast evaluator called a Polynomial, which is more efficient than reading the unoptimized curve.

Remove: Deletes the selected curve.
To edit the way in which the Particle System plays curves, click the cog next to a selected key and choose one of the following options:

Loop: Plays the curve the specified number times over a particle's life. For example, if you make a curve that scales a particle's size up and down, you can tell it to loop multiple times, which causes the "up and down" animation to happen multiple times before the particle dies, instead of just once.

Ping Pong
: Similar to **Loop**, but plays the curve forwards then backwards in a continuous oscillation.

Clamp: Limits particle queries that fall outside the curve time range to the first or last value of the curve.

The **Start Color** property in the main module has the following options:

Color: All particles start with this color throughout the lifetime of the Particle System. Particles can still change color during their lifetime.

Gradient: The Particle System emits particles which start with the colour at the beginning of the gradient, and end at the colour at the end of the gradient. The gradient line represents the lifetime of the Particle System; the Particle System picks a color from the gradient at the point corresponding to the current age of the Particle System.

Random Between Two Colors: The Particle System chooses a starting particle color from a random linear interpolation between the two given colors.

Random Between Two Gradients: The Particle System chooses a color from each of the given gradients at the point corresponding to the current age of the system. The starting particle color is chosen as a random linear interpolation between the two chosen colors.

Random Color: Similar to **Gradient** mode, where particles take their initial color from the defined **Gradient**. However, in this mode, the Particle System does not choose samples based on the age of the Particle System, but instead it selects them at random. This mode also works well with the **Fixed Gradient Mode**, which is inside the Gradient Editor. When enabled, you can select a predefined list of precise starting colors, and apply a probability to each color.

Other color properties, such as **Color over Lifetime**, can use the **Gradient** or **Random Between Two Gradients** modes.

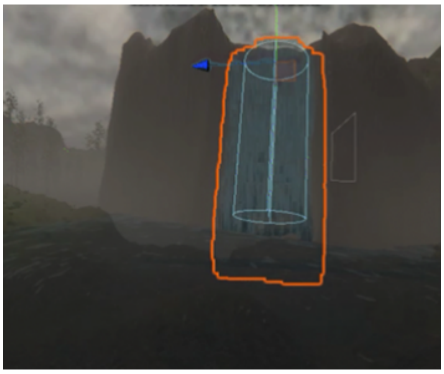
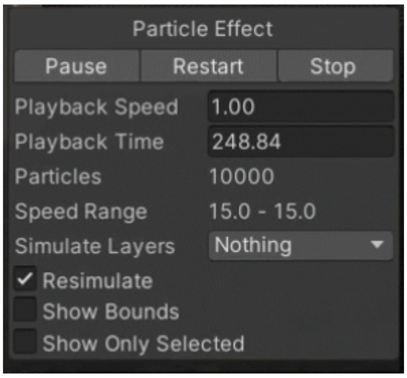


Fig 26: waterfall with particle systems



Particle Effect

Playback Speed 1.00
Playback Time 248.84
Particles 10000
Speed Range 15.0 - 15.0
Simulate Layers Nothing
☑ Resimulate
□ Show Bounds
□ Show Only Selected

Fig 27: Particle effect

11. Terrain and Map

The map is a meadow surrounded by mountains creating a green valley with a river and a waterfall, there is also a small cottage town uphill.
The terrain was built using different sized brushes on Unity.

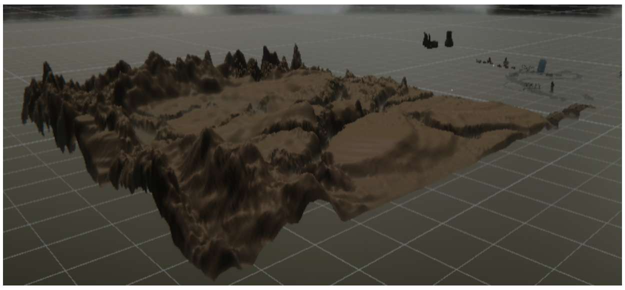
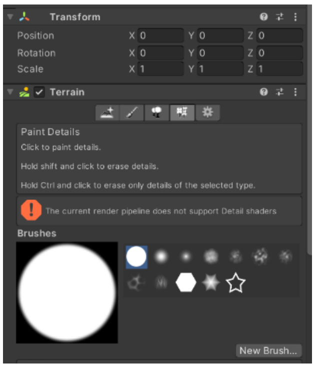


Fig 28:Terrain sculpting



Transform

Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

☑ Terrain

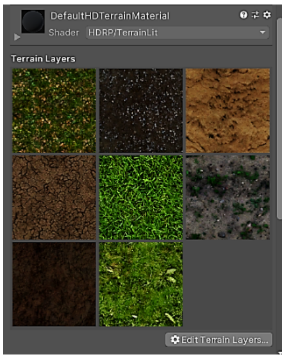
Paint Details
Click to paint details.
Hold shift and click to erase details.
Hold Ctrl and click to erase only details of the selected type.

The current render pipeline does not support Detail shaders

Brushes

Fig 29:Terrain brushes

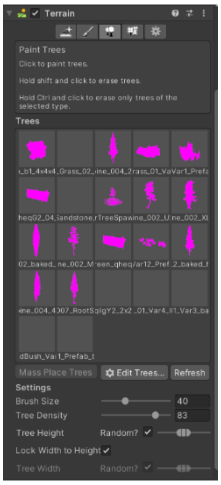
To create a Terrain Layer directly from the Terrain Inspector, click the paintbrush icon in the **toolbar**
at the top of the Terrain Inspector, and select **Paint Texture** from the drop-down menu. At the bottom of the **Terrain Layers** section, click the **Edit Terrain Layers** button, and choose **Create Layer**.
A **Terrain** might have grass clumps and other small objects (such as rocks) covering its surface. Unity renders these objects using textured **quads**
or full Meshes, depending on the **level of detail**
and performance you require.



DefaultHDTerrainMaterial
Shader HDRP/TerrainLit

Terrain Layers

Fig 30: terrain layers



☑ Terrain

Paint Trees
Click to paint trees.
Hold shift and click to erase trees.
Hold Ctrl and click to erase only trees of the selected type.

Trees

Settings
Brush Size 40
Tree Density 83
Tree Height Random? ✓
Lock Width to Height ✓
Tree Width Random? ✓

Fig 31: trees prefab

Fig 32: Stones prefab

12. 3D objects and models:

9.1 Meshes are the main graphics primitive of Unity. They define the shape of an object.

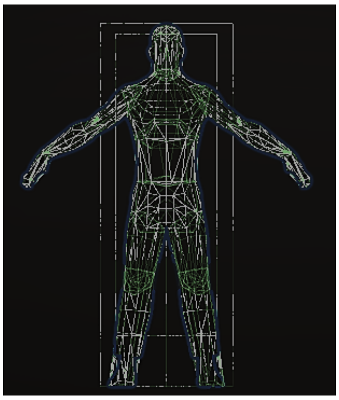


Fig 33: Mesh

9.2 Materials define how a surface should be rendered, by including references to the Textures it uses, tiling information, Color tints and more. The available options for a Material depend on which Shader the Material is using.

Fig 34: Material

9.3 Shaders are small **scripts** that contain the mathematical calculations and algorithms for calculating the Color of each **pixel** rendered, based on the lighting input and the Material configuration.

Fig 35: Shader Script

9.4 Textures are bitmap images. A Material can contain references to textures, so that the Material's Shader can use the textures while calculating the surface color of a **GameObject** In addition to basic Color (Albedo) of a GameObject's surface, Textures can represent many other aspects of a Material's surface such as its reflectivity or roughness.

13. Rendering

Rendering or image synthesis is the process of generating a photorealistic or non-photorealistic image from a 2D or 3D model by means of a computer program.
Rendering pipeline: s a conceptual model that describes what steps a graphics system needs to perform to render a 3D scene to a 2D screen.

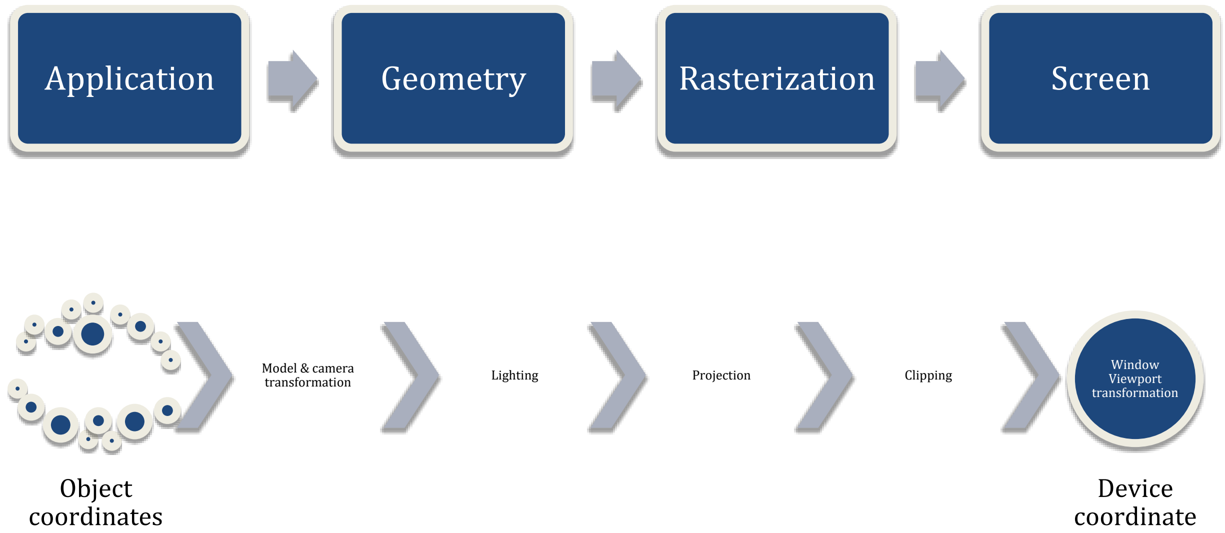


Fig 36: Rendering Pipeline

We have used the High-definition Render pipeline, Use HDRP for AAA quality games, automotive demos, architectural applications and anything that requires high-fidelity graphics. HDRP uses physically-based lighting and materials, and supports both forward and deferred **rendering** HDRP uses compute **shader** technology and therefore requires compatible GPU hardware.

Forward	HDRP calculates the lighting in a single pass when rendering each individual Material.
Deferred	HDRP renders all GameObjects into a GBuffer that stores the Material properties that are visible on the screen. HDRP then processes the lighting for every GameObject in the Scene.

References

[1] Unity learn- Beginner scripting https://learn.unity.com/project/beginner-gameplay-scripting
[2] Unity learn –FPS microgame template https://learn.unity.com/project/fps-template
[3] YouTube – Brackeys https://www.youtube.com/user/Brackeys
[4] Introduction to game development with unity GeeksForGeeks https://www.geeksforgeeks.org/game-development-with-unity-introduction
[5] Concept ideas from various zombie games like Left4dead, Into the dead and The Last of Us.
[6] Unity User Manual (2019.4 LTS) https://docs.unity3d.com/Manual/index.html