International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 08 Issue: 04 | Apr 2021 www.irjet.net p-ISSN: 2395-0072

Game Development using Artificial Intelligence in Unreal Engine

Ritik Kothari[1], Smit Nawar[2], Siddharth Kothari[3], Asst. Prof. Jaya Jeswani[4]

[1-3]*Student, Information Technology, Xavier Institute of Engineering, Mumbai, India*
[4]*Assistant Professor, Department of Information Technology, Xavier Institute of Engineering, Mumbai, India*

Abstract - *Artificial intelligence has been a growing resource for video games for years now. Most video games, whether they're racing games, shooting games, or strategy games, have various elements that are controlled by AI, such as the enemy bots or neutral characters. Even the ambiguous characters that don't seem to be doing much are programmed to add more depth to the game, and to give you clues about what your next steps should be. AI in video games is a distinct subfield and differs from academic AI. It serves to improve the game-player experience rather than machine learning or decision making. During the golden age of arcade video games, the idea of AI opponents was largely popularized, in the form of graduated difficulty levels, distinct movement patterns, and in-game events de-pendent on the player's input. AI is often used in mechanisms which are not immediately visible to the user, such as data mining and procedural-content generation. However, there are also many other ways that AI and game development are growing through each other. Although AI continues to be used to bring video games to life, video games are now being trained to study their own patterns so as to improve their own algorithms, which is just one of many ways that AI is becoming more advanced.*

***Key Words*: Artificial Intelligence, AI Agents, Video games.**

1. INTRODUCTION

The term "Game AI" is used to refer to a broad set of algorithms that include techniques from control theory, robotics, computer graphics and computer science in general, and so video game AI may often not constitute "True AI" in that such techniques do not necessarily facilitate computer learning or other standard criteria, only constituting "automated computation" or a predetermined and limited set of responses to a predetermined and limited set of inputs.[1][2][3] Game AI/heuristic algorithms are used in a wide variety of quite disparate fields inside a game. The most obvious is in the control of any NPCs in the game, although "scripting" (decision tree) is currently the most common means of control.[4] These handwritten decision trees often result in "artificial stupidity" such as repetitive behaviour, loss of immersion, or abnormal behaviour in situations the developers did not plan for.[5] Pathfinding, a common use for AI, is widely seen in real-time strategy games. Pathfinding is the method for determining how to get a NPC from one point on a map to another, taking into consideration the terrain, obstacles and possibly "fog of war".[6][7] Commercial videogames often use fast and simple "grid-based pathfinding", wherein the terrain is mapped onto a rigid grid of uniform squares and a pathfinding algorithm such as A* or IDA* (graph traversals) is applied to the grid.[8][9][10] Instead of just a rigid grid, some games use irregular polygons and assemble a navigation mesh out of the areas of the map that NPCs can walk to.[8][11] As a third method, it is sometimes convenient for developers to manually select "waypoints" that NPCs should use to navigate; the cost is that such waypoints can create unnatural-looking movement. In addition, waypoints tend to perform worse than navigation meshes in com-plex environments.[12][13] Beyond static pathfinding, navigation is a sub-field of Game AI focusing on giving NPCs the capability to navigate in a dynamic environment, finding a path to a target while avoiding collisions with other entities. Rather than improve the Game AI to properly solve a difficult problem in the virtual environment, it is often more cost-effective to just modify the scenario to be more tractable. If pathfinding gets bogged down over a specific obstacle, a developer may just end up moving or deleting the obstacle.[14]

2. LITERATURE REVIEW

Bogost, Ian (March 2017). ""Artificial Intelligence" Has Become Meaningless". Retrieved 19 May 2020.

Excerpt: When the mathematician Alan Turing accidentally invented the idea of machine intelligence almost 70 years ago, he proposed that machines would be intelligent when they could trick people into thinking they were human. At the time, in 1950, the idea seemed unlikely; Even though Turing's thought experiment wasn't limited to computers, the machines still took up entire rooms just to perform relatively simple calculations. But today, computers trick people all the time. Not by successfully posing as humans, but by convincing them that they are sufficient alternatives to other tools of human effort. Twitter and Facebook and Google aren't "better" town halls, neighborhood centers, libraries, or newspapers—they are different ones, run by computers, for better and for worse. The implications of these and other services must be addressed by understanding them as particular implementations of software in corporations, not as totems of otherworldly AI.

Kaplan, Jerry (March 2017). "AI's PR Problem". MIT Technology Review.

Excerpt: AI makes use of some powerful technologies, but they don't fit together as well as you might expect. Early researchers focused on ways to manipulate symbols according to rules. This was useful for tasks such as proving mathematical theorems, solving puzzles, or laying out

integrated circuits. But several iconic AI problems such as identifying objects in pictures and converting spoken words to written language proved difficult to crack. More recent techniques, which go under the aspirational banner of machine learning, proved much better suited for these challenges. Machine-learning programs extract useful patterns out of large collections of data. The power recommendation systems on Amazon and Netflix, hone Google search results, de-scribe videos on YouTube, recognize faces, trade stocks, steer cars, and solve a myriad of other problems where big data can be brought to bear. But neither approach is the Holy Grail of intelligence. Indeed, they coexist rather awkwardly under the label of artificial intelligence. The mere existence of two major approaches with different strengths calls into question whether either of them could serve as a basis for a universal theory of intelligence.

Eaton, Eric; Dietterich, Tom; Gini, Maria (December 2015). "Who Speaks for AI?" (PDF). AI Matters.

Excerpt: These are boom times for AI. Articles celebrating the success of AI research appear frequently in the international press. Every day, millions of people routinely use AI-based systems that the founders of the field would hail as miraculous. And there is a palpable sense of excitement about impending applications of AI technologies.

Good, Owen S. (5 August 2017). "Skyrim mod makes NPC interactions less scripted, more Sims-like". Polygon. Retrieved 19 May 2020.

Excerpt: Researchers at Portugal's Universidade de Lisboa and North Carolina State University have developed a tool and a mod that makes NPC interactions in The Elder Scrolls 5: Skyrim a little more like those in The Sims, with the overall goal of varying NPC behavior for a richer player experience. The tool is called CIFCK, built on the Comme il-Faut AI model developed in 2012. It's implemented in "Social Skyrim," which is part of the master's thesis of Manuel Guimarães, a student at Lisbon. The mod creates greater variability in NPC interactions by allowing them to act on their changing opinions of other NPCs, which are shaped by their interactions with the rest of the NPCs and the player. The original CIF architecture would keep track of NPCs' feelings but didn't turn those feelings into action.

Lara-Cabrera, R., Nogueira-Collazo, M., Cotta, C., & Fernández-Leiva, A. J. (2015).Game artificial intelligence: challenges for the scientific community.

Excerpt: Traditionally the Artificial Intelligence (AI) of a game has been coded manually using predefined sets of rules leading to behaviors often encompassed within the so called artificial stupidity, which results in a set of known problems such as the feeling of unreality, the occurrence of abnormal behaviors in unexpected situations, or the existence of predictable behaviors, just to name a few. Advanced techniques are currently used to solve these problems and achieve NPCs with rational behavior that takes logical decisions in the same way as a human player. The main advantage is that these techniques perform automatically the search and optimization process to find these smart strategies.

Yannakakis, G. N. (2012, May). Game AI revisited. In Proceedings of the 9th conference on Computing Frontiers (pp. 285–292). ACM.

Excerpt: Since those first days of academic game AI the term was mainly linked to nonplayer character (NPC) behavior (i.e. NPC AI) and pathfinding as most of the early work in that field was conducted by researchers with AI, optimization and control background and research experience in adaptive behavior, robotics and multiagent systems1. AI academics used the best of their computational intelligence and AI tools to enhance NPC behavior in generally simple, research-focused, non-scalable projects of low commercial value and perspective. In almost every occasion the two (academic and industrial game AI), rather immature, communities would meet they would conclude about the gap existent between them and the need of bridging it for their mutual benefit. The key message of academic AI has been that industry does not attempt to use sophisticated AI techniques with high potential (e.g. neural networks) in their games. On the other end, the central complaint of industrial game AI has been the lack of domain knowledge and practical wisdom when it comes to realistic problems and challenges faced during game production.

Hagelback, Johan, and Stefan J. Johansson. "Dealing with fog of war in a real-time strategy game environment." In Computational Intelligence and Games, 2008. CIG'08. IEEE Symposium On, pp. 55-62. IEEE, 2008.

Excerpt: A bot that uses potential fields can be modified to deal with imperfect information, i.e. the parts of the game world where no own units are present are unknown (usually referred to as Fog of War, or FoW).

Abd Algfoor, Zeyad; Sunar, Mohd Shahrizal; Kolivand, Hoshang (2015). "A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games". International Journal of Computer Games Technology. 2015: 1–11. doi:10.1155/2015/736138.

Excerpt: The graph generation problem for “terrain topology” is considered a foundation of robotics and video games applications. In this problem, the pathfinding navigation is conducted in different continuous environments, such as known 2D/3D environments and unknown 2D environments. Several different techniques have been proposed to represent the navigation environment for the graphs of these three scenarios. Each of the representative environment graphs in this paper refers to one of two techniques, skeletonization or cell decomposition.

Yap, Peter. "Grid-based path-finding." In Conference of the Canadian Society for Computational Studies of Intelligence, pp. 44-55. Springer, Berlin, Heidelberg, 2002.

Excerpt: Square grids are simple yet informative models of both artificial (typically appearing in video games) and physical (involved in robotics) 2D environments used for path planning [Yap, 2002]. Typically, in grid path finding an agent is presumed to move from one traversable (unoccupied) cell to one of its 8 adjacent neighbors. Sometimes diagonal moves are prohibited restricting agent’s moves to only the 4 cardinal directions.

Sturtevant, N. R. (June 2012). "Benchmarks for Grid-Based Pathfinding". IEEE Transactions on Computational Intelligence and AI in Games. 4 (2): 144–148. doi:10.1109/TCIAIG.2012.2197681.

Excerpt: The study of algorithms on grids has been widespread in a number of research areas. Grids are easy to implement and offer fast memory access. Because of their simplicity, they are used even in commercial video games. But, the evaluation of work on grids has been inconsistent between different papers. Many research papers use different problem sets, making it difficult to compare results between papers. Furthermore, the performance characteristics of each test set are not necessarily obvious. This has motivated the creation of a standard test set of maps and problems on the maps that are open for all researchers to use.

Goodwin, S. D., Menon, S., & Price, R. G. (2006). Pathfinding in open terrain. In Proceedings of International Academic Conference on the Future of Game Design and Technology.

Excerpt: The lack of geo-awareness of such representations becomes most obvious in open terrain environments. In contrast to maze-like environments where A*'s expanding search fringe is constrained by walls and obstructions, in open terrains, expansion is only constrained by the heuristic and terrain cost. To find a path of a given length in the open is typically much more expensive than finding a path of the same length in a constrained environment. Approaches which are feasible for the confines of a dungeon fall down in the light of day. A* search is among the most popular pathfinding techniques adopted by commercial game developers.

Nareyek, A. (2004). AI in computer games. Queue, 1(10).

Excerpt: "The main role of graphics in computer games will soon be over; artificial intelligence is the next big thing!" Although you should hardly buy into such statements, there is some truth in them. The quality of AI (artificial intelligence) is a high-ranking feature for game fans in making their purchase decisions and an area with incredible potential to increase players' immersion and fun.

Cui, X., & Shi, H. (2011). A*based pathfinding in modern computer games. International Journal of Computer Science and Network Security, 11(1), 125-130.

Excerpt: Pathfinding in computer games has been investigated for many years. It is probably the most popular but frustrating game artificial intelligence (AI) problem in game industry. Various search algorithms, such as Dijkstra's algorithm, bread first search algorithm and depth first search algorithm, were created to solve the shortest path problem until the emergence of A* algorithm as a provably optimal solution for pathfinding. Since it was created, it has successfully attracted attention of thousands of researchers to put effort into it. A long list of A*based algorithms and techniques were generated.

Design Techniques and Ideals for Video Games". Byte Magazine. 7 (12). 1982. p. 100.

Excerpt: Game manufacturers and authors constantly try to answer why some computer games are better than others. Many factors contribute to the appeal of a computer game, including technical quality, graphics, sound, pace, gameplay, and action. Yet one cannot merely list the properties of a given game and expect the length of the list to tell whether that game will be a success. Giving the player an advantage of sheer numbers is to provide it with intelligence adequate to meet the human player on equal terms. Unfortunately, AI techniques are not understood well enough to be useful in such context.

3. PROPOSED WORK

To develop a sandbox environment where the player character completes a specific set of objectives to proceed the main goal. The game will simulate combat from a third-person perspective. The player will encounter opponent waves they need to survive in order to progress. The opponents will be predictable AI which the player can predict in order to overcome and defeat them. The proposed work can be divided into ten modules.

3.1. Setting up the Character and Controller class

Prerequisite: Create a C++ based project with no starter content in Unreal Engine 4.

Create a new C++ class in Unreal Engine 4 (UE4), de-rived from the Parent Class: ACharacter. Let UE4 generate the C++ files, declare a USpringArmComponent* variable in the header file, this component tries to maintain its children at a fixed distance from the parent, but will retract the children if there is a collision, and spring back when there is no collision. USpringArmComponent* is used as a 'camera boom' to keep the follow camera for a player from colliding into the world. The U prefix indicates that it's derived from a UObjectBase class, topmost in the Unreal Engine 4 hierarchy. UE4 prime hierarchy is as follows: UObject -> AActor -> APawn -> ACharacter. The A prefix indicates that it's an Actor. Prefixes are used for class names for UE4's reflection and garbage collection system.

```
/** Positions the camera behind the player. */
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Camera, meta = (AllowPrivateAccess =
  "true"))
class USpringArmComponent* CameraBoom;
```

In the .cpp file associated with the header file for the Character class created, initialize and setup the 'camera boom' by using CreateDefaultSubobject <USpringArmComponent> which is a template function for creation of a component or subobject. This is done in the Constructor itself which gets fired up at the start of the program.

```
// Pulls towards the player if there's a collision.
CameraBoom = CreateDefaultSubobject<USpringArmComponent>(TEXT("CameraTPP"));
CameraBoom->SetupAttachment(GetRootComponent());
CameraBoom->TargetArmLength = 600.f; // Camera distance from the char.
CameraBoom->bUsePawnControlRotation = true; // Rotates arm based on controller.
```

The Camera is then defined similarly as the CameraBoom but using the template function for <UCameraComponent> and attached to the Actor. Derive a blueprint from this C++ Character class to set the Skeletal Mesh in the blueprint to the Actor mesh which will be visible in the viewport.

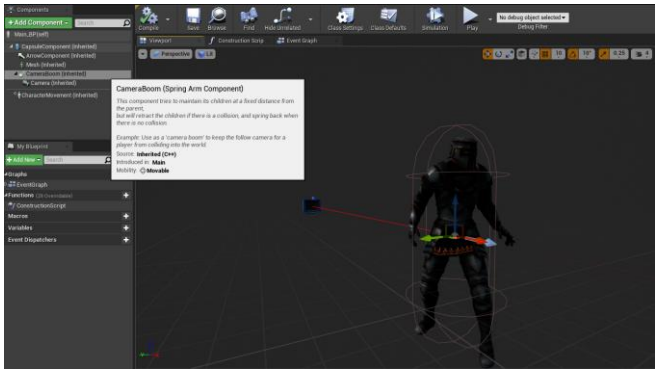

Fig 1: Spring Arm Component.

For Character movement, declare a function MoveForward() which takes a float argument to move the Character a specific value.

```
void AMain::MoveForward(float val) {
    bMovingForward = false;
    if (CanMove(val)) {
        // Finds which way is forward.
        const FRotator Rotation = Controller->GetControlRotation();
        const FRotator YawRotation(0.f, Rotation.Yaw, 0.f);
        const FVector Direction = FRotationMatrix(YawRotation).GetUnitAxis(EAxis::X);
        AddMovementInput(Direction, val);
        bMovingForward = true;
    }
}
```

3.2. Creating animation Blueprints and Blend spaces.

To add animations to this Character, create an Animation Blendspace for idle, walking and running animations.

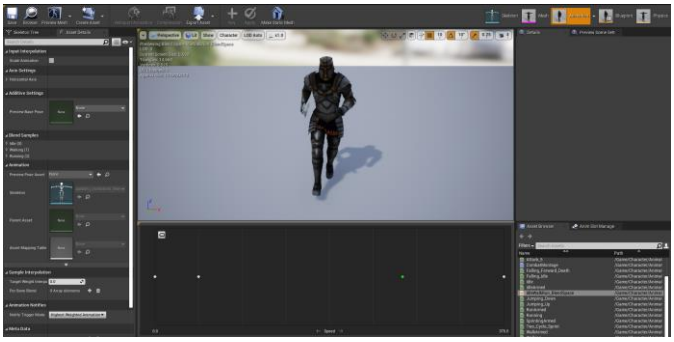

Fig 2: Animation blendspace for movement.

In the AnimGraph for the Animation blueprint, create a State Machine. This State Machine takes care of various states the Character can be in during any particular time in game.

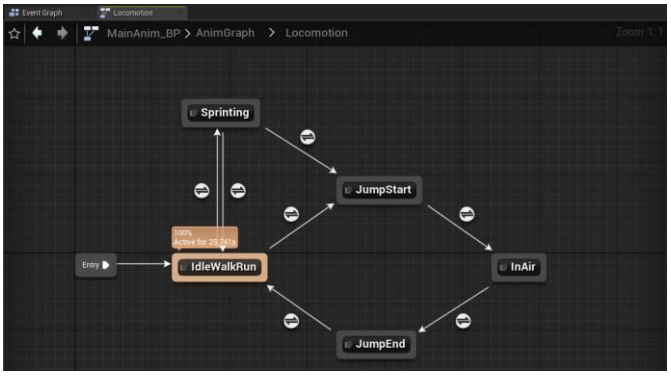

Fig 3: State machine for movement.

The first state termed as IdleWalkRun consists of a state machine. The state machine is further expanded for Unarmed and Armed states.

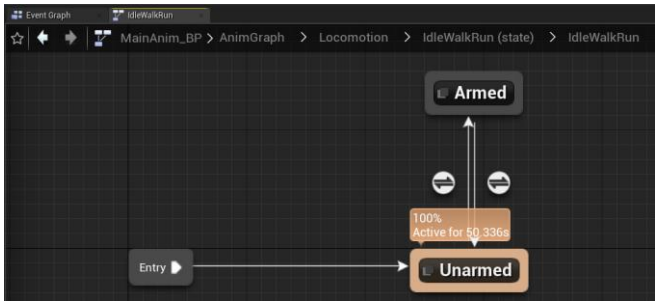

Fig 4: Transition and states for weapons equipped.

The Unarmed state consists of getting the MovementSpeed variable from C++ and using the blendspace which was created for Idle/Walk/Run animations, connecting it to the output.

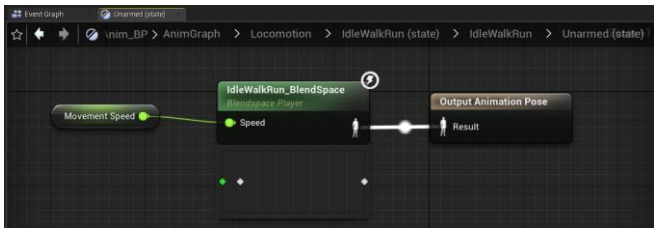

Fig 5: Linking the blendspace to the output pose.

Going back to the state machine inside IdleWalkRun, a transition is made from Unarmed to Armed with the following transition rules.

Fig 6: Transition rule for Unarmed to Armed.

Similar to the blendspace created for Unarmed animations, create a 1D blendspace for Armed idle, walking and running animations

3.3. Gameplay Mechanics.

This module includes implementation of different gameplay mechanics like floor switch, interactable items, floating platform, spawn volume and creating the HUD.

A new C++ class is created derived from the Parent Class: Actor. In the FloorSwitch.h header file, UBoxComponent* and UStaticMeshComponent* variables are declared for TriggerBox, FloorSwitch and the Door.

```
/** Volume which overlaps to trigger some functionality. */
UPROPERTY(VisibleAnywhere, BlueprintReadWrite, Category = FloorSwitch)
class UBoxComponent* TriggerBox;

/** Switch for the character to step on. */
UPROPERTY(VisibleAnywhere, BlueprintReadWrite, Category = FloorSwitch)
class UStaticMeshComponent* FloorSwitch;

/** Functionality when the floor switch is stepped on. */
UPROPERTY(VisibleAnywhere, BlueprintReadWrite, Category = FloorSwitch)
UStaticMeshComponent* Door;
```

These are defined further in the .cpp file by CreateDefaultSubboject template function.

```
TriggerBox = CreateDefaultSubobject<UBoxComponent>(TEXT("TriggerBox"));
RootComponent = TriggerBox;

TriggerBox->SetCollisionEnabled(ECollisionEnabled::QueryOnly);
TriggerBox->SetCollisionObjectType(ECollisionChannel::ECC_WorldStatic);
TriggerBox->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Ignore);
TriggerBox->SetCollisionResponseToChannel(ECollisionChannel::ECC_Pawn,
  ECollisionResponse::ECR_Overlap);

TriggerBox->SetBoxExtent(FVector(62.f, 62.f, 32.f));

FloorSwitch = CreateDefaultSubobject<UStaticMeshComponent>(TEXT("FloorSwitch"));
FloorSwitch->SetupAttachment(GetRootComponent());

Door = CreateDefaultSubobject<UStaticMeshComponent>(TEXT("Door"));
Door->SetupAttachment(GetRootComponent());
```

In the BeginPlay() function this Actor, namely FloorSwitch, the AddDynamic macro is used. OnComponentBeginOverlap event is called when something starts to overlaps this component, for example a player walking into a trigger. Helper macro for calling AddDynamic() on dynamic multi-cast delegates. Automatically generates the function name string. It takes UserObject and FuncName as parameters. UserObject passed is this, which is essentially the FloorSwitch Actor itself and the function names for the relevant functions passed by reference.

```
void AFloorSwitch::OnOverlapBegin(UPrimitiveComponent* OverlappedComponent, AActor* OtherActor,
 UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult) {
   UE_LOG(LogTemp, Warning, TEXT("Overlap begun."))

   if (!bIsCharOnSwitch)
       bIsCharOnSwitch = true;
   RaiseDoor();
   LowerFloorSwitch();
}

void AFloorSwitch::OnOverlapEnd(UPrimitiveComponent* OverlappedComponent, AActor* OtherActor,
 UPrimitiveComponent* OtherComp, int32 OtherBodyIndex) {
   UE_LOG(LogTemp, Warning, TEXT("Overlap ended."))

   if (bIsCharOnSwitch)
       bIsCharOnSwitch = false;
   GetWorldTimerManager().SetTimer(SwitchHandle, this, &AFloorSwitch::ClosedDoor, SwitchTime);
}
```

The boolean bIsCharOnSwitch is used to determine whether the Character is overlapping with the TriggerBox. On OverlapBegin fires the RaiseDoor() and LowerFloorSwitch() which are both BlueprintImplementableEvents, defined in blueprints for this Actor.

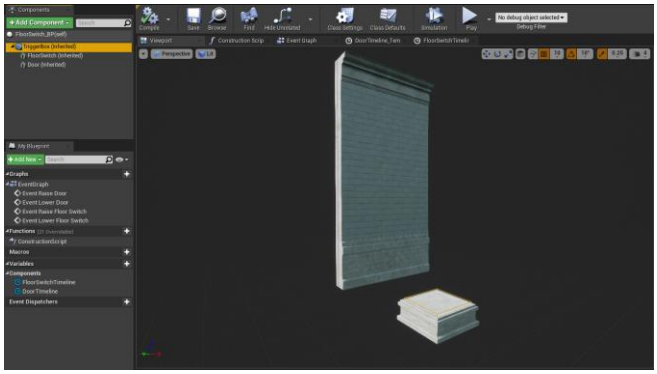

Fig 7: Static mesh for the trigger-box and door.

Appropriate Static Mesh are set for the Door and the FloorSwitch itself. In the Event Graph for this blueprint, the C++ declared BlueprintImplementableEvents are called here. DoorTimeline and FloorSwitchTimeline timelines are created. Both of these timelines consist of float tracks. The float track varies from value 0.0 to 0.75 and stops at the 0.75 value. This allows the Door to operate smoothly on the function of time. A blueprint is derived from this FloorSwitch C++ class.

Create another C++ class derived from Parent Class: AItem. This class will be used as pickup items for the Character which can heal the player or add coins. In the header file, OnOverlapBegin() and OnOverlapEnd() override functions are declared.

```
UCLASS()
class KAERMORHEN_API APickup : public AItem
{
    GENERATED_BODY()

public:

    APickup();

    virtual void OnOverlapBegin(UPrimitiveComponent* OverlappedComponent, AActor* OtherActor,
      UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult&
      SweepResult) override;

    virtual void OnOverlapEnd(UPrimitiveComponent* OverlappedComponent, AActor* OtherActor,
      UPrimitiveComponent* OtherComp, int32 OtherBodyIndex) override;

    UFUNCTION(BlueprintImplementableEvent, Category = Pickup)
    void OnPickupBP(class AMain* Char);
};
```

OnPickupBP() function is designed to be overridden (implemented) in blueprint. Derive a blueprint from Pickup class for Coins.

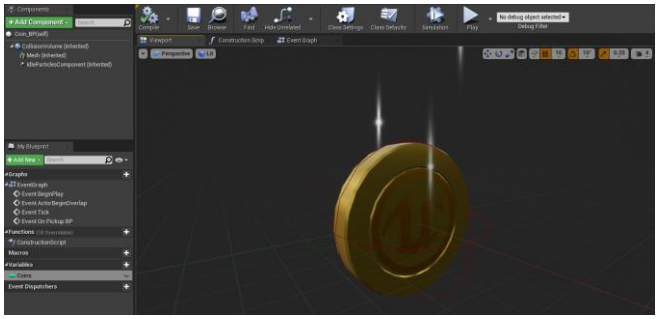

Fig 8: Static mesh for the coin.

Set the Static Mesh Component for Coins and add the IdleParticlesComponent, OverlapParticles and OverlapSound for it in blueprints. Create a new variable in Event Graph called Coins and set the variable type as Integer. This will increment the Coins count by 1 when the Character overlaps with the Pickup Item. The Increment Coins function declared in Main.h and defined in Main.cpp which is the Character class.

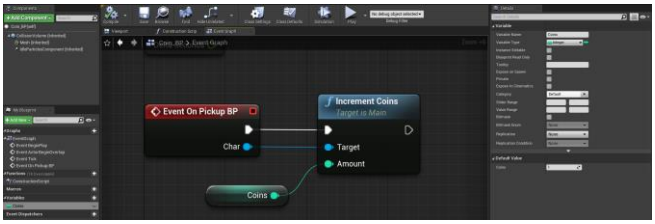

Fig 9: Increment logic.

In the header file, the function is set as BlueprintCallable which allows the function to be called from blueprints. It takes an amount which is integer with value 1. In the Main.cpp file, the function simply increases the Coins amount by 1.

```
void AMain::IncrementHealth(float Amount) {
    if (Health + Amount >= MaxHealth) {
        Health = MaxHealth;
    } else {
        Health += Amount;
    }
}
```

Create a new C++ class derived from the Parent Class: APlayerController. APlayerController class is derived from AController. For HUD creation and use of Unreal Motion Graphics, add UMG module in the Build.cs file. Slate UI is also essential and must be added. In the newly derived C++ class called MainPlayerController class, declare TSubclassOf <UUserWidget> to be selected in blueprints and the UUserWidget* object itself.

```
/** Reference to the UMG asset in the Editor. */
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Widgets)
TSubclassOf<class UUserWidget> HUD_OverlayAsset;

/** Variable which holds the widgets after creation. */
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Widgets)
UUserWidget* HUD_Overlay;
```

The user widget is extensible by users through the WidgetBlueprint. In the MainPlayerController.cpp file, if HUD_OverlayAsset is set in blueprints and is valid, create the HUD_Overlay with CreateWidget<> function.

```
if (HUD_OverlayAsset) {
    HUD_Overlay = CreateWidget<UUserWidget>(this, HUD_OverlayAsset);
}
HUD_Overlay->AddToViewport();
HUD_Overlay->SetVisibility(ESlateVisibility::Visible);
```

In UE4, create a new Widget Blueprint for User Interface, delete the Canvas panel and add Vertical boxes to align the Health and Stamina bars at the top left. Add some more boxes to display the Coins count in the bottom right. Name the boxes appropriately for easier usage. This is the backbone HUD_OverlayAsset which will further consist of the HUD_Overlay like HealthBar, StaminaBar, Coins, HostileHealthBar and PauseMenu.

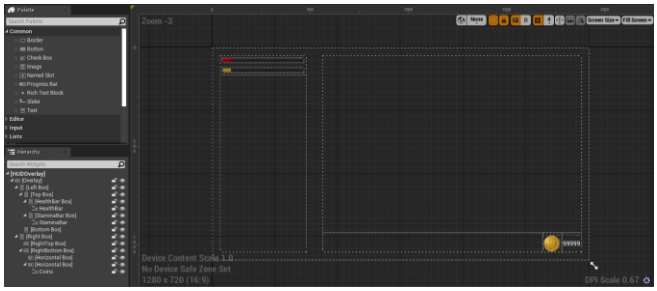

Fig 10: UI Widget blueprint.

3.4. Combat Mechanics.

Create a new C++ class derived from Parent Class: AItem. In the header file, declare a USkeletalMeshComponent* variable which is used to create an instance of an animated SkeletalMesh asset. The behavior of audio playback is defined within Sound Cues. bWeaponParticles is used for particle effects implementation while UBoxComponent* CombatCollision is a box component which will be on the weapon mesh to activate and deactivate collision for it.

```
UPROPERTY(VisibleAnywhere, BlueprintReadWrite, Category = SkeletalMesh)
class USkeletalMeshComponent* SkeletalMesh;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Item | Sound")
class USoundCue* OnEquipSound;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Item | Sound")
USoundCue* SwingSound;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Item | Particles")
bool bWeaponParticles;

UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Item | Combat")
class UBoxComponent* CombatCollision;
```

In the Weapon.cpp class file, set SkeletalMesh and attach it to the RootComponent. Similarly, for CombatCollision.

```
AWeapon::AWeapon() {
    SkeletalMesh = CreateDefaultSubobject<USkeletalMeshComponent>(TEXT("SkeletalMesh"));
    SkeletalMesh->SetupAttachment(GetRootComponent());

    CombatCollision = CreateDefaultSubobject<UBoxComponent>(TEXT("CombatCollision"));
    CombatCollision->SetupAttachment(GetRootComponent());

    bWeaponParticles = false;
```

Back in UE4, open the Character's skeleton and add a socket under the RightHand, naming it "RightHandSocket" which is used in the code. This particular socket is used to attach the weapon into the hand of the Character. Once attached, this weapon will move in any direction according the bones created in the Character's skeleton.

Fig 11: Sword attachment to skeleton.

A new Animation Montage is created in UE4, adding various kinds of attack animation for the Character when equipped with a weapon.

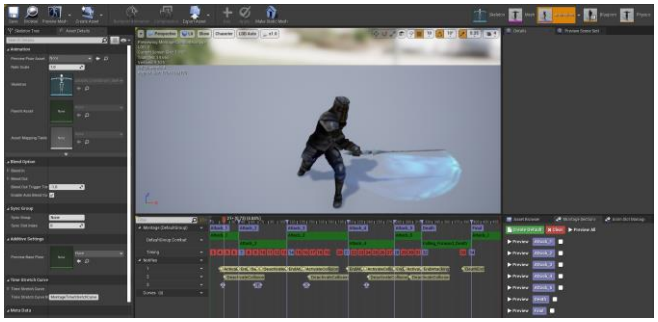

Fig 12: Animation Montage for attack anims.

Each attack animation is using notifies. SwingSound notify is used to play the SwingSound for the weapon. ActivateCollision and Deactivate Collision notify fires the relevant function from C++. Notifies are events which are fired when the animation reaches that specific frame in time. EndAttacking notify is added before the animation ends. Each animation is separated by a montage section. Alongside, a new blueprint is derived from AHostileNPC C++ class. The SkeletalMesh is set, Spheres are tweaked to fit the mesh accordingly and CapsuleComponent is tweaked to further suit collision events with other Actors.

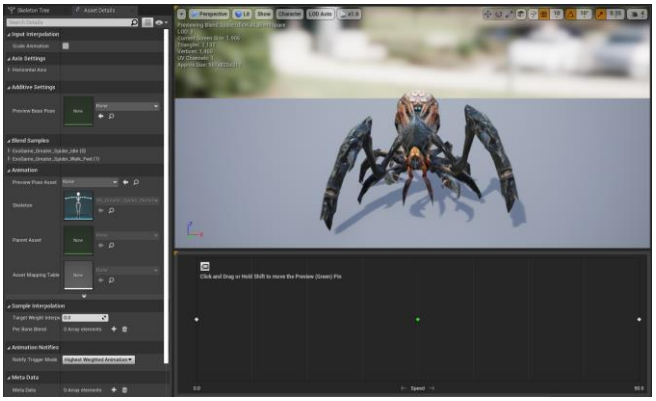

Fig 13: Skeletal mesh for the Hostile class.

Create a Widget Blueprint for Hostile's HealthBar. Delete the canvas panel and add a horizontal box with a progress bar. Bind a function to it, create a new variable for RefToMain which is the same as RefToMainChar variable created for other widget blueprints. Similar to Main's widget, check if RefToMain is valid, if not, get the Owning Player Pawn and cast it to Main_BP. Set the cast as RefToMain and return.

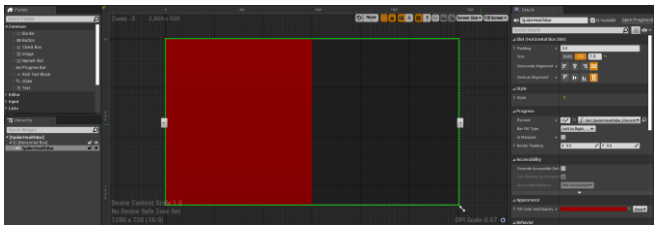

Fig 14: Health widget for Hostile class.

3.5. Extending the Hostile class.

Based on the AHostileNPC C++ class, new blueprints are derived for different types of hostiles.

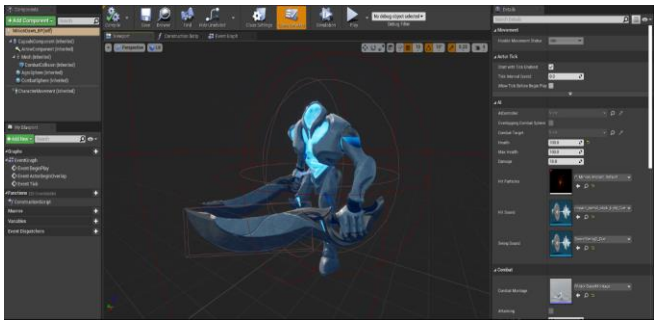

Fig 15: Extension of Hostile class with minions.

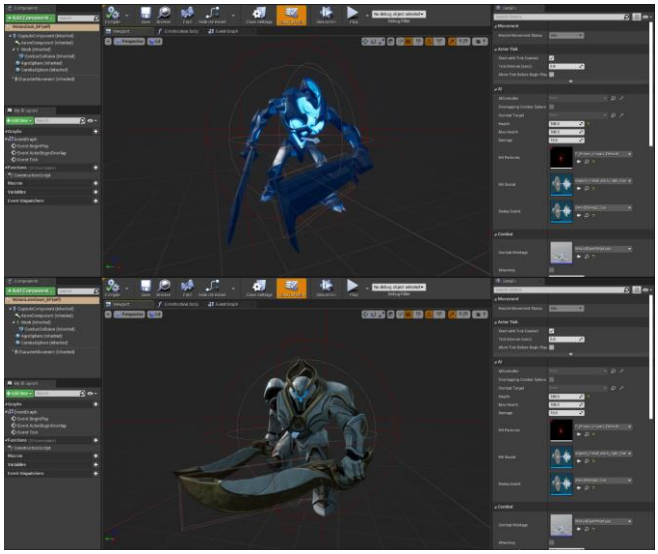

Fig 16: Extension of Hostile class with minions.

Each Hostile is given separate SkeletalMesh but all 3 use the same skeleton. Their CombatSphere, CombatCollision and CapsuleComponent is tweaked to better fit their mesh. Add sockets LeftAttackSocket and LeftParticlesSocket for the Hostile's skeleton for CombatCollision and HitParticles.

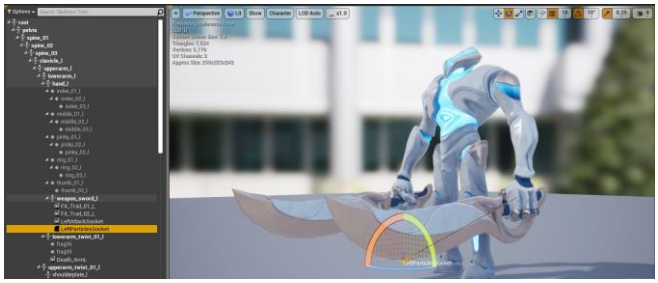

Fig 17: Attaching hitpoint for sword.

Create a new animation blueprint for this skeleton and create a clone of the EventGraph from previous SpiderAnim_BP. In the AnimGraph, create a clone of the StateMachines as well.

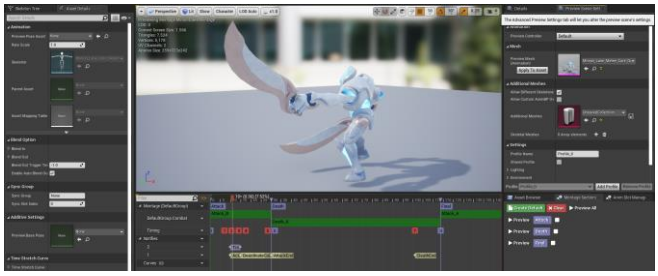

Fig 18: Animation montage for attack anims.

Create Animation Montage for it and add similar notify events.

3.6. Extending the Weapon class.

Based on AWeapon C++ class, derive various blueprints for different kinds of weapons to be added into the world. Set the SkeletalMesh, IdleParticlesComponent for each. Tweak all CombatCollision since different weapons have different skeletal mesh and different skeletons as well. Add socken WeaponSocket to the appropriate skeleton files to enable HitParticles during attacking animations with HostileNPCs.

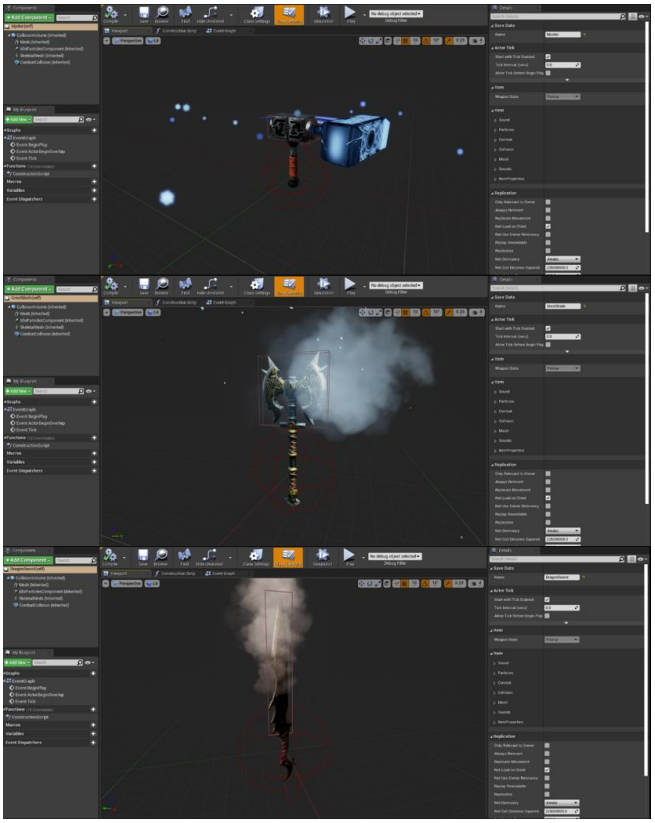

Fig 19: Adding weapons with dynamic effects.

3.7. Adding the Weapon Trail effects.

For the Hostiles and Character who have access to weapon like swords, create two sockets for each at the hand itself and then at the tip of the weapon for adding trail effect.

Fig 20: Sword trail effects for character.

Create a trail in the animation montage and then use the sockets created with the PS template. This adds a trail effect when the Character swings their sword.

Fig 21: Trail points for the effect.

Similarly, for the Hostile's animation montage and the skeleton.

Fig 22: Trail effect for minions.

3.8. Switching Levels: Level Transition Volume.

Create a new C++ class derived from the Parent Class: AActor. In the header file, declare a UBoxComponent* TransitionVolume which will be used to load the next level when the Main Character overlaps with it.

```
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Transition)
class UBoxComponent* TransitionVolume;

class UBillboardComponent* Billboard;

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = Transition)
FName TransitionLevelName;

protected:
// Called when the game starts or when spawned
virtual void BeginPlay() override;

public:
// Called every frame
virtual void Tick(float DeltaTime) override;

UFUNCTION()
virtual void OnOverlapBegin(UPrimitiveComponent* OverlappedComponent, AActor* OtherActor,
UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult&
SweepResult);
```

UBillboardComponent* is a 2d texture that will be rendered always facing the camera. This is useful to select and tweak the TransitionVolume in the viewport while setting it up in the world. FName TransitionLevelName is used to set the next level name. Fname is a public name, available to the world. Names are stored as a combination of an index into a table of unique strings and an instance number. OnOverlapBegin() function is declared for overlap functionality.

```
// Sets default values
ALevelTransitionVolume::ALevelTransitionVolume()
{
    // Set this actor to call Tick() every frame.  You can turn this off to improve performance if you
      don't need it.
    PrimaryActorTick.bCanEverTick = false;

    TransitionVolume = CreateDefaultSubobject<UBoxComponent>(TEXT("TransitionVolume"));
    SetRootComponent(TransitionVolume);

    Billboard = CreateDefaultSubobject<UBillboardComponent>(TEXT("Billboard"));
    Billboard->SetupAttachment(GetRootComponent());

    TransitionLevelName = "SunTemple";
}
```

Inside the constructor, setup TransitionVolume as the RootComponent and setup Billboard and attach it to the RootComponent. TransitionLevelName is set to "SunTemple" by default which is the current LevelName.

```
// Called when the game starts or when spawned
void ALevelTransitionVolume::BeginPlay()
{
    Super::BeginPlay();

    TransitionVolume->OnComponentBeginOverlap.AddDynamic(this, &ALevelTransitionVolume::OnOverlapBegin);
}
```

In BeginPlay(), use AddDynamic function to map the OnOverlapBegin function for OnComponentBeginOverlap.

```
void ALevelTransitionVolume::OnOverlapBegin(UPrimitiveComponent* OverlappedComponent, AActor* OtherActor,
UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult) {
    if (OtherActor) {
        AMain* Main = Cast<AMain>(OtherActor);
        if (Main) {
            Main->SwitchLevel(TransitionLevelName);
        }
    }
}
```

In OnOverlapBegin(), cast OtherActor to AMain if it is valid. If the cast is successful, call the SwitchLevel() function defined in Main class and passing the TransitionLevelName to it. Create a blueprint derived from LevelTransitionVolume C++ class.

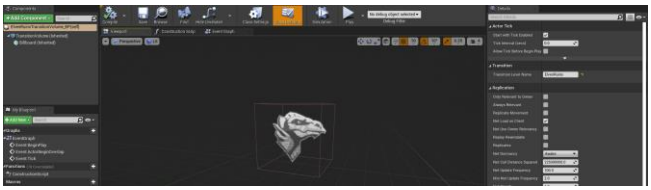

Fig 23: Placeholder for the transition volume.

Set the LevelTransitionName in blueprint to the next level name to switch to. Now back in Main class, declare the SwitchLevel() function.

```
void AMain::SwitchLevel(FName LevelName) {
    UWorld* World = GetWorld();
    if (World) {
        FString CurrentLevel = World->GetMapName();

        FName CurrentLevelName(*CurrentLevel);
        if (CurrentLevelName != LevelName) {
            UGameplayStatics::OpenLevel(World, LevelName);
        }
    }
}
```

3.9. Saving and Loading the game.

Create a new C++ class derived from the Parent Class: USaveGame. In the header file, create a struct for storing Character related data and other information which can be used for saving and loading the game.

```
USTRUCT(BlueprintType)
struct FCharacterStats {
    GENERATED_BODY()

    UPROPERTY(VisibleAnywhere, Category = SaveGameData)
    float Health;

    UPROPERTY(VisibleAnywhere, Category = SaveGameData)
    float MaxHealth;

    UPROPERTY(VisibleAnywhere, Category = SaveGameData)
    float Stamina;

    UPROPERTY(VisibleAnywhere, Category = SaveGameData)
    float MaxStamina;

    UPROPERTY(VisibleAnywhere, Category = SaveGameData)
    int32 Coins;

    UPROPERTY(VisibleAnywhere, Category = SaveGameData)
    FVector Location;

    UPROPERTY(VisibleAnywhere, Category = SaveGameData)
    FRotator Rotation;

    UPROPERTY(VisibleAnywhere, Category = SaveGameData)
    FString WeaponName;

    UPROPERTY(VisibleAnywhere, Category = SaveGameData)
    FString LevelName;
};
```

Back in the Main class header file, declare functions from saving and loading.

```
UFUNCTION(BlueprintCallable)
void SaveGame();

UFUNCTION(BlueprintCallable)
void LoadGame(bool SetPosition);

void LoadGameNoSwitch(bool SetPosition);

UPROPERTY(EditDefaultsOnly, Category = SaveData)
TSubclassOf<class AItemContainer> WeaponContainer;
```

In the Main class file, for SaveGame(), use CreateSaveGameObject() which creates a new, empty SaveGame object to set data on and then pass to SaveGameToSlot. StaticClass() is passed as an argument to it which returns a [UClass] object representing this class at runtime. The returned object from CreateSaveGameObject is cast to USave and assigned to SaveGameInstance variable.

It is essential to give these variables the category, SaveGameData. In the class header file, declare a FString PlayerName, uint32 UserIndex which is a 32-bit unsigned integer. FCharacterStats variable CharacterStats is declared. FCharacterStats is the struct variable. F prefix is used in UE4 C++ for struct.

3.10. User Interface: Pause Menu.

Create a new Widget Blueprint for PauseMenu. Use the Canvas Panel to anchor the menu in the bottom left of the screen. Use button in Palette to create 4 buttons.

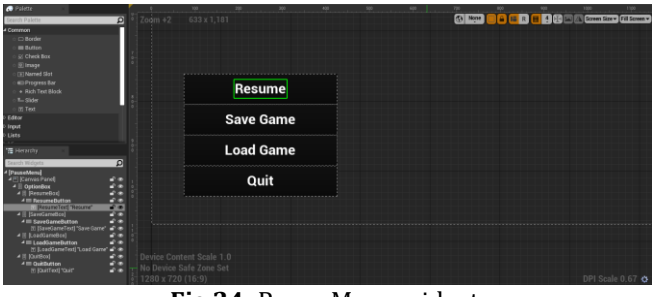

Fig 24: Pause Menu widget.

Create a new animation for the OptionBox and in the track, choose OptionBox. Animation is set for PauseMenu which slides the PauseMenu from out of the screen and into the viewport in a time of 0.25 secs. Hitting ESC will bring up the pause menu which will pause the game in place. Resume button will resume it or pressing ESC again. SaveGame fires up the SaveGame() C++ function, similarly, LoadGame fires up the LoadGame() C++ function. Quit button will quit the game directly.

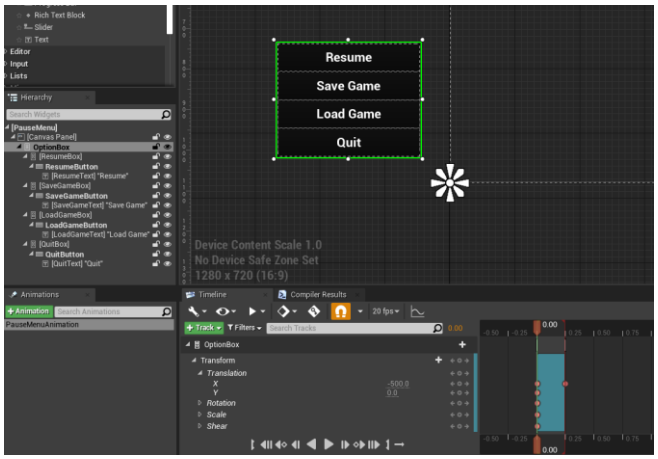

Fig 25: Transition effect for the menu.

For the Event Graph, create event nodes for the buttons.

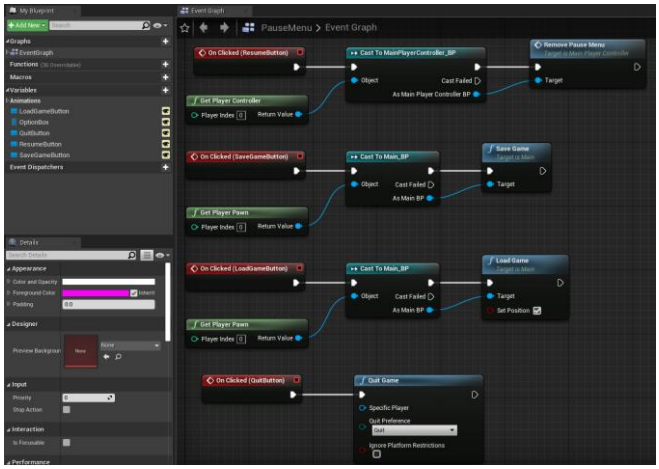

Fig 26: Event nodes for the pause menu buttons.

Get the PlayerPawn and cast it to MainPlayerController_BP which is a blueprint derived from AMainPlayerController class. Relevant functions are called for events fired. This implements a PauseMenu for Resume, Save Game, Load Game and Quit functionalities.

The game is then ready to play.

4. RESULTS

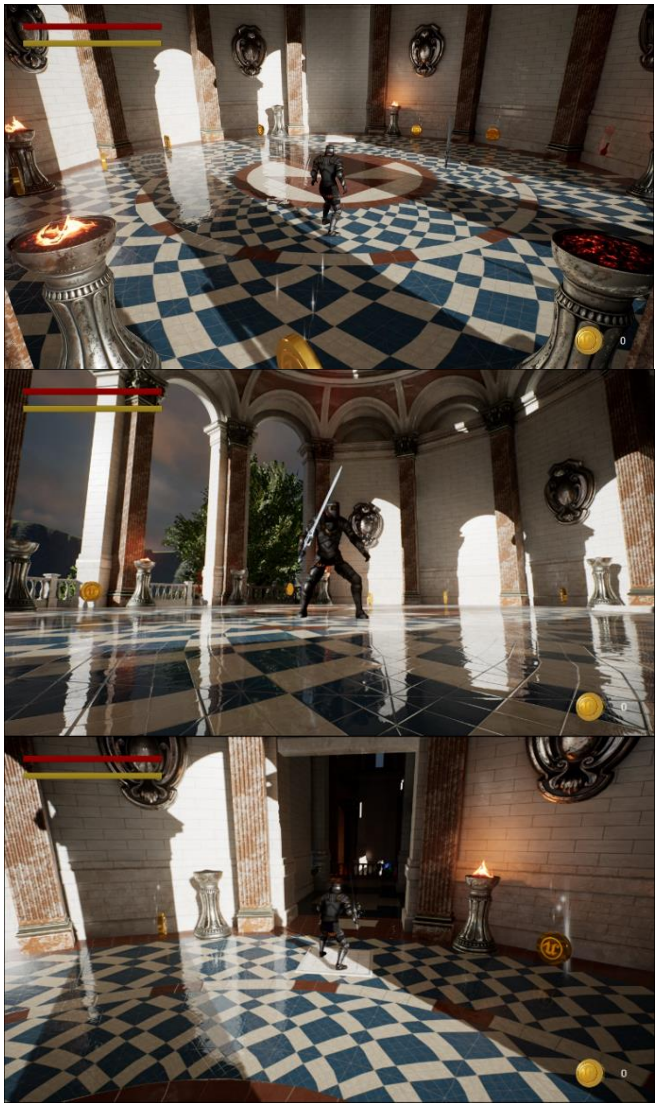

Fig 27: Spawn location of the character.

Fig 28: Enemy character.

Fig 29: Animated flaming sword and health potion.

Fig 30: Level completion screen.

5. CONCLUSION

Artificial Intelligence adds realism to the gaming experience and make the game more powerful and meaningful. This RPG game is a prototype of using AI in Unreal Engine 4. Here, this game simulates combat from a third-person perspective and player will encounter opponent waves they need to survive in order to progress. AI can bring the revolution in the gaming industry. This project shows the basic use of AI in a game. Unreal Engine consist of many different functions which can be used in implementing advanced AI.

6. REFERENCES

[1] Bogost, Ian (March 2017). ""Artificial Intelligence" Has Become Meaningless". Retrieved 19 May 2020.
[2] Kaplan, Jerry (March 2017). "AI's PR Problem". MIT Technology Review.
[3] Eaton, Eric; Dietterich, Tom; Gini, Maria (December 2015). "Who Speaks for AI?" (PDF). AI Matters.
[4] Good, Owen S. (5 August 2017). "Skyrim mod makes NPC interactions less scripted, more Sims-like". Polygon. Retrieved 19 May 2020.
[5] Lara-Cabrera, R., Nogueira-Collazo, M., Cotta, C., & Fernández-Leiva, A. J. (2015). Game artificial intelligence: challenges for the scientific community.
[6] Yannakakis, G. N. (2012, May). Game AI revisited. In Proceedings of the 9th conference on Computing Frontiers (pp. 285–292). ACM.

[7] Hagelback, Johan, and Stefan J. Johansson. "Dealing with fog of war in a real-time strategy game environment." In Computational Intelligence and Games, 2008. CIG'08. IEEE Symposium On, pp. 55-62. IEEE, 2008.
[8] Abd Algfoor, Zeyad; Sunar, Mohd Shahrizal; Kolivand, Hoshang (2015). "A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games". International Journal of Computer Games Technology. 2015: 1–11. doi:10.1155/2015/736138.
[9] Yap, Peter. "Grid-based path-finding." In Conference of the Canadian Society for Computational Studies of Intelligence, pp. 44-55. Springer, Berlin, Heidelberg, 2002.
[10] Sturtevant, N. R. (June 2012). "Benchmarks for Grid-Based Pathfinding". IEEE Transactions on Computational Intelligence and AI in Games. 4 (2): 144–148. doi:10.1109/TCIAIG.2012.2197681.
[11] Goodwin, S. D., Menon, S., & Price, R. G. (2006). Pathfinding in open terrain. In Proceedings of International Academic Conference on the Future of Game Design and Technology.
[12] Nareyek, A. (2004). AI in computer games. Queue, 1(10).
[13] Cui, X., & Shi, H. (2011). A*-based pathfinding in modern computer games. International Journal of Computer Science and Network Security, 11(1), 125-130.
[14] "Design Techniques and Ideals for Video Games". Byte Magazine. 7 (12). 1982. p. 100.