EAD Prototypen Beschreibung

Project Overview

A prototype for a rogue-like top-down video game has been developed. The game sets place on a single map which is played over and over again, but the AI changes with the use of a reinforcement model with every new start of the map. If the player kills all enemies a door opens where the map restarts when the player walks through. I the player is killed the map also restarts.

The player can be controlled by Gamepad or Keyboard and Mouse. The mouse or the right analog stick controls the direction the player is facing. The left analog stick or the Keys WASD control the movement. Switching between a sword and a bow is down with the keyboard key ‘Q' or the upper Gamepad button ('Y'). The right trigger or the left mouse button initiates an attack.

Enemies spawn with different weapons, attack strategies and items based on a reinforcement model and their configuration gets rewarded based on the damage they inflict.

Proper material

The materials **[M_Assyrian_Hair]**, **[M_Assyrian_Skin]**, **[M_Assyrian_Cloth]** and **[M_Assyrian_Belt]** were assigned to the skeletal mesh **[SK_Assyrian].** These are simple color materials and are used for the enemies.

The materials **[M_ProjectileHitParticle]** and **[M_ProjectileLaunchParticle]** are respectively used for the particle effects for hitting and launching a projectile.

[M_Rock_Sandstone] is used for some of our dungeon assets to give them a more suitable look.

Light setup

A lightmass importance volume is put around the whole room to concentrate the light in the room and directly around it.

On the player character **[BP_PlayerCharacter]** a point light has been added to make the surrounding of the player brighter. “Cast Shadows” is deactivated to prevent irritating shadows from the player itself. Also, the specular part was set to zero and the light color was adjusted.

Four point lights acting as torches are placed in the world. The light source radius and length are set to be similar to the size of the flame in the sconce, emulating the light the flames would throw. Color and attenuation are set to light the room in a warm color. Intensity is slightly randomized to give a flickering flame effect.

Additionally, a weak directional light is used for slight visibility in places the point lights don’t reach.

Finally, a spotlight is placed behind the door to the next level, which activates when the door opens up. This light shines from behind the door into the room with a green light, which is easily visible, and signals the players that the door is now open.

User Interface

HUD

Our main HUD element is the player’s health bar. It is created with a Widget Blueprint that gets created and added to the viewport in the player character BegonPlay event. The player character also subscribes to an event in the **[BaseHealthComponent]** that gets called when it receives damage. When the event fires the health bar widget in **[WB_HealthBar]** gets updated with the current health percentage.

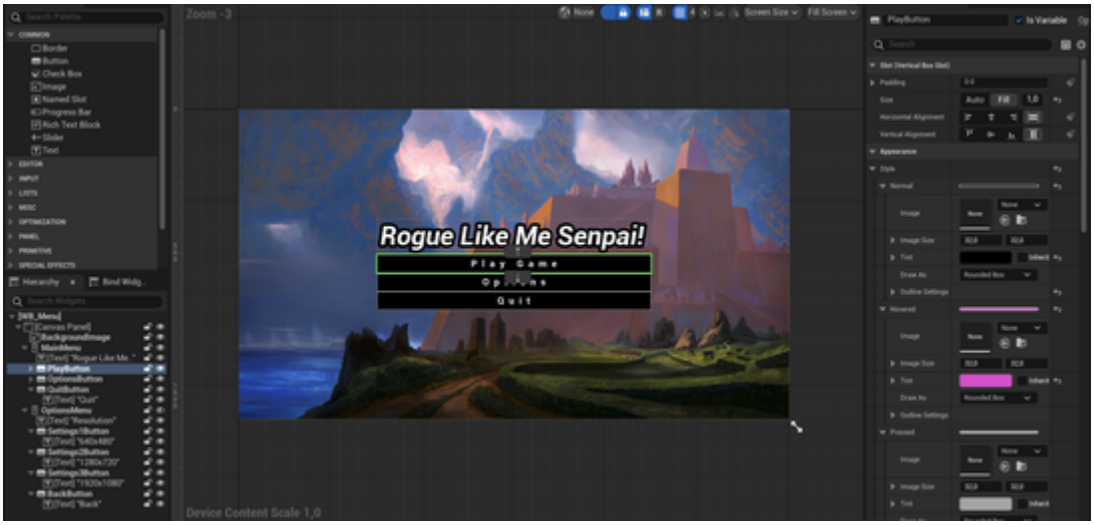
Additionally, we applied the same technique to the enemy. The main difference is that the enemy’s health bar is displayed on a Widget Component **[BP_EnemyHealthBarComponent]** using **[WB_HealthBar_Enemy]** that is located above the enemy's head. This widget component also rotates toward the inverse camera’s forward direction so that is always visible.

Both widgets can be found in the **Core UI** folder.

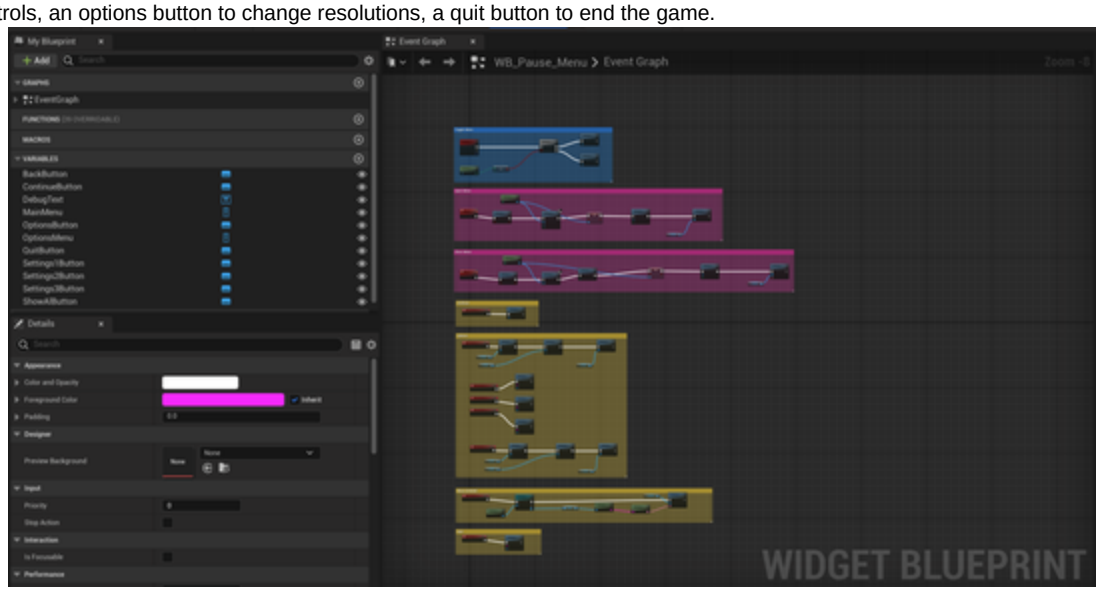
Menu

We added two menus to the game: a start menu and an in-game pause menu.

The start menub **[WB_Menu]** is the first screen the player reaches when opening the game and it contains: a background image, the title of the game, a play button to start the game, an options button to change resolutions, a quit button to to end the game.



The in-game pause menu **[WB_Pause_Menu]** can be opened in the game via the “p” key or the “start” button on a gamepad. When it opens, the whole game pauses and the menu is being focused. It contains: the title of the game, a continue button to unpause the game and reactivate the gameplay controls, an options button to change resolutions, a quit button to end the game.



Both menus can be used via mouse, keyboard or gamepad.

Particles

We use the Niagara Particle System which is the default in UE5. Specifically, each weapon in the **Core Weapons** folder has a particle effect that gets triggered when its “Play Attack Effects” function is called. For Melee Weapons this happens when the “Attack“ function is called. For Ranged Weapons, it gets called after the Projectile Actor is spawned. The Projectile Actors themselves also have a Particle effect that gets triggered when it overlaps with an enemy.

Red Particle effects mean that damage has been dealt (Melee hit or Projectile) while blue particles mean that a projectile has been spawned. For both effects, we created a Material that adds or subtracts a noise value from the color.

The particle effect variables are defined in BP_BaseParticle and in BaseWeapon in cpp. Both are set in Blueprint.

The items **[BP_DamageBoost]**, **[BP_HealthBoost]** and **[BP_MovementBoost]** which inherit from **[BP_Item_Boost]** and are assigned to the enemies. They have the particle effects **[NS_BoostEffect_Damage]**, **[NS_BoostEffect_Health]** and **[NS_BoostEffect_Movement]** attached. The template particle effect HangingPartices has been used with the color and particle lifespan modified.

Audio

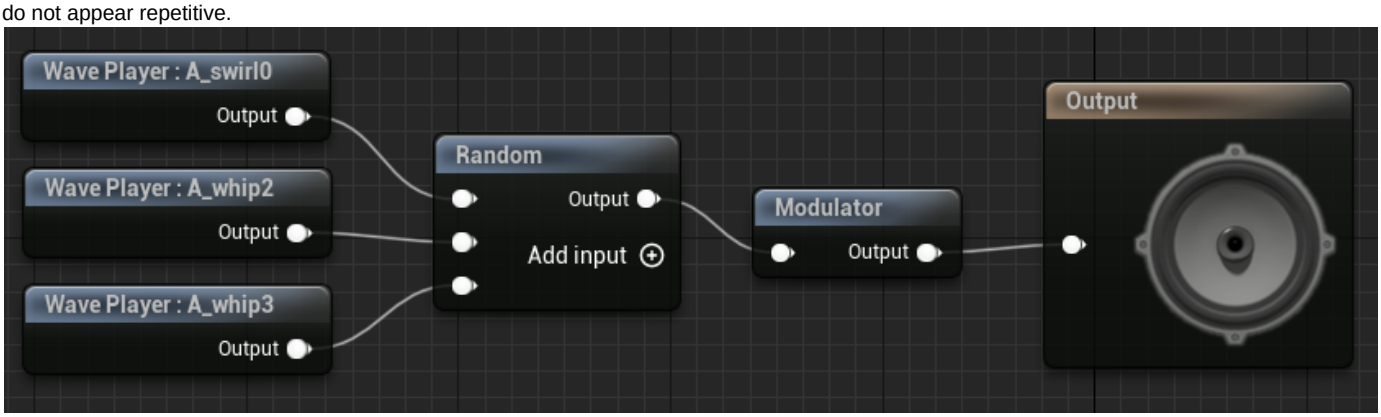
Background music

Two different songs start playing in the background: while in the menu **[A_Background_Menu_Cue]** and during actual gameplay **[A_Background_Level_Cue]**. Both songs were taken off Youtube, converted to WAVE files and slightly edited in Cubase12.

Sound effects

All sound effects are royalty-free foleys off the internet, adapted within Cubase12 if needed, and imported into Unreal Engine as WAVE files.

[A_whoosh_Cue] plays one of three distinct whipping sounds at random and modulates it. By randomizing and modulating, consecutive sounds do not appear repetitive.



The cue **[A_whoosh_Cue]** is timed with a Notify Track within the punching animation for the player character to play at the right time.

[A_Footsteps_Cue] is built analogous to the whoosh cue (i.e. three sounds, randomized and modulated). It is then timed with a Notify Track within the walking and running animations to play at each footstep.

[A_gore_impact_Cue] plays when a character (enemy or player) receives their lethal blow.

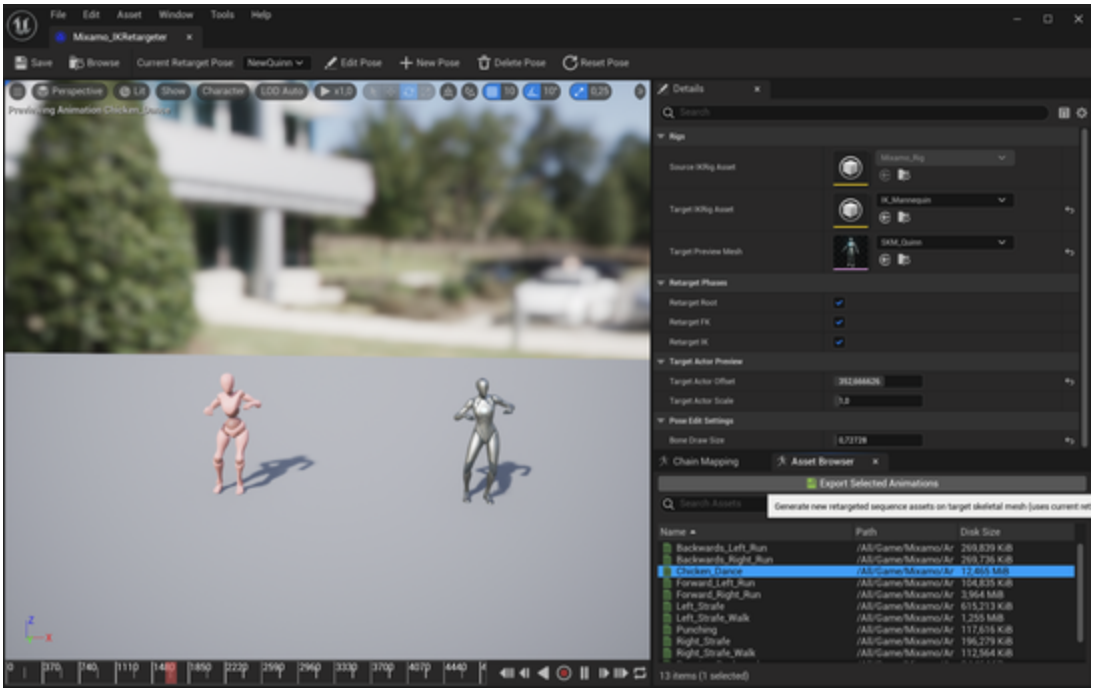
[A_drop_on_floor_Cue] is timed when the player character hits the floor during their death animation.

Animations

Player Animation

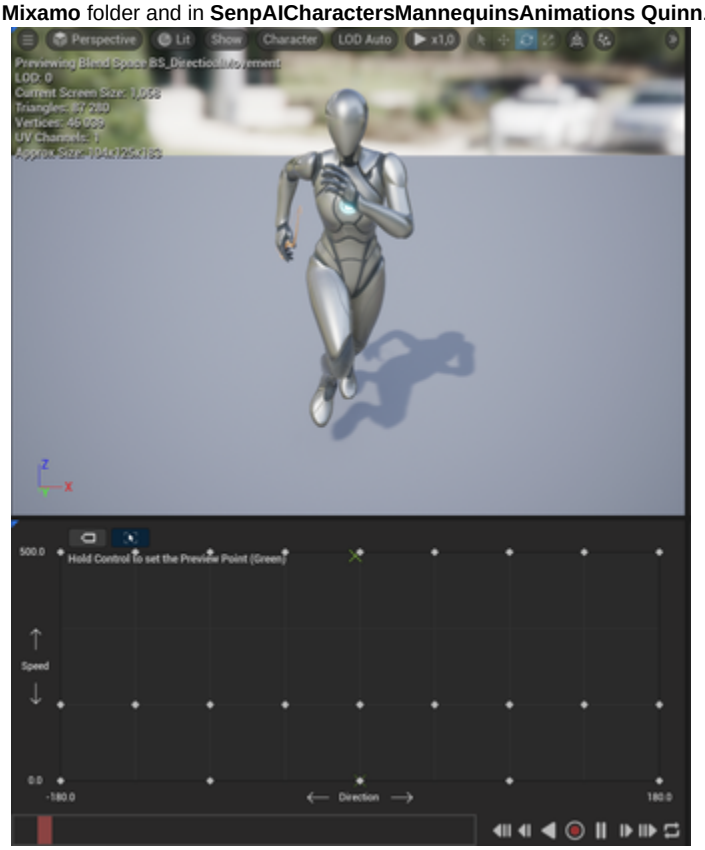
The player animations use mixamo animations (https://www.mixamo.com/) on the unreal's mannequin skeleton. We are using: a hit animation, a death animation, strife animations, a run backward animation, and other locomotion animations.

To make mixamo animations work with the mannequin's skeleton we had to add an Inverse Kinematics Retargeter **[Mixamo_IK_Retargeter]**. In this retargeter we had to map the mannequin skeleton to the mixamo skeleton. Once this was done we were able to export each individual mixamo animation for unreal's mannequin.



To actually use the animations in the game we had to update the animation blueprint **[ABP_Manny]** to play the death, & directional movement animations **[BS_DirectionalMovement]**.

The animations can be found in the **Mixamo** folder and in **SenpAICharactersMannequinsAnimations Quinn**.



Enemy Animation

The enemy animations also use mixamo animations and rigging with a modified human model (https://opengameart.org/content/very-low-poly-human). The animations include: Arrow shooting, Death, Idle, Melee Attack and Run.

The animation blueprint **[ABP_Assyrian]** enables movement with the blendspace [**BS_Assyrian_Movement]**. The attack animation montages **[AM_Assyrian_ShootingArrow]** and **[AM_Assyrian_StandingMeleeAttackHorizontal]** are mixed with the movement using the slot UpperBody and the skeleton's bone "Spine".

AI-controlled actors/pawns/characters

Reinforcement model

We developed and implemented an AI model, **ReinforceModel,** which is loosely derived from classical reinforcement algorithms. Its purpose is to create enemy configurations. Depending on the enemies' performance against the player the model receives a reward/penalty. The model is built in C++ as an Actor Component. The following end-points are available within Unreal Engine (all internal functionality is hidden)

```
UCLASS( ClassGroup=(Custom), meta=(BlueprintSpawnableComponent) )
class SENPAI_API UReinforceModel : public UActorComponent
{
    GENERATED_BODY()

public:
    // Sets default values for this component's properties
    UReinforceModel();

    UFUNCTION(BlueprintCallable, Category = "AImodel")
    FString createEnemy();

    UFUNCTION(BlueprintCallable, Category = "AImodel")
    TArray<int> enemyToIntArray(const FString enemy) const;

    UFUNCTION(BlueprintCallable, Category = "AImodel")
    void giveReward(const FString enemyIn, double reward);

    UFUNCTION(BlueprintCallable, Category = "AImodel")
    TMap<FString, double> getQtable() const;

    UFUNCTION(BlueprintCallable, Category = "AImodel")
    void setQtable(const TMap<FString, double> Qtable);

    UFUNCTION(BlueprintCallable, Category = "AImodel")
    FString printQtable() const;
```

createEnemy() builds an enemy according to the current state of the model and returns it as a String of 1s and 0s. **enemyToIntArray()** is a utility function, which splits the String into an array - each entry in this array corresponds to one enemy modification (see below)

giveReward() feeds rewards/penalties for a given enemy configuration back into the model.

Internally the model stores its state as a Map. **getQtable()** and **setQtable()** are used to retrieve the model's state, save it to a file (via **[BP_SaveGame_AI_model]**) and to later load it from there. This way, the AI model can be made persistent even after closing and re-opening the game.

printQtable() returns the model's state as a String in a somewhat interpretable way.

The current state of the AI model consists of three layers and a total amount of 8 possible modifications. The model always starts with a base enemy, which by itself cannot do anything. In each layer, the enemy gets modified by picking one out of a subset of predefined modifications.

- Layer 1: {melee, ranged, nothing}
  - changed which weapon the enemy is holding + behavior in tree (see *AI-controlled enemies*)
- Layer 2: {hit&run, stay at target}
  - changes behavior in tree (see *AI-controlled enemies*)
- Layer 3: {health boost, movement boost, damage boost, nothing}
  - buffs to the enemy

Based on the enemy's performance against the player it receives a reward/penalty, which makes the given configuration more/less likely to be chosen again. Starting at the last layer this reward propagates back until it reaches the root of the model.
The choice of modification in each layer is stochastic, i.e. the 'best decision' is taken most likely, but other options can be explored with lower probability.

AI-controlled enemies

The reinforcement model is attached to the **[BP_GameState_ReinforcementModel]** as a component to be accessed globally and persisted throughout the game. It is used to generate enemy configurations as an array of integer values.

[BP_BaseCharacter] serves as a parent class for **[BP_PlayerCharacter]** and **[BP_SimpleEnemy_Character]** which is a parent class for **[BP_Assyrian_Character]**.

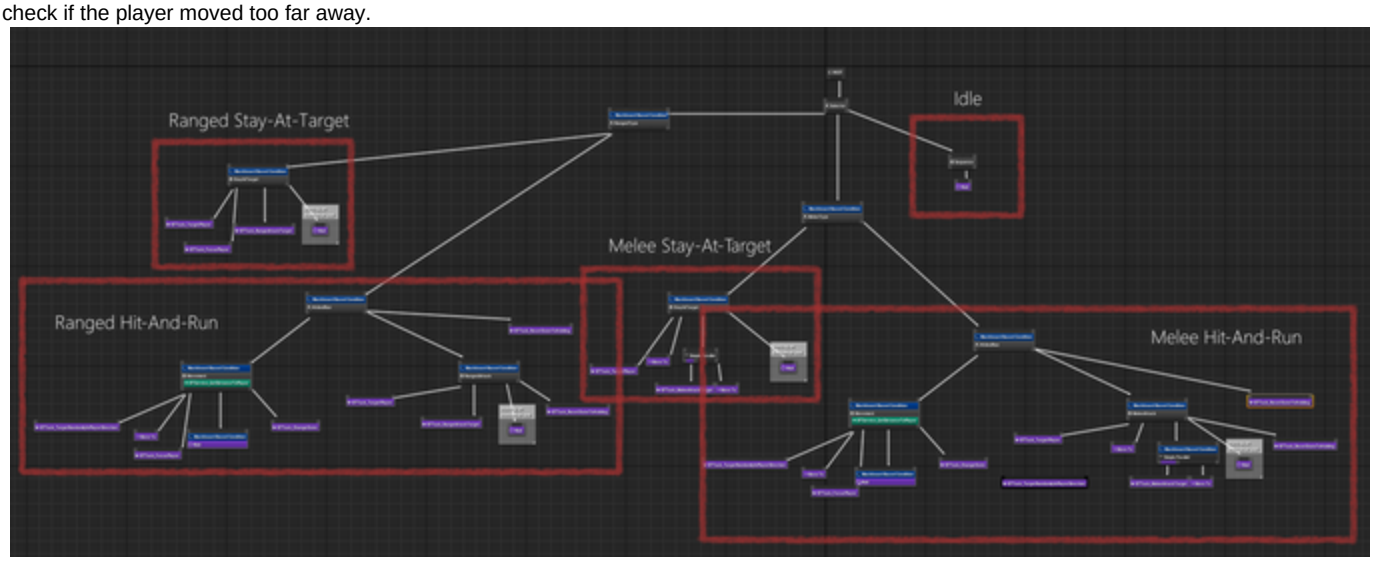
The AI controller **[AIC_SimpleEnemy]** is responsible for controlling instances of **[BP_SimpleEnemy_Character]** and therefore instances of the subclass **[BP_Assyrian_Character]**. The AI controller uses the behaviour tree **[BT_SimpleEnemy]** with the blackboard **[BB_SimpleEnemy]**.

Upon spawning the AI controller uses the reinforcement model to retrieve an enemy configuration to apply. It keeps track of the reward for the enemy configuration, which is fed back to the reinforcement model upon the end play event.

The enemy configuration is handled with the data table **[DataTable_ReinforcementModel_SimpleEnemy]** and its structure **[Structure_ReinforcementModel]**. The rows in the data table correspond to the elements of the enemy configuration array from the reinforcement model and therefore define the meaning for each element. The enum **[Enum_ReinforcmentModel_TagType]** distinguishes between blackboard keys and items. For items, the item class and the skeleton socket for the placement are defined. For blackboard keys the weapon class can be defined. In the ai controller's function **applyTag** the implementation for applying the enemy configuration can be found.

	Row Name	Name	Type	ItemClass	WeaponClass	SocketName
1	Melee	Melee	BLACKBOARD_KEY	None	BlueprintGeneratedClass'/Game/SenpAI/Core/Weapons/BP_BaseMeleeW	HandSocket_r
2	Ranged	Ranged	BLACKBOARD_KEY	None	BlueprintGeneratedClass'/Game/SenpAI/Core/Weapons/BP_BaseRangec	HandSocket_l
3	HitAndRun	HitAndRun	BLACKBOARD_KEY	None	None	None
4	StayAtTarget	StayAtTarget	BLACKBOARD_KEY	None	None	None
5	HealthBoost	HealthBoost	ITEM	BlueprintGeneratedClass'/Game/SenpAI/Core/Items/Boosts/BP_Healt	None	None
6	MovementBoos	MovementBoost	ITEM	BlueprintGeneratedClass'/Game/SenpAI/Core/Items/Boosts/BP_Move	None	None
7	DamageBoost	DamageBoost	ITEM	BlueprintGeneratedClass'/Game/SenpAI/Core/Items/Boosts/BP_Dama	None	None

The behavior tree distinguishes between ranged, melee and idle behavior using the blackboard. For each behavior type the attack strategies hit&run and stay-on-target are implemented. The task **[BTTask_TargetPlayer]** stores the player's location and sets focus on the player. The task **[BTTask_TargetRandomlyInPlayerDirection]** finds and stores a random location in front of the player. **[BTTask_RangeAttackTarget]** and **[BT_MeleeAttackTarget]** call the ai controller which delegates it to the character that plays the animation and triggers the attack for the weapon. The service [**BTService_GetDistanceToPlayer]** updates the DistanceToPlayer blackboard key, which is used in the Melee Hit&Run behavior to check if the player moved too far away.



Combat System

We implemented a simple combat system that is used by both the player and enemy characters.

It consists of a **[BaseHealthComponent]** implemented in C++ that takes care of all the damage that is sent by the built-in Unreal damage system. On reaching zero health it broadcasts an OnDeath Event that can be used by other game systems. It also broadcasts an event directly after the damage has been applied that is mainly used for updating the health bars.

The main combat interaction is done via Weapon actors that inherit from a c++ **[BaseWeapon]** class. We implemented one ranged weapon **[BP_BaseRangedWeapon]** and one melee weapon **[BP_BaseMeleeWeapon]**. Melee weapons apply their damage to the first non-owner character that they overlap with during the animation. Ranged weapons spawn a projectile actor **[BP_BaseProjectile]** from a specified socket location in the weapon's mesh during the attack animation. When the projectile overlaps it deals damage and destroys itself afterward.

Weapon actors spawn at socket locations defined in the Skeletal Mesh on BeginPlay. The player can also switch weapons during gameplay.

Lessons Learned

IK-Retargeter

If you want to apply an animation which was created on a different skeleton (e.g. Mixamo) to your current skeleton (e.g. Unreal Mannequin), you can use the Inverse Kinematics Retargeter, in which you map the bones to the new animation and then export the animation for your current skeleton. See also this nice tutorial: https://www.youtube.com/watch?v=JXBGw-6PpCE

Nested Arrays in Unreal

Do not work out of the box. You have to create a structure with an array inside and put that into an array.

Plan well

Use Events over Delays. 😉

Game Instance vs. Game State

Clean up Asset Redirectors

After moving or renaming assets in the asset browser Unreal creates redirections that can be found with the original name in the original location so that dependencies don't break. These redirectors can be "cleaned up" so that the old paths to these assets get changed to point to the new asset location/name.

Time Invested

	Time invested	**Tasks**
Christian Clemenz	45h	Materials, User Interface, Particles, Combat System, Project Setup
Maximilian Deutsch	58h	Materials, Particles, Animations, AI-controlled enemies, Combat System, Project Management
Alexander Graf	30h	Materials, Light, Level
Florentin Rieger	50h	User Interface, Audio, Animations, Project Setup
Peter Schlagnitweit	46h	Audio, AI Reinforcement model