iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 11 **Issue:** V **Month of publication:** May 2023

DOI: https://doi.org/10.22214/ijraset.2023.51039

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 11 Issue V May 2023- Available at www.ijraset.com

3D First-Person Shooter Game Development Using Unreal Engine 5

Nihar Raut[1], Sanket Chopade[2], Tejas Nichat[3], Himanshu Kuhad[4], Shreeniwas Bangade[5], Prof. Devchand Choudhari[6]
[1, 2, 3, 4, 5]*Department of Computer Science and Engineering, Government College of Engineering Chandrapur, Maharashtra, India*
[6]*Asst. Prof. Department of Computer Science and Engineering, Government College of Engineering Chandrapur, Maharashtra, India*

***Abstract:** This research paper will explore the development of a 3D first-person shooter game using Unreal Engine 5. The paper will cover the basics of game development using Unreal Engine 5 and how it can be used to create a 3D first-person shooter game. Unreal Engine has changed significantly over the years and thus this paper will also explore the new features and techniques introduced in recent years. Additionally, the paper will discuss the challenges faced during the game development process and how they were overcome.*

I. INTRODUCTION

The video game industry has been growing rapidly over the past few years, with an estimated global market value of over $200 billion in 2022. As a result, game development has become an increasingly popular career path, with many aspiring developers looking to create their games.

One popular genre of games is 3D games. These games immerse the player in a virtual world, allowing them to explore, interact with objects and characters, and engage in combat. With the recent release of Unreal Engine 5, game developers have access to a powerful tool for creating 3D games. This research paper aims to provide a comprehensive overview of the process of developing a 3D first-person shooter game using Unreal Engine 5. The paper will cover the various stages of game development, including the design phase, implementation, testing, and optimization. It will also provide a detailed explanation of the tools and features available in Unreal Engine 5 that are relevant to 3D first-person shooter game development.

By the end of this paper, readers will have a comprehensive understanding of the process of developing a 3D first-person shooter game using Unreal Engine 5, as well as an appreciation for the design principles that make these games successful. This information will be useful to aspiring game developers, students, and professionals in the video game industry.

II. DESIGN

In the design phase, it is crucial to create a comprehensive plan that outlines the game's concept, gameplay mechanics, and key features. By defining the plan before moving into full development, the design team can avoid confusion and streamline the development process. Some of the steps involved in creating a plan for the game include defining the game concept, creating a game prototype, developing the game world, designing player abilities and weapons, creating level designs, designing the user interface, and testing and iterating on the design. Through this planning process, developers can ensure that the final product meets the intended vision and provides a compelling gameplay experience.

1) Define the game concept: The first step in the design phase is to define the game concept, including the story, setting, and game mechanics.
2) Create a game prototype: Once the game concept is defined, create a game prototype that demonstrates the core gameplay mechanics. This allows developers to test and refine the gameplay before committing to a full development cycle.
3) Develop the game world: The game world is the setting in which the game takes place. This includes designing the environment, including landscapes, buildings, and other objects, as well as designing non-playable characters (NPCs) and enemies.
4) Define player abilities and weapons: Player abilities and weapons are key elements of a first-person shooter game. These should be designed to be engaging and balanced, offering players a range of options for engaging with the game world.
5) Create level designs: Levels are the individual stages or sections of the game that players progress through. Level designs should be carefully crafted to provide a challenging and rewarding experience, with a balance of exploration, combat, and puzzles.

6) Design the user interface (UI): The UI is how players interact with the game world, including menus, HUD elements, and other visual feedback. The UI should be designed to be intuitive and easy to use, while also providing players with the information they need to make decisions in the game.
7) Test and iterate: Throughout the design phase, it is important to test the game and gather feedback from players. This feedback should be used to iterate on the design, adjusting as needed to improve the gameplay experience.

In the design phase of creating our game using Unreal Engine 5, we determined that our game would be a fast-paced shooter with simple gameplay mechanics, featuring three types of weapons and a single map. We decided to create an open-world game, which would allow the player to move freely throughout the world without getting stuck in quests. We chose this approach because we had time and resource restrictions and wanted to keep the game relatively simple. By focusing on creating a single map and allowing for open-world exploration, we could ensure that the game remained engaging while still being achievable within our timeframe. Additionally, we believed that this approach would provide players with a sense of freedom and immersion within the game world.

III. IMPLEMENTATION

The implementation phase is where we put our design plan into action. This phase involves translating the game design into working game assets and code, including creating a basic prototype of the game, programming gameplay mechanics, and implementing user interface elements. The implementation phase is a critical step in the game development process, as it is where the game truly takes shape and comes to life. In this phase, we will be leveraging the capabilities of Unreal Engine 5 to create an immersive and engaging gameplay experience, while also paying close attention to optimization and performance to ensure a smooth and enjoyable experience for players. For the development of this game, we divided the whole process into different modules with each module adding a different element to the game.

A. Project Setup

The first module of the implementation phase focused on project setup. To begin, we created the project in Unreal Engine and chose the 3rd Person Game mode. We then had the option to add starter content or not, and Unreal Engine generated a default map for us to start development. Our next step was to create the game mode blueprint, which defined the game rules and mechanics. This blueprint was responsible for setting up the initial state of the game, including spawning player characters and handling setup and initialization tasks. By creating the game mode blueprint, we were able to easily modify and iterate on the gameplay experience without extensive code changes. After creating the game mode blueprint, we implemented a basic character class with a Mesh, which is a 3D model used to represent objects in the game world. Unreal Engine provided tools for working with meshes, including importing meshes from external software, creating and editing meshes within the engine, and applying materials and textures. We also implemented a camera system and movement system that allowed players to see and follow the character and move around the game world. These systems formed the foundation for the player's interaction with the game and set the stage for the rest of the game's development.

B. Animation

In the second module of the implementation phase, we focused on implementing animations for our character. While the character could move around, it lacked animations which made it appear comical. This module aimed to provide a tutorial on how to implement animations for our character using Animation Blueprint and state machines.

Animating characters in Unreal Engine involves creating an Animation Blueprint which defines different animations and their transitions based on the character's current state. Animation Blueprints in Unreal Engine contain a set of state machines that define the various states a character can be in such as idle, walking, running, jumping, etc. Each state machine contains animation nodes that define the animations that should be played in that state. To create animations for the character, you can use external software tools such as Maya or Blender to create animation sequences and import them into Unreal Engine or buy animation assets from other artists. Once imported, the animation sequences can be added to the appropriate animation nodes in the state machine.

To ensure a smooth transition between different states, transition rules can be added between the state machines. These rules define when the character should transition from one state to another and what animation should be played during the transition.

By using Animation Blueprints and state machines in Unreal Engine, you can create complex character animations with smooth transitions between different states. We created a simple state machine for the movement mechanics that we had implemented till that point. This state machine was named "Ground Locomotion" and comprised 4 states: "idle," "run start," "run," and "run stop." Each of these states was accompanied by its animation

C. Aiming and Crosshair

In the third module of the implementation phase, we focus on implementing aiming functionality and a crosshair for our shooter game. Aiming is crucial in a shooter game as it enables the player to accurately shoot at enemies. To achieve this effect, we use the FOV (Field of View) values for our camera component, which determines how much of the environment the camera can see at a time. When the player holds the aim button, it calls a function that reduces the FOV value gradually through interpolation, creating a zoom effect.

Apart from aiming, we also create a crosshair in this module. A crosshair is an on-screen marker that helps players aim at their targets. It typically consists of a small graphic that is centered on the screen and provides visual feedback to the player on where their weapon is pointing. In Unreal Engine, you can implement a crosshair using the HUD (Heads-Up Display) blueprint.

To implement a crosshair using the HUD blueprint, you need to create a new blueprint class of type "HUD" and add a "DrawHUD" event to the Event Graph. In this event, use the "DrawLine" or "DrawTexture" function to draw a small crosshair graphic in the center of the screen, which can be customized as per the game's aesthetics. Finally, set the HUD blueprint to the level or character blueprint using the "SetHUD" function. This basic implementation can be further customized to fit the game's specific needs.

D. Weapons

During the Weapons Module of the implementation phase, the goal is to implement a base weapon class which will then be used to create three different types of weapons - pistol, assault rifle, and sub-machine gun - that will inherit from a single weapon class. To ensure that the weapons can be picked up from the ground, an "Item Class" will also be created which will be inherited by both the weapon and ammo class. This class will contain all the required variables and components so that the children of this class can interact with the world.

To create a simple weapon class with a mesh and a muzzle flash using C++ in Unreal Engine, first, create a new C++ class that inherits from the AActor class. Next, add a skeletal mesh component and a particle system component to the weapon class. The skeletal mesh component will represent the weapon in the game world, while the particle system will be used to create the muzzle flash effect when the weapon is fired. Then, add a socket to the weapon mesh where the muzzle flash particle effect will be spawned from. In the constructor of the weapon class, set the mesh and particle system components to their respective assets.

After creating the weapon class, create a fire function in your weapon class that will spawn the muzzle flash particle effect at the muzzle socket, perform a line trace from the barrel socket of your weapon mesh to detect any hit targets, and spawn impact particle effects at the hit location, if any.

In the character class, create a socket where you want to attach the weapon mesh. Then, create an instance of the weapon class and attach it to the character mesh at the socket created earlier. Finally, call the fire function of your weapon class whenever the player presses the fire button.

Additionally, a pickup widget will be created during this module. A pickup widget in Unreal Engine is a visual element that appears on the screen when a player approaches a pickup item such as weapons, ammunition, or health kits. To create a pickup widget, create a new user interface (UI) widget blueprint and customize the text properties such as font, size, and color to fit your game's aesthetic. In the event graph tab, create a new custom event called "SetPickupText" and add a "SetText" node to it.

In your level blueprint or character blueprint, create a new variable of type "Widget Class" and set it to the pickup widget blueprint you just created. When the player approaches a pickup item, call the "SetPickupText" event and pass in the text that you want to appear on the widget. When the player leaves the pickup item's trigger area, call the "SetPickupText" event again and pass in an empty string to hide the widget.

Lastly, functions to pick up and throw items from the ground will be implemented so that the player can interact with and use the items after picking them up from the ground.

E. Item Interpolation

To enhance the player's experience when picking up an item in the game, we implemented an item interpolation effect. This effect causes the item to fly up to the player's face and hover there for a brief period before being equipped. To achieve this effect, we used interpolation to manipulate the item's coordinates.

To implement the Item Interpolation module, we first added a UPROPERTY for the Z-curve and a float variable for the duration of the pickup effect to the Item class header file. We then created a function called "Pickup" that takes in a reference to the player character as a parameter.

Inside the "Pickup" function, we disabled collision and physics simulation on the item. Then, we used the SetActorLocationAndRotation() function to move the item up along the z-axis using the Z-curve for interpolation.
Once the item had moved up, we held it in place for the specified duration, using a delay node or a timer.
After the hold duration had passed, we moved the item towards the player character using SetActorLocationAndRotation() again, this time interpolating with the player character's location instead of the Z-curve.
Finally, we disabled collision and physics simulation on the item so that it can't be interacted with by the player when the item is equipped. By implementing the Item Interpolation module, we made interacting with the world much more exciting for the player.

F. Reloading

In this module, we are going to implement a reloading mechanism for our weapons. This means that after a certain number of bullets have been fired from a weapon, the player character will need to reload the weapon to continue shooting.
To achieve this, we will be keeping track of the number of bullets fired from each weapon using a counter. Once the counter reaches a predetermined number (the magazine capacity of the weapon), the player character will need to reload the weapon by pressing a reload button. This will trigger a reload animation and reset the bullet counter to zero.
In addition to implementing a reloading mechanism, we will also be adding animations to our weapons. These animations will be stored in an animation montage, which is a container for multiple animations that can be played together in a sequence. We will create an enum to keep track of the current weapon the player character is holding and use this enum to play the correct animations for each weapon type.
The precise control over the timing, blending, and looping of animations provided by animation montages allows us to create seamless and realistic animations that add to the immersion of our game.
Finally, we will be creating a widget to display the number of bullets remaining in the player character's current weapon. This widget will be updated in real-time as the player character fires their weapon and reloads, providing important feedback to the player and helping to keep them immersed in the game.

G. Advanced Movement

In any shooter game, the movement mechanics of the player character are crucial to the overall gameplay experience. While basic movement mechanics may work fine, implementing additional movement mechanics can help make the player feel more immersed in the game world. One such mechanic is turn-in-place animations.
Turn-in-place mechanics allow the player to rotate their character in place without moving their position, which is especially useful in situations where the player needs to quickly respond to an enemy or obstacle without moving their position. This mechanic can be implemented in various ways, but one common approach is to use animation blending.
Animation blending allows for multiple animations to be played together seamlessly. In the case of turn-in-place animations, a turn-in-place animation is played on the character while they remain stationary. This animation can be triggered by the player inputting a specific command, such as pressing a button or moving the mouse in a certain way. The animation can also be looped until the player stops the input command.
Another approach to implementing turn-in-place mechanics is to use root motion. Root motion is a technique used in game development where the movement of a character is controlled entirely by their animation. In this case, the turn-in-place animation would move the character's root bone (the bone at the base of the character's skeleton) to rotate them in place, without actually moving their position. This approach can give more precise control over the character's movement but may require a more complex animation setup.
In the case of our 3D FPS shooter game made using Unreal Engine, we are going to use a mix of both methods mentioned above. We are going to divide the character mesh into two parts: the upper body and the lower body, with each of these parts running different animations.
These parts will be divided by the root bone. When the player moves the camera beyond a certain threshold (in our case, 90 degrees), we are going to play the turn-in-place animations on the lower body and make the character face the current direction we are looking at.
This implementation approach will provide the player with a more immersive and realistic movement experience, allowing them to respond to situations more quickly and effectively. It will also add an extra layer of depth to the gameplay mechanics, making the game more engaging and enjoyable for the player.

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 11 Issue V May 2023- Available at www.ijraset.com

H. Ammo Pickups

In this module, we are going to delve into the implementation of a crucial aspect of any shooter game: ammo pickups. Ammo pickups serve as a means for the player to replenish their ammunition and are typically spread throughout the game level to ensure that the player is sufficiently equipped for the fight ahead.

To implement ammo pickups in our game, we will leverage the item class we created in an earlier module. We chose to use the item class because it already has the functionality required for the player to interact with the item, and by inheriting from it, we can reuse that functionality. However, the implementation of the picking-up functionality will differ from weapons, as the player will be able to pick up ammo by simply running over it.

To achieve this, we will add an overlap sphere to our ammo pickup class. When the player overlaps with this sphere, the ammo will be automatically picked up, and the pickup will be registered in the player's inventory. We will use a similar effect as the one used for weapon pickups, but instead of interpolating to the player's hand, we will interpolate to one of six different locations around the player. Each time the player picks up an ammo pickup, we will randomly select a location from these six options and interpolate the pickup to that location.

By implementing ammo pickups in our game, we add an important layer of strategy for the player. Players must not only manage their ammunition effectively but also keep an eye out for ammo pickups scattered throughout the level to maintain their ammo supplies. Additionally, by utilizing the existing item class and adding a simple overlap sphere, we can easily and efficiently implement this essential gameplay mechanic.

I. Outline and Glow Effects

In a shooter game, pickable weapons are often scattered around the game world for the player to find and use. These weapons need to stand out and be easily identifiable so that players can quickly recognize them and pick them up when they need to.

This is where outline and glow effects come in. By adding an outline effect to the weapons, we can make them visually distinct from the surrounding environment, which can help them stand out and be easily recognized by players. Similarly, by adding a glow effect to the weapons, we can make them appear more prominent and eye-catching, which can help draw the player's attention to them.

The outline effect can be implemented by rendering the object's edges in a different color or with a thicker line, which creates a distinct silhouette around the object. This can be particularly useful in darker environments or in situations where the object may blend in with the surroundings.

On the other hand, the glow effect can be implemented by adding a halo of light or color around the object, which makes it appear brighter and more noticeable. This can be particularly useful when the object is in a dimly lit environment or when the player's attention is focused on other things in the game world.

First up is the outline effect we achieve this by using custom depth. Custom depth is a rendering technique used in the Unreal Engine that allows developers to customize the depth of an object in a scene. In simpler terms, custom depth allows you to assign a specific depth value to an object in a 3D scene, which can be used for various purposes such as selecting, highlighting, or masking that object.

Custom depth works by using a separate depth buffer, which stores depth values for each pixel in the scene. By default, the depth buffer is used to determine the visibility of objects in a scene, but with custom depth, you can modify the depth values of certain objects to achieve specific visual effects.

For example, in a shooter game, custom depth can be used to highlight important objects such as weapons or enemies by assigning them a unique depth value and then applying a post-processing effect such as an outline or glow effect to those objects. This can help the player easily identify important objects in the game and enhance the overall visual appeal of the scene.

Custom depth can also be used for other purposes such as occlusion culling, where objects that are not visible to the camera are skipped during rendering, or for creating more advanced visual effects such as depth of field or motion blur.

So, to create an outline effect in our game we followed the following process:

1) Enable Custom Depth for the object: In the Details panel of the object, you want to apply the outline effect to, under the Rendering section, set "Render Custom Depth" to true.
2) Create a Post Process Material: In the Content Browser, right-click and select Material. Name it "PostProcessMaterial" or something similar. Then, double-click to open it.
3) Set up the Material: In the Material Editor, add a Custom Depth node, and connect it to an Emissive node. Then, connect the Emissive node to the Material Output node. You can adjust the Emissive value to control the intensity of the outline effect.

4) Apply the Material to the Post Process Volume: In the Level Editor, drag a Post Process Volume into your level. Then, under the Post Process Volume's settings, set "Blendables" to your PostProcessMaterial.
5) Adjust the Custom Depth Stencil Value: By default, Custom Depth writes to the stencil buffer with a value of zero. You can adjust the stencil value for the objects you want to outline. To do this, select the object, and in the Details panel under the Rendering section, set "Custom Depth Stencil Value" to a value greater than zero.
6) Add an outline post-process material: In the Content Browser, create a new Material and add the following nodes to it: Scene Color, Custom Depth, and PostProcessInput0. Then, subtract Custom Depth from Scene Color and connect the result to PostProcessInput0. Finally, connect PostProcessInput0 to the Material Output.
7) Add a post-process volume: In the Level Editor, drag a Post Process Volume into your level. Then, under the Post Process Volume's settings, set "Blendables" to your new outline post-process material.
8) Adjust outline settings: You can adjust the outline settings in your new post-process material to achieve the desired effect. For example, you can adjust the emissive value, the thickness of the outline, and the color of the outline.

Another effect we added in this module is the fresnel effect. The Fresnel effect is an optical phenomenon that occurs when light reflects off a surface at a grazing angle. It causes the reflected light to be more intense around the edges of the surface, creating a distinct visual effect. In video games, this effect is often used to create a highlight around the edges of objects, giving them a more pronounced and stylized appearance.

To add this in our game we used material functions and material instances and the whole process is explained below:

a) Create a new material.
b) Create a scalar parameter named "Fresnel Power" and set its default value to 5. This parameter will control the strength of the Fresnel effect.
c) Create a scalar parameter named "Fresnel Bias" and set its default value to 0. This parameter will control the position of the Fresnel effect.
d) Create a material function by right-clicking in the Material Editor and selecting "Create Material Function". Name the function "Fresnel".
e) In the material function, create a Custom node and connect it to the output node.
f) Inside the Custom node, create a Lerp node and connect it to the output.
g) Connect the Fresnel Bias parameter to the Alpha input of the Lerp node.
h) Create a Fresnel node and connect it to the A input of the Lerp node.
i) Create a Power node and connect it to the Fresnel node. Connect the Fresnel Power parameter to the Power input of the Power node.
j) Create a Constant node with a value of 1 and connect it to the B input of the Lerp node.
k) Save the material function.
l) Back in the Material Editor, create a new material instance from the base material.
m) In the material instance, find the "Fresnel" material function and drag it into the graph.
n) Connect the output of the material function to the Emissive Color node.
o) Adjust the "Fresnel Power" and "Fresnel Bias" parameters to achieve the desired effect.
p) Apply the material instance to the object you want to have the Fresnel effect.

The last effect that we are going to add in this module is a scrolling effect this effect also works along with the other two effects to create a glow and outline effect.

And we follow the following process to implement it:

- First, create a new Material in Unreal Engine by right-clicking in the Content Browser, selecting "Material" and naming it appropriately.
- Inside the Material Editor, create a new Texture Sample node and connect it to the Base Color input of a Material Output node.
- Select the Texture Sample node, and in the Details panel, select the texture you want to use for your scrolling effect.
- Add a Texture Coordinate node to your material by right-clicking in the Material Graph and selecting "Texture Coordinate."
- Create a new Scalar Parameter node by right-clicking in the Material Graph and selecting "Scalar Parameter."
- Name the Scalar Parameter "ScrollSpeed" or something similar, and set its default value to a low number, such as 0.1.

- Create a new Material Function by right-clicking in the Content Browser, selecting "Material Function," and naming it something like "ScrollingEffect."
- In the Material Function, create a new Constant node and set its value to 0.5.
- Connect the Constant node to the A input of a Multiply node.
- Connect the Texture Coordinate node to the B input of the Multiply node.
- Connect the output of the Multiply node to the UV input of the Texture Sample node.
- Create a new Material Instance by right-clicking in the Content Browser, selecting "Material Instance," and selecting the Material you created in Step 1.
- In the Details panel of the Material Instance, find the Scalar Parameter you created in Step 5 and set its value to something higher, such as 10.
- Apply the Material Instance to the object you want to apply the scrolling effect to.
- To animate the scrolling effect, create a Blueprint or Matinee sequence that adjusts the Scalar Parameter over time, creating the illusion of scrolling.

By adjusting the ScrollSpeed value in the Material Instance, you can control the speed at which the texture scrolls.

IV. MULTIPLE WEAPON TYPES

The 10th module in our implementation phase focuses on creating three different types of weapons based on the base class we designed in the previous module. These weapons differ in terms of properties such as fire rate, damage, sound, reload animations, magazine capacity, muzzle flash, custom depth stencil, and whether they are automatic or not.

To manage all these properties and make changes to them when needed, we use data tables in Unreal Engine. A data table is a way to store and organize data in a table-like format, similar to a spreadsheet. It is a powerful tool for managing large amounts of data, such as game settings, level layouts, character attributes, and more.

In our case, we created a "Weapon Data Table" to store all the properties of each specific gun. This data table consists of rows and columns, where each row represents a single entry or record, and each column represents a specific property or attribute of that record. For example, one row in our Weapon Data Table might represent a particular weapon, such as an assault rifle, and its columns would contain the various properties that define that weapon.

Whenever we change the weapon, the weapon class retrieves the properties for that weapon from the data table and updates them accordingly. This process is done using an overridden function called "OnConstruction()", which is called whenever the object is moved in the world.

By using data tables, we can easily manage all the different properties of each weapon type and make changes to them when needed. This helps us to create a more dynamic and customizable game experience for players, where they can choose from different types of weapons that have unique properties and characteristics.

A. Footsteps

Footsteps are an essential audio element in any shooter game, as they provide important audio cues to players about the location and movement of other players or enemies, add to the game's overall immersion and realism, and can be used as a tool for the game design.

In this module, we will create a footsteps system using a custom AnimNotify called FootstepNotify to implement five types of footsteps: wood, metal, stone, water, and grass in Unreal Engine. Here are the steps we will follow:

1) Create an enum called EFootstepType in C++ to include all the types of footsteps we want to include in the game.
2) Create an AnimNotify called FootstepNotify in the character blueprint to trigger the sound effect for each footstep.
3) Create a new variable called FootstepType of type EFootstepType in the FootstepNotify to determine the type of surface the character is walking on and play the appropriate sound effect.
4) Create a sound cue for each type of footstep, either by using existing sound effects or creating our own.
5) Add code to play the appropriate sound cue based on the FootstepType variable in the FootstepNotify using the PlaySoundAtLocation function.
6) Add the FootstepNotify to the animation timeline at the appropriate time in the animation blueprint using the Notify section.
7) Create a function called SetFootstepType in the character blueprint to determine the type of surface the character is walking on and set the FootstepType variable in the FootstepNotify accordingly.

8) Use a trace from the character's feet to determine the type of surface they are walking on and set the FootstepType variable in the SetFootstepType function accordingly.
9) Test the footsteps system in the game, making sure that the sound effects are playing at the appropriate times and that the FootstepType variable is being set correctly based on the character's movement.
10) By implementing this footsteps system, we can enhance the player's experience by providing them with vital audio cues, increasing the game's immersion and realism, and using sound as a tool for game design.

B. Enemy

In this 12th module of our implementation phase, we are going to focus on creating and fleshing out the enemy class. This class will be similar to the character class we created earlier, with the key difference being that the enemy class will not have any movement mechanics. Instead, all of the enemy movement will be controlled by the AI system, which we will discuss in the next module.

We will add various properties to the enemy class, such as base damage and headshot damage, which will determine how much damage an enemy does when it attacks the player. We will also create a simple widget to display the health of the enemies and create a function to handle enemy death. All the animations related to enemy death will be compiled into a death montage, and a movement montage will be created for the running animations.

We will also implement a HUD element to show hit numbers indicating the location of bullet hits. To achieve this, we will create an interface called the BulletHitInterface, which will be implemented by both the enemy and character classes. This interface consists of a bullet hit function that will be implemented in both classes and will help us track each bullet hit and apply damage accordingly.

By the end of this module, we will have a fully functional enemy class with various properties and animations, and a HUD element to display hit numbers for bullet hits. This will add another layer of depth to our game and make it more engaging for the player.

C. AI and Behavior Trees

This module involves implementing various AI-related features using Unreal Engine's AI tools and features.

Firstly, you will create a Behavior Tree asset using the Behavior Tree Editor in Unreal Engine. This Behavior Tree will be used to create the AI behaviors for your game's enemies, including their patrol behavior and attacking behavior. The Behavior Tree Editor provides a visual interface that allows you to easily create and modify AI behaviors.

Within the Behavior Tree, you will create a Patrol node that represents the enemy's patrol behavior between two points. This will involve adding two MoveTo task nodes that instruct the enemy to move to the first and second patrol points. You will also create a Selector node that checks if the player is within aggro and combat ranges and branches out to either a Chase node or a Wait node depending on the player's location.

If the player is within aggro range, you will create a Chase node that represents the enemy's behavior of chasing the player. This will involve adding a MoveTo task node that instructs the enemy to move toward the player's location. Once the enemy reaches the player's location, the Selector node will check if the player is within combat range and branch out to an Attack node if so.

Within the Attack node, you will create an attack task node that instructs the enemy to attack the player. This will allow the enemy to attack the player and make the gameplay more challenging.

In addition to AI-related features, you will also create animation montages for attacking and adding death functionality for the player. You will also add particle effects for blood to make the game more visually appealing and immersive.

D. Level Creation

Creating a map in Unreal Engine can involve a variety of tasks, such as building terrain, placing static meshes, creating materials, and lighting the scene.

To begin, you can start with a base terrain, which can be generated using the terrain sculpting tools in the engine. You can then add various landscape layers, such as grass, rocks, and sand, to add more detail and realism to the terrain.

Once you have a base terrain, you can start placing static meshes to create the structures and props for your map. Unreal Engine provides a vast library of built-in assets that you can use, or you can import your assets from external sources. In addition, you can use the engine's foliage tool to place trees, bushes, and other vegetation on the terrain.

Next, you can create materials and textures to give your assets and terrain the desired look and feel. Unreal Engine provides a robust material editor that allows you to create complex shaders and apply them to your assets. You can also use various texture maps, such as diffuse, normal, and specular maps, to add more details and realism to your materials.

Finally, you can add lights to your scene to create the desired mood and atmosphere. Unreal Engine provides various types of lights, such as point lights, spotlights, and directional lights, which you can use to light up your scene. You can also adjust the color, intensity, and other properties of the lights to achieve the desired effect.

Overall, creating a map in Unreal Engine involves a mix of technical and artistic skills, as you need to use the engine's tools and features to create a visually stunning and immersive game environment.

E. Testing

Testing is a crucial phase in game development that ensures the game is functional, stable, and enjoyable for players. It involves identifying and addressing bugs, glitches, and other issues that could affect gameplay, as well as fine-tuning game mechanics and balancing difficulty levels.

During the testing phase, game developers typically perform several types of tests, including:

1) Functional testing: This involves testing the game's basic features, such as movement, combat, and interactions with objects and characters in the game world. Testers will look for bugs, glitches, and other issues that affect the game's functionality.
2) Performance testing: This involves testing the game's performance on different hardware and software configurations, including different devices and operating systems. Testers will look for issues such as slow loading times, frame rate drops, and other performance issues.
3) Compatibility testing: This involves testing the game's compatibility with different hardware and software configurations, including different graphics cards, processors, and operating systems. Testers will ensure that the game works on a wide range of devices and platforms.
4) User experience testing: This involves testing the game's overall user experience, including its interface, controls, and overall feel. Testers will look for ways to improve the game's usability and make it more enjoyable for players.
5) Regression testing: This involves re-testing previously fixed bugs to ensure that they have not reappeared and that the fixes have not introduced new issues.

Testing can be performed in-house by the game development team or by external testers who are recruited specifically for testing purposes. Testing can also be performed by a combination of both internal and external testers.

F. Deployment

Once the game has been tested and is functioning correctly, it can be deployed. This involves packaging the game and distributing it to players. Unreal Engine 5 provides a range of options for deploying games, including publishing to the Epic Games Store, as well as other distribution platforms.

G. Challenges Faced

During the game development process, developers often face various challenges such as bugs, glitches, code inconsistencies, and performance optimization. For our game, the biggest challenge was to optimize it to run smoothly without any performance issues. This was particularly important since our game had visually stunning graphics, with a lot of props and textures, which could be taxing on the system. Unfortunately, this resulted in texture stream overflow and exhausted video memory, which had to be addressed to ensure a smooth gameplay experience.

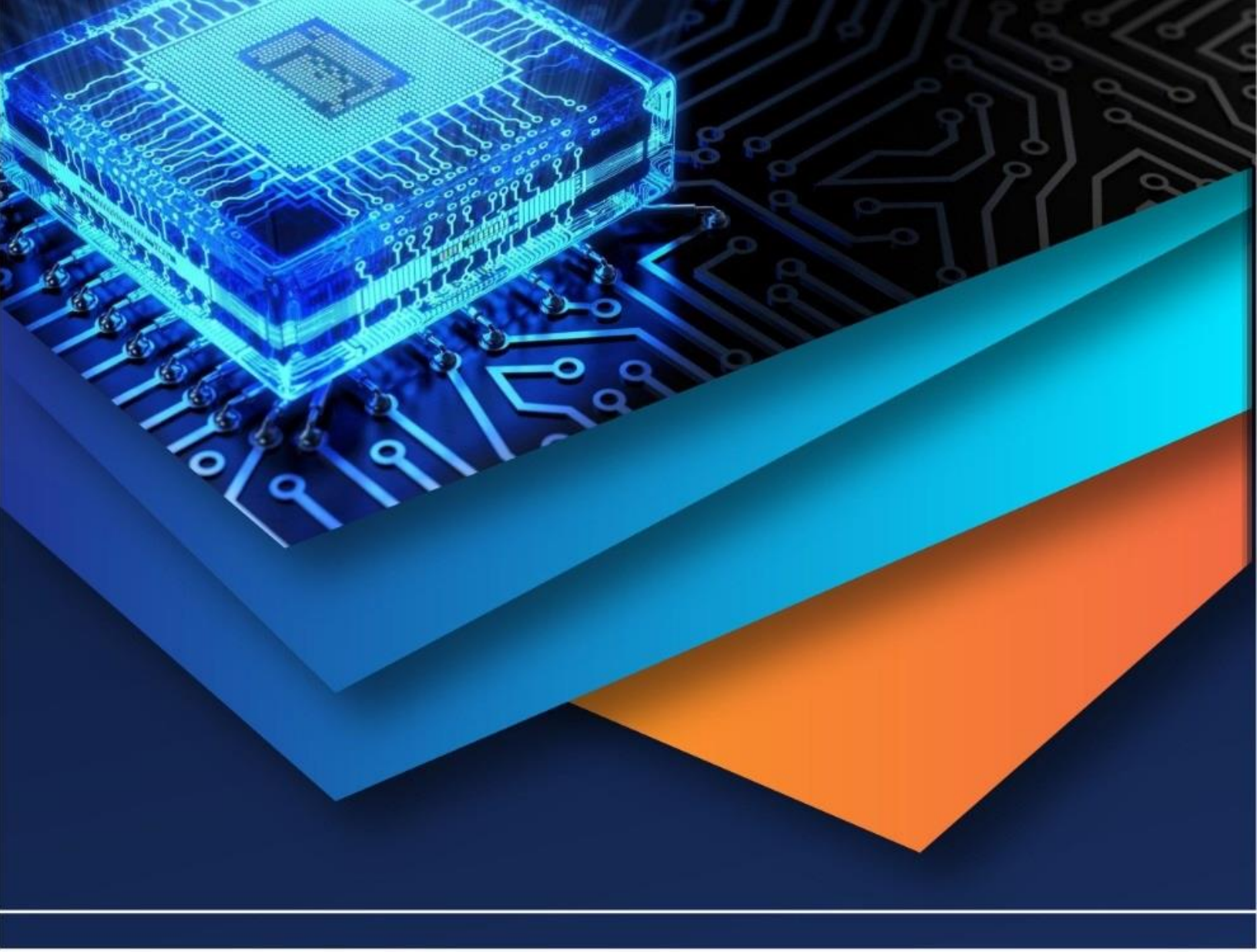
V. CONCLUSION

Unreal Engine 5 is an excellent tool for game development, and it can be used to create high-quality 3D first-person shooter games. This research paper has covered the basics of game development using Unreal Engine 5 and how it can be used to create a 3D first-person shooter game. The paper has also covered the steps involved in the game development process, from designing the game to testing and deployment. Additionally, the paper has discussed the challenges faced during the game development process and how they were overcome. Overall, developing a 3D first-person shooter game using Unreal Engine 5 is a challenging but rewarding experience.

REFERENCES

[1] Epic Games. (2021). Unreal Engine 5. Retrieved from https://www.unrealengine.com/en-US/blog/a-first-look-at-unreal-engine-5
[2] Hill, A. (2019). Mastering Unreal Engine 4.x. Packt Publishing.
[3] Holcomb, J. (2018). The Ultimate Guide to 2D Mobile Game Development with Unity. Apress.

[4] Juul, J. (2005). Half-real: Video games between real rules and fictional worlds. MIT Press.
[5] Sutherland, I., & Wiberg, C. (2018). Game development with Unreal Engine 4: From beginner to professional. CRC Press.
[6] Unreal Engine. (2021). Unreal Engine Documentation. Retrieved from https://docs.unrealengine.com/en-US/index.html
[7] Waltham, C., & Drysdale, T. (2017). Game programming using Qt 5 Beginner's Guide, Second Edition. Packt Publishing.



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY