Game Development with Unreal Engine 4

Jani Hämäläinen

2020 Laurea

Laurea University of Applied Sciences

Game Development with Unreal Engine 4

Jani Hämäläinen
Business Information Technology

Bachelor’s Thesis
October, 2020

Laurea University of Applied Sciences **Abstract**
Business Information Technology
Bachelor's Degree in Business Information Technology

Jani Hämäläinen

Game Development with Unreal Engine 4

Year 2020 Number of pages 31

Thesis' purpose was to build a game prototype and describe its working parts. The thesis was a part of a bigger entity and it focused mainly on character animation, artificial intelligence and combat mechanics. This project didn't have an employer and, therefore, it was more guided by self-imposed interests towards gaming industry. The main goal was to deepen the writer's knowledge about Unreal Engine 4. This would also lead to better job opportunities in the gaming industry.

The theoretical section composes mainly of Unreal Engine 4 documentation. With the documentation, the terminology, which appears in this paper, is explained to the reader. The functional part of this thesis was built by using prototyping methodologies. The functional part describes how character animation, artificial intelligence and combat mechanics were implemented by using Unreal Engine 4 game engine.

As a product of this thesis project, an unfinished game prototype was built. The prototype has a few architectural problems, but it is deemed acceptable. Furthermore, the writer gained more knowledge about inner workings of Unreal Engine 4 game engine and C++ programming language became more familiar to program with.

Keywords: Game Development, Unreal Engine, Animation, Artificial Intelligence, Combat Mechanics

Laurea-ammattikorkeakoulu **Tiivistelmä**
Tietojenkäsittely
Tietojenkäsittelyn tradenomitutkinto (AMK)

Jani Hämäläinen
PeliKehitystä Unreal Engine 4 -pelimoottorilla
Vuosi 2020 Sivumäärä 31

Opinnäytetyön tarkoitus oli rakentaa pelin prototyyppi ja kuvata pelin toimivia osia. Opinnäytetyö oli osa suurempaa kokonaisuutta ja siinä keskityttiin pääosin pelihahmojen animaatioon ja tekoälyyn sekä hyökkäyksen implementointiin. Projektilla ei ollut toimeksiantajaa eli projektin rakennusprosessi oli itseohjattavaa toimintaa. Tavoitteena oli syventää projektin tekijän tietoa Unreal Engine 4 -pelimoottorista ja videopelialasta yleisellä tasolla, joiden seurauksena tekijän asema työmarkkinoilla myös paranisi.

Projektin teoreettinen tietoperusta koostui pääosin Unreal Engine 4 -dokumentaatiosta. Sen avulla selitetään lukijalle opinnäytetyössä käytetty keskeinen terminologia. Opinnäytetyön toiminnallinen puoli rakennettiin hyödyntämällä prototyyppimenetelmiä. Toiminnallisessa osassa kuvataan, miten prototyypin animaatio, tekoäly ja pelihahmojen hyökkäys rakennettiin Unreal Engine 4 -pelimoottoria käyttäen.

Projektin tuloksena syntyi keskeneräinen pelin prototyyppi, jossa on muutamia arkkitehtuurillisia ongelmia. Prototyypin animaatio, tekoäly ja hahmon hyökkäys -osioiden toiminnallisuudet ovat kuitenkin riittäviä. Projektin seurauksena prototyypin rakentaja kasvatti omaa tietämystään Unreal Engine 4 -pelimoottorista sekä C++ -kielellä ohjelmoinnista.

Asiasanat: Pelikehitys, Unreal Engine, Animaatio, Tekoäly, Taistelumekaniikat

Contents

1 Introduction

This project is about building a game prototype with Unreal Engine 4 (UE4). We won't be going into every part, but rather describe the main working parts. Furthermore, this project does not a have other demand than my interest towards game industry. The goal was to gain more knowledge about inner workings UE4 and computer programming in general. In future these two skills will help me to either create my game or to get a job from game industry.

1.1 Project's objective

The goal was to create a game prototype with combat mechanics. Combat mechanics include things like attacking, animations and AI behaviors. This paper focuses mainly on these three things. There are other parts that are included in project's combat mechanics, but these are omitted due to the length of this paper. The Other main goal is to create well-structured systems that are scalable and reusable. It would be nice to be able to add functionalities to this project without too much code refactoring. Code reusability enables faster prototype building for projects that are derived from this project.

2 Technologies and development methodologies

My game engine of choice was Unreal Engine (UE4). UE4 uses C++ as a programming language. However, C++ is not required to build things in UE4 – one can use UE4's visual scripting system, Blueprints. This project chose both C++ and Blueprints as developing tools.

UE4 can be integrated with Visual Studio by installing a package. Since this is easy to do my IDE (Integrated Development Environment) of choice was Visual Studio. Epic Games use Visual Studio as their IDE, so my choice was further supported.

2.1 Why Unreal compared to other game engines

Before this project I had familiarized myself with Unreal Engine 4. I had some knowledge of Unity as well. So, the choice was between Unity and UE4. There were couple of reasons compared to why Unreal instead of Unity: UE4 graphics look better, UE4 is open source and UE4 uses C++, whereas Unity C# (Matthew Palaje 2017; Creative Blog Staff 2019). However, Unity has other strengths. For instance, the learning curve for Unity is much lower. This is affected by the language C# being easier language compared to C++. Furthermore, Unity has more tutorials online. As of writing this paper (02.10.2020, a search in Udemy for Unreal Engine yields 2766 results, where for Unity the number is 6540 (Udemy 2020a; Udemy 2020b).

Unity is considered have lower learning curve and is probably more suitable for indie studios (PontyPants 2020; Creative Blog Staff 2019). Considering this, Unity would have been better choice at the start of my game development journey, since I had zero knowledge about video game industry. However, as information grows, Unreal provides better tools for achieving desired results (Prinke 2019). So, in the long run I consider UE4 to be the better option for my usage.

2.2 Blender and Assets

Blender was my choice for 3D editing software. This was purely because it is free (Blender 2020). Assets, such as 3D models and animation, were gotten either from Adobe's Mixamo or Unreal Engine's Market place. Some of these were modified with Blender. Modifications and distributing these assets as a part of software is allowed according to Adobe's Mixamo license and Unreal Engine Marketplace's license. Only selling or distributing modified assets as themselves are prohibited, so distributing games with these assets is allowed. (Adobe 2018, Unreal Engine Marketplace)

2.3 Development methodologies

This project didn't knowingly use any development methods that are defined. This is due to the fact that this project is a one man project. There are also no employers Involved. Everything was on my responsibility. We can, however, define some methods which are defined that happen to coincide with how I managed this project.

The closest development methodology used in this project would be incrementally built prototype, which evolves as the project matures. There are two kinds of prototypes: throwaway and evolutionary. Throwaway prototype, like the name implies, is thrown away at the end of the development, because usually throwaways are built fast and only to fulfill the required specifications, quality is in many cases second-rate. Evolutionary prototypes are built quality in mind. These prototypes are modified as new information and requirements present themselves. (Davis 1995,17-18)

As mentioned above this project uses the evolutionary prototype. C++ is a kind of stiff language to build prototypes on since it is a statically typed language. Creating throwaway prototypes with C++ might be too expensive, so evolutionary option seems a bit more appealing. Other reason may be my own discontent creating fickle systems that in the end get destroyed. In other words, quality is important to me.

Regarding to quality, software gurus have defined guidelines which could be used to create cleaner and maintainable code. We can use some retrofitting here as well. There are five:

Single Responsibility Principle (SRP), Open-Closed Principle (OCP), Liskov Substitution Principle (LSP), Dependency Inversion Principle (DIP) and Interface Segregation Principle (ISP) (Martin 2014, 86). Apparently, I used SRP and DIP in this project.

SRP is used to make it easier to change class implementation and to reason about code. The rules states that: “A class should have only one reason to change” (Martin 2014, 95). We can interpret this as follows: “A class should only have one responsibility”. This is not easily achieved in systems like UE4 or game programming in general. However, if there is an opportunity to create a class with only a single responsibility, it should be taken.

DIP method is for reducing dependency issues between objects. While programming we might create dependency between two classes. To call one class’ functions from another the caller needs to know about the called class. This is a problem which becomes more prevalent when there are multiple different classes that needs to be called from the caller class. We could introduce a base class for every called class. Through polymorphism we could call the wanted function, however, what happens when the called class can’t inherit from base class? This solution is incomplete. What DIP suggests is that we could create an Interface, from which called class inherits and caller class knows of. Now the caller class needs to only know about the interface and nothing about the called class. Called class on other hand needs to only implement the interface. In other words, the only dependency is the interface. (Martin 2014, 127-134)

2.4 Relevant consepts

Collision Volume

A volume, e.g. sphere, cube, capsule and so on, that can collide within the game world (Volume Reference).

Primitive Component

A component that contains or generates geometry in game world. Can be rendered or used as collision data. (UPrimitiveComponent)

Blueprint

Blueprint name is used mainly to refer either UE4’s visual scripting system or an object that uses UE4’s visual scripting system (Blueprints Visual Scripting).

Physics Engine

System that handles physical simulation calculations and collision calculations in a video game (Physics Simulation).

Shape trace

Is a type of trace. Trace is defined by start point and end point. A shape trace is where we move a shape between these points and collect information about objects that collide with this moving shape. (Tracing)

Animation Montage

An asset that contains multiple animation assets. In this asset it is possible to define further functionalities for animations like attaching events. (Animation Montage Overview)

Event

Event is a function that fires when a certain thing happens. For example, if a variable's value changes, we notify value change to some object.

Skeletal Mesh

Visual representation of an animated object (Skeletal Meshes).

Fps

Frames per second. How many times we can update our screen in a second.

DeltaTime

1/fps. How much time was spent from last rendered frame to current frame.

Animation Pose Snapshot

Takes a Snapshot, which stores all current bone transformations from a Skeletal Mesh (Animation Pose Snapshot).

AI Perception Component

Component which defines what senses AI uses to query information about the surrounding environment. For example, one can configure AI to use sight, hearing, touch or all of these. (AI Perception Component)

3 UE4 Architecture

A video game character consists of many multiple components. For simplicity's sake, we are going to differ character types into two main categories: Player and AI. AI and player only differ a bit, in player input is given from hardware, whereas AI's "input" is driven by decisions AI's "brain" makes.

3.1 Actors

Everything that can be placed into the world is an actor. There may be many kinds of actors. An actor may be and a character or inanimate object of some sort. Actor is the base class for most thing that exist in the game world. (Actors)

3.2 Pawn and Character

Pawn is a base class for all controllable, either player or AI, things. Character is a child class of a pawn and is usually used for humanoid like creatures. Pawn on the other hand, can be a vehicle or a creature of some sort. Character class only provides more specialized kind of behavior. (Pawn; Character)

3.3 Events

Events are functions to be called when something happens. An example would be some value that changes. Rather than constantly checking if a value has changed, we can bind event to the value. In code this could mean when the value changes a function gets called, which notifies the change to everyone that has bound to this event. In UE4 this is done by delegates. Delegate holds all the functions we want to call after value gets changed.

3.4 Input

To have an agent move in the game world, we must collect player's input. In Figure 1, we first give the input to PlayerInput Mapping. We could define this as a place where hardware input is changed to meaningful input from engine's perspective. For this input we may bind events from multiple classes, that own InputComponent. As an example, if we would want to fire of a jump command, we would first need a button and an event name associated with it. Then we would bind an event function to it using Input component. In Figure 2 we set our key mappings. For example, "MoveForward" is the action name and under it is keys that binds to it. Figure 3 shows how we bind events to these actions. Events are bound by giving an action name and function we wish to call. To summarize, when we press button the function bound to that button executes. (Input)

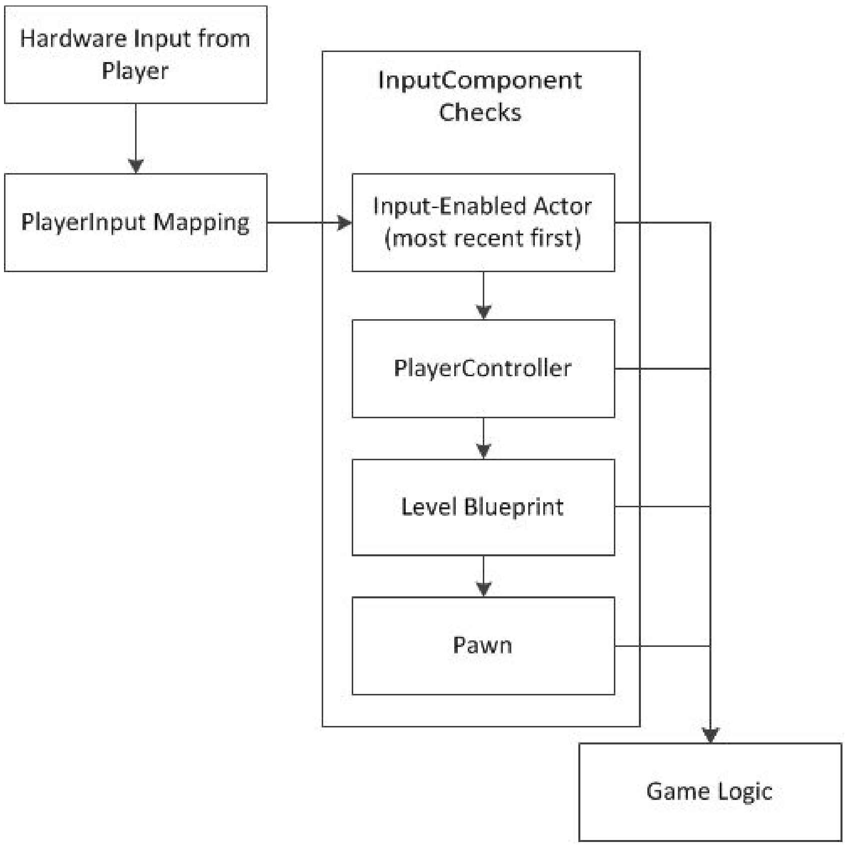
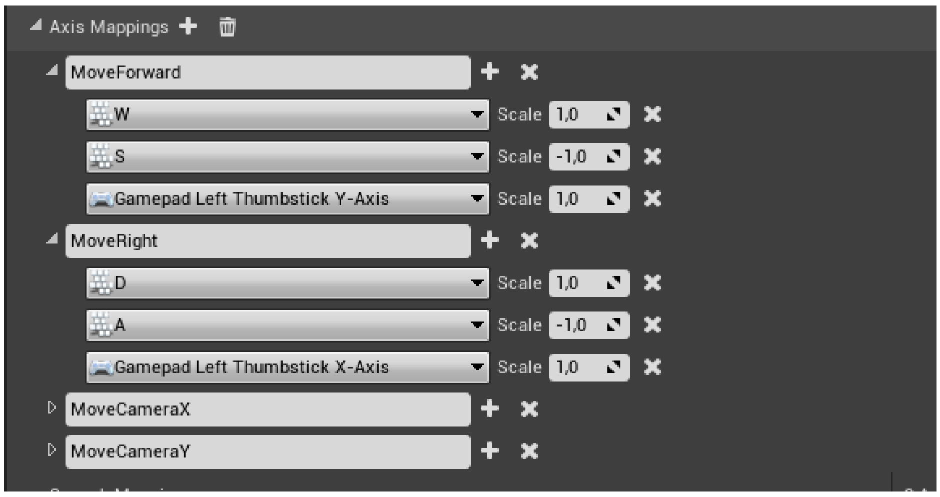


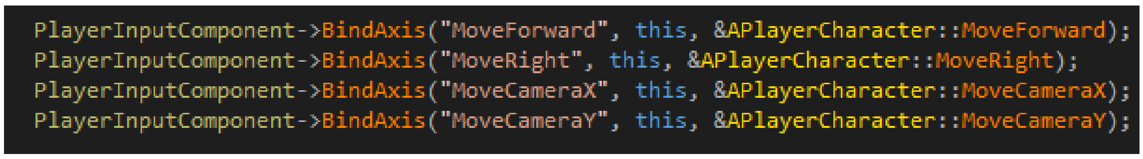
Figure 1: Input graph.



Axis Mappings

MoveForward		
W	Scale	1,0
S	Scale	-1,0
Gamepad Left Thumbstick Y-Axis	Scale	1,0
MoveRight		
D	Scale	1,0
A	Scale	-1,0
Gamepad Left Thumbstick X-Axis	Scale	1,0
MoveCameraX		
MoveCameraY		

Figure 2: Typical input mappings



```
PlayerInputComponent->BindAxis("MoveForward", this, &APlayerCharacter::MoveForward);
PlayerInputComponent->BindAxis("MoveRight", this, &APlayerCharacter::MoveRight);
PlayerInputComponent->BindAxis("MoveCameraX", this, &APlayerCharacter::MoveCameraX);
PlayerInputComponent->BindAxis("MoveCameraY", this, &APlayerCharacter::MoveCameraY);
```

Figure 3: Input mappings in C++ (Exists in character class).

3.5 Controllers

Controllers are actors that can possess a pawn, or any other pawn derived class (Controller). The possession event can be thought as a soul or a ghost that possesses the target pawn and starts to move it.

Controllers are different for player and AI. PlayerController holds multiple InputComponents as a stack of weak pointers, which might have bound events with them. These events are responsible for player's actions. AI, on the other hand, has its ”input” coming from code. AI's ”input” may be driven from a behavior tree or some other system. In UE4 case this is most likely to be a behavior tree.

Controllers have other functionalities as well, but we won't be delving into them. Usually it is recommended to store important and persistent variables in PlayerController. We might want to destroy player's character or possess a new character at certain point. This would mean that all stuff that player holds might get lost or would require some sort of item dump. PlayerController rarely gets destroyed, so all the important information should lie there.

3.6 Movement component

As the name implies this component is responsible for movement. In games, we move a PrimitiveComponent with simple collision around the game world. Collision volume would be something like capsule, sphere, cube and so on. The point here is that it is a simple shape. Movement component is in charge of the physics forces that are applied to the player's movement. Forces are based on physic states and player input. For instance, player that is swimming would probably have slower movement speed that of moving on ground and, also, some buoyancy forces would be applied as well. It is Important to note that vehicles and projectiles have also their own associated movement components. (Movement Components)

3.7 Animation blueprint and AnimInstance

AnimInstance is a parent class of Animation Blueprint. Animation Blueprint animates targeted Skeletal Mesh (Animation Blueprints). Many times, Animation Blueprint works together with a movement component, we want to know the physics state of the player, since the animation is decided by it. Animation Blueprint is also responsible for animations, which are triggered by events. An example would be player's attack animation, which happens when a player presses a button.

3.8 AI, Behavior tree and Blackboard

There are many ways to create an AI system. UE4 however, uses behavior trees in AI decision making. Behavior tree uses blackboard as a data store which holds all relevant values that are required for tree’s execution (Behavior Tree Overview). Blackboard values might for example define some threshold values or simple booleans, which affect how the behavior tree runs. Blackboard variables are populated by AI Perception Component.

Behavior tree is node base system, which executes from root node to leaf nodes, based on rules and conditions. Nodes can have certain rules assigned to them, which decide if a node is to be executed. In behavior tree nodes are executed from left to right. UE4 uses event driven behavior tree - the tree remembers what node is executing or should be executing. This is contrary to a conventional tree which always starts executing from root node until we find a node which the tree can run. Event based tree remembers the node, so, we can just start execution from the node. Therefore, using event driven tree can save performance compared to conventional behavior trees. (Behavior Tree Overview)

3.9 Node types

As discussed, behavior tree consists of nodes. In UE4 there are four main node types: Composite, Task, Decorator, Service. (Behavior Tree Overview)

Composite nodes are the tree’s main structure, that create the branching behavior. There are three sub types of composites: Selector, Sequence and Simple parallel. Selectors run their child nodes until one succeeds or until the last child has executed. Sequence will run all of its child nodes and stop executing if one of them fails. Simple Parallel allows a task node to be run alongside with tree. Selector and Sequence are the most used composites. (Behavior Tree Node Reference: Composites)

Task nodes are the ones which tells an AI agent to do the things it specifies. After task is completed, aborted or failed, node sends a status flag to the node’s parent node for further processing. In many cases parent is a Selector or Sequence node. (Behavior Tree Node Reference: Tasks)

Decorators are also called conditionals. These nodes can be placed before Composite nodes or Task nodes. Decorators may decide whether to take certain branch or execute a certain task. A branch may also be aborted based on decorator. This is done by event, for instance, if the monitored value changes we can abort executing this branch (and all branches left of this node) and move to next node”. Moreover, Decorators have also the power to modify returning flag from Task node and cause looping behavior. (Behavior Tree Node Reference: Decorators)

Service nodes are run at certain intervals. Service node starts executing when the branch where service node lies becomes active. In turn, inactivation happens when branch becomes inactive. Service nodes are useful for variables that require constant checking. If we would want our AI to behave differently based on player's distance, we could calculate the distance on specific intervals. (Behavior Tree Node Reference: Services)

3.10 Animation

In UE4 animation is handled by Animation Blueprint (Unreal Engine 4 Documentation q). There are couple of main asset types for animation: AnimSequence, Animation Montage, BlendSpace and PoseAsset. AnimSequence is a container for a basic type of animation. Animation Montage can own multiple AnimSequences and defines a slot in which the animation should be played.

In Animation Blueprint it is possible to build state machines which handles Skeletal Mesh's animation from state to state. Usually these states are queried from movement component. After the state machine has decided animation, we may modify it or override it with other animations. It is common to have Animation Montages, which can override either completely or just modify some part of base animation. For instance, our character is moving on ground and wants to attack. Rather than having a state specific for attack, we can modify the animation at the later stages by checking if Animation Montage Slot has an animation that wants to be played. If such animation exists, we may modify the base animation.

In Figure 4 we have an example of a simple state machine in Animation Blueprint. State machine starts from "Entry" and moves to "BS_IdleWalkRun". "BS_IdleWalkRun" is a state that state machine is currently executing. "BS_IdleWalkRun" defines some animation that the Animation Blueprint should be playing. State machine changes state if one of the transition rules becomes true (Figure 5). In one frame the state may change multiple times. There are, however, a set amount of state changes we can take in one frame. Limiting state changes prevent any infinite loops from happening.

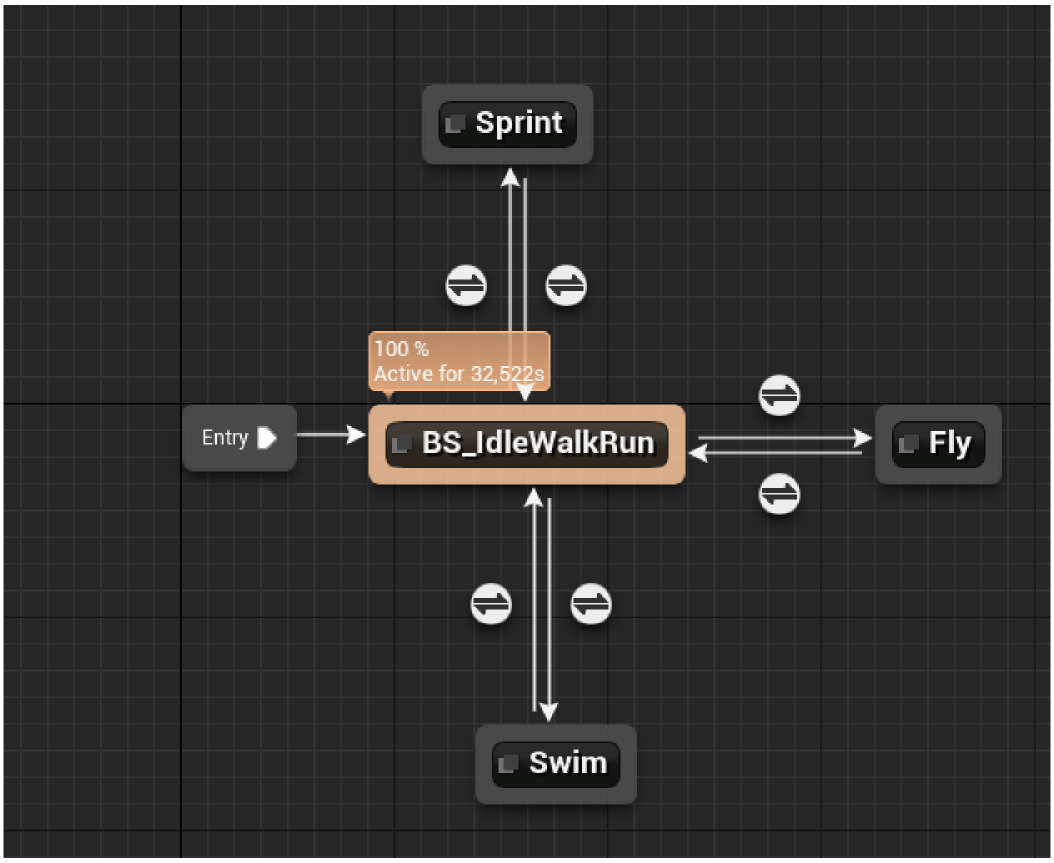


Figure 4: Example state machine

Figure 5: Transition condition

4 Project run through

In this chapter I'll go through the parts this project holds. Some parts are omitted or abridged.

4.1 Creating attack

Every attack needs an animation and a button to perform the animation. Animation assets for this project were gotten from Mixamo or Unreal Engine's marketplace. Animation assets can be contained in Animation Montage asset. Here animations can be separated by sections. It is also possible to set events in Animation Montage asset. With these events we can fire of functions when animation plays and finally arrives at a specific time point.

Using events, one can make things appear or disappear, activate or deactivate things and so on. As of creating an attack, my first thought was to have a collision volume overlapping a weapon and when event fires collision volume becomes activated. This collision volume could also be overlapping player's fists if hand to hand combat was required. Activation here means that the collision volume can damage enemies or any damageable unit. Collision volumes are a valid way to do things, however, I disliked the idea of having the collision volumes to exist in the world, when they were not obviously needed. Collision volumes would be only needed when an attack happens.

My second thought was then to make a collision volume appear when a certain event plays and then disappear when another event fires. This seemed like a valid solution, but what happens if, for example, the disappear event never fires. This can happen if the animation stops before the disappear event can fire. Case is the same for if we would only activate and deactivate the collision volume with events.

The collision volume not disappearing would mean that every frame we simulate the game, we would be able to hit enemy. Enemies would die in an instant. This bug seems a lot like the Infinite Sword Glitch from Zelda Ocarina of Time (Puissance-Zelda 2013). There is also another problem as it turns out, when a collision volume hits an enemy, we only get information about that it has indeed hit an enemy. What we don't get, is information about where we hit the enemy. There is no location information. This is due to how physics engine handles collision.

The physics engine kind of sees the game world in snapshots, it just checks are there any object penetrating (two objects are partly or completely inside of one other). If penetration occurs, one of the overlapping objects gets pushed out of the other object. So, physics engine only knows that one of the objects needs to be pushed some amount in some direction to solve the penetration. Turns out I needed the location information later, so third idea was required.

Third and the last idea, I settled into, was to simply create a shape trace from some location to target location. In case of a sword for example, we would use a shape trace from sword's last known location to the sword's current location. This was the best solution which solves the problem with Infinite Sword Glitch and location problem. Shape traces can be thought like moving a shape on a linear path by some amount. If the shape trace hits, we get information about the hit event, location included of course. It is possible to fire an event in every frame of animation, this done by using AnimNotifyState class. This would mean if animation gets canceled prematurely, there is no more event to fire of the shape trace. Infinite Sword Glitch prevented.

Usually video game characters have multiple different attacks. I had planned to have at least a punch, a kick and a sword slash. These attacks are different in a sense that they use different body parts and sword slash requires a sword object of course. Punch and kick would start their shape trace at a socket (can be thought as a point in character’s Skeletal Mesh) located in fist or in feet, whereas sword could have its own socket.

Weapon shape trace is tricky though. There is a problem with thinner and longer shapes (Figure 6). There is an arc going with sword swing. We cannot rotate the shape trace. We can only move it linearly. The question is which rotation we should use swords last rotation or current rotation? Using either one of these rotations lead to decrease in accuracy of hit event. This is bad, since for player it may look like sword should have hit, when it didn’t. Sometimes it may be the other way around, a clear miss results into a hit. If we use this same method to generate enemy’s attacks, players will get frustrated, when they take damage from a clearly missed attack. Solution to this is to have multiple shape traces or line traces along the sword.

As seen in Figure 6 the sword swing seems to generate shape traces that are somewhat accurate. The shape traces are representing pretty much the sword path of motion. However, shape trace density depends on fps. When fps is low (Figure7), the accuracy problem becomes too prevalent. In Figure 7, in the right image, the blue area gets ignored. Imagine that we have an animation that lasts one second. In 120fps we get 120 snapshots that means 120 shape traces. In 12fps we get only 12 shape traces. If the sword moves fast, then in lower fps, the sword appears to teleport.

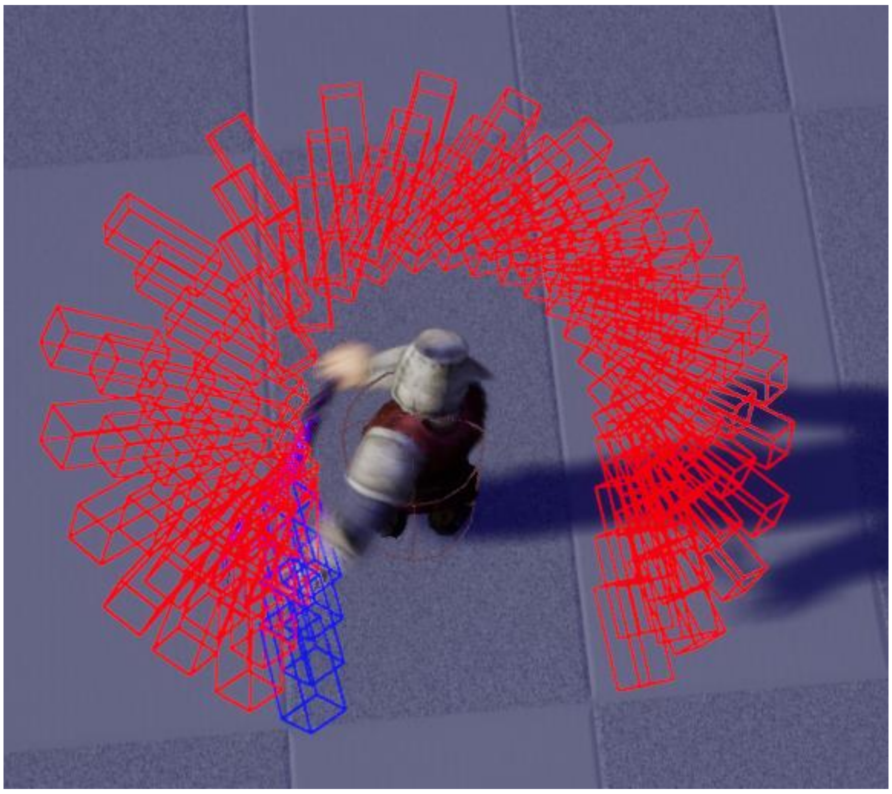


Figure 6: With 120fps we get full arc and a precise enough hit detection.

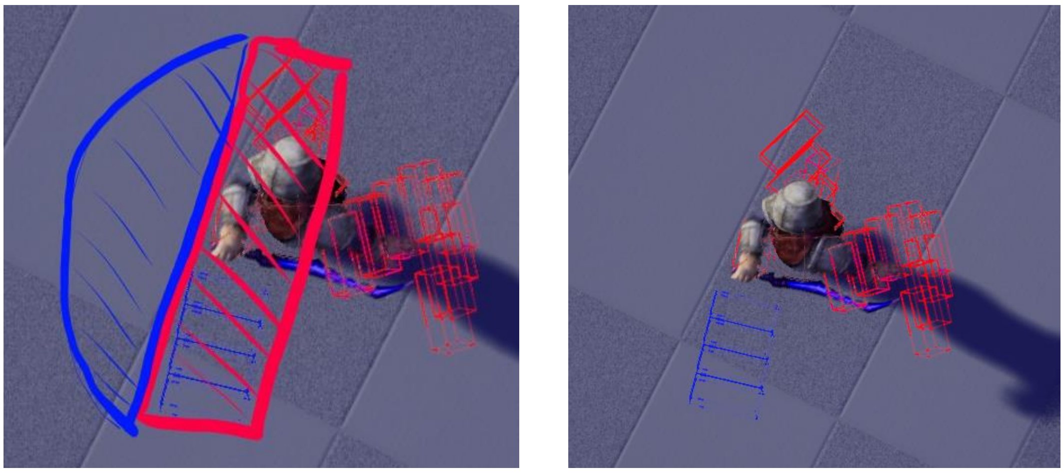


Figure 7: With 12fps we lose accuracy too much.

4.2 Animation

UE4 uses AnimInstance and Animation Blueprint (ABP) to drive animation. Animation Blueprint provides a graphical way to organize animation flow. Usually ABP contains a state machine or multiple state machines that drive animation based on boolean values. After state machine phase we can modify animations further and override them either partially or completely. Animation Montages usually do this by having an associated slot in which Animation Montage plays. Piping the incoming animation from state machine through this slot may modify or override the whole animation.

Creating an organized state machine in ABP can be a challenging thing. Complexity increases as the amount of states increases. State machine turning into spaghetti seems to be a common problem which is exaggerated in Figure 8. Luckily, I foresaw this problem before I had to deal with it. So, I thought up some relieving remedies.

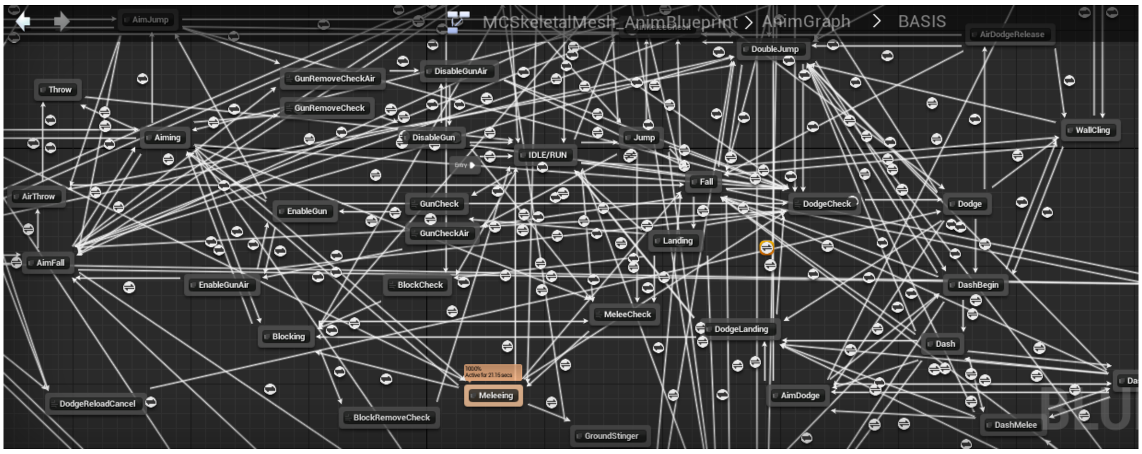


Figure 8: Spaghetti state machine (UE4 AnswerHub 2014).

First insight was to divide movement state into two: super state and sub state. Super states are for example, moving on ground, moving in liquid, flying and such. Whereas sub state defines how we move for example on ground we could be running, walking, idling, sitting and so on. Our character doesn't have to be moving per se, being idle on ground is considered a type of movement. Having super and sub states differentiated, we can now build a state machine for every super state. So, we have one main state machine (Figure 9), which holds one state machine for each defined super state. Super states are InLiquid, InAir and OnGround. From entry execution enters into ”BS_IdleWalkRun” state which transitions to conduit by default, since the transition is always true. From Conduit the execution proceeds into one of the super states based on transition rules. Inside a super state node lies another state machine which ultimately decides what animation ABP should play.

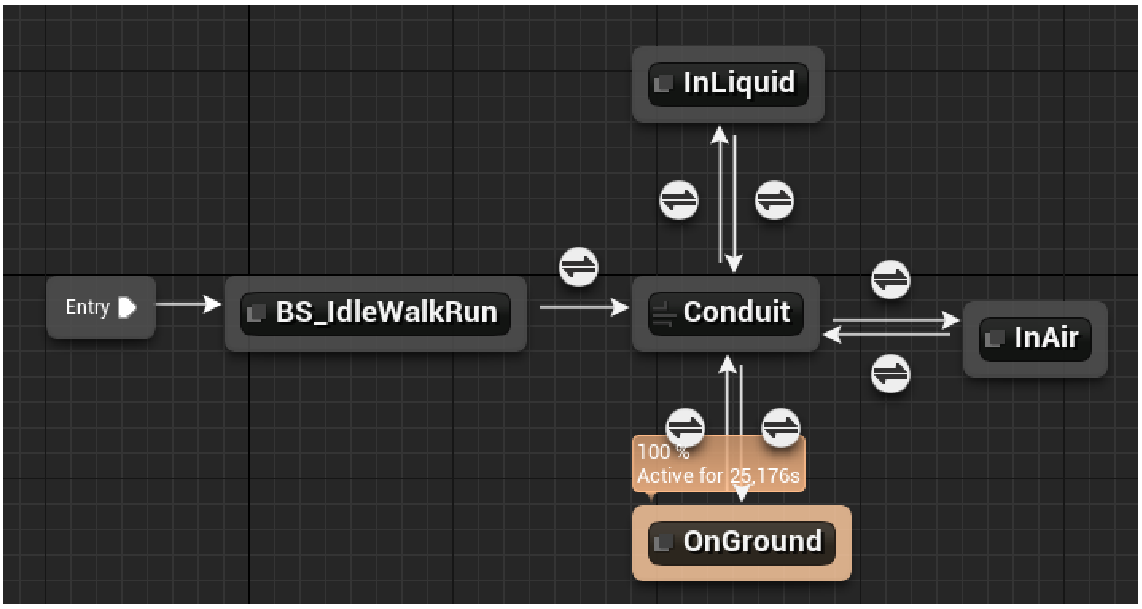
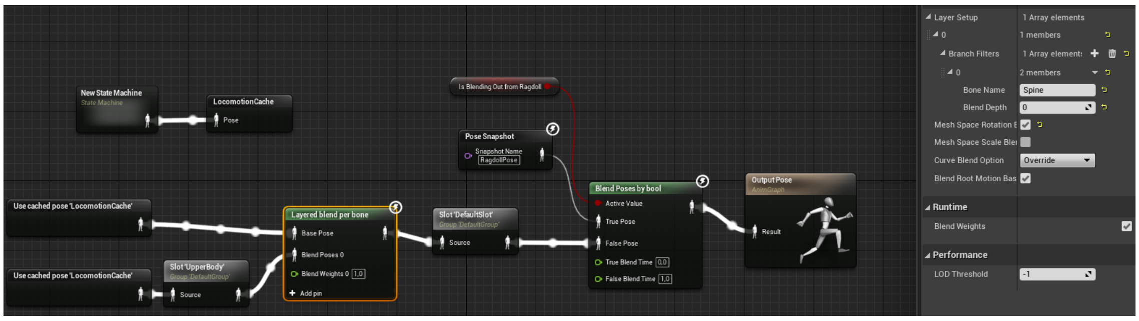


Figure 9: Animation State machine super states

After the state machine phase, we can modify the animation by slots. In Figure 10 node going out from ”LocomotionCache” we use ”Slot UpperBody” to be layered up from Spine bone. This means whatever animation happens in UpperBody slot does not affect the feet. Then our execution moves to ”Slot DefaultSlot”. This slot overrides our whole animation, if an animation is happening in DefaultSlot, if not the animation will not change. After DefaultSlot execution goes to ”Blend Poses by bool”. This has nodes going in. Red one is the boolean and ”Pose Snapsot” is a saved one frame animation. This blending is intended only when we are blending out of ragdoll phase. Ragdoll is a term for Skeletal Mesh which is simulating physics. In ragdoll phase keyframed animations get ignored.



Layer Setup	1 Array elements
0	1 members
Branch Filters	1 Array element:
0	2 members
Bone Name	Spine
Blend Depth	0
Mesh Space Rotation E	✓
Mesh Space Scale Bler	
Curve Blend Option	Override
Blend Root Motion Bas	✓
Runtime	
Blend Weights	✓
Performance	
LOD Threshold	-1

Figure 10: AnimGraph. Slot Blend

4.3 AI

Goal was to create a simple AI with simple behavior. Behavior tree focuses mainly on combat behavior. These enemies are kind of only good for combat and patrol.

In game world, we could define AI as an agent that perceives game world through some perception system and makes decisions based on perception system's observations. An agent that AI system controls could be a character, but also some simple thing, like a homing projectile.

When dealing with humanoid like AI, player expects them to behave more like a human. Essentially, we as game developers, want to sell the player an illusion of rational agent controlling the AI character. The player will notice if AI agent makes an obviously wrong decision. A good example would be when AI agent keeps running into walls or repeating same actions in a loop without success. AI character should behave like a human would. However, some human like behavior in AI can be seen as buggy behavior. For instance, when AI would change its mind about taking some action, it would be seen as faulty AI, even though humans change their minds quite frequently. (Rabin 2014, 4-5)

AI shouldn't make obviously wrong decisions, but what happens when AI makes always the most optimal decisions? If the player is competing against such AI, winning might be hard or even impossible. The game might not be fun for the player and the player probably stops playing the game completely. Too smart AI can be a problem. This can be solved by introducing some artificial stupidity. (Rabin 2014, 5)

In UE4 AI characters use different controller from player since they are AI controlled – AI controller. AI controller contains BehaviorTreeComponent in which we can assign our behavior tree. Behavior tree contains or should contain a blackboard. Blackboard contains information which can be populated by AI Perception Component (Figure 11). Perception can be configured to use engine's own AI Senses. This project only used sight as a sense.

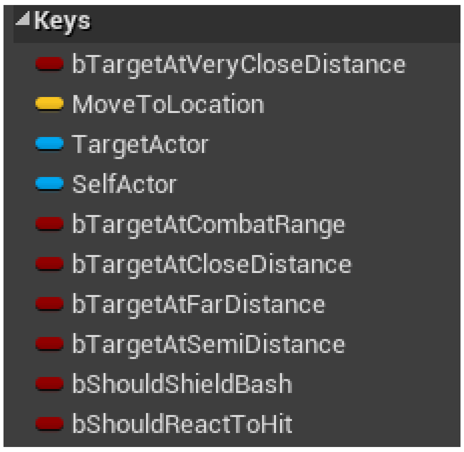
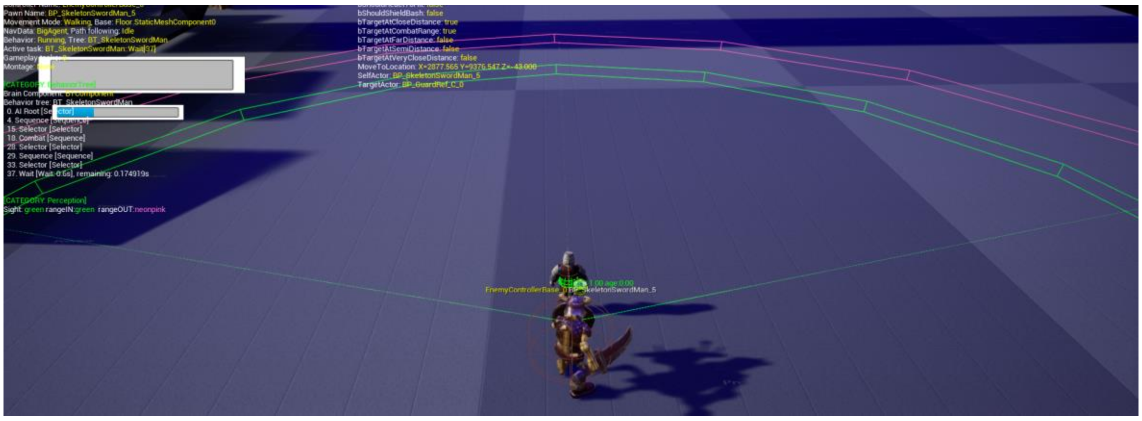

Keys
bTargetAtVeryCloseDistance
MoveToLocation
TargetActor
SelfActor
bTargetAtCombatRange
bTargetAtCloseDistance
bTargetAtFarDistance
bTargetAtSemiDistance
bShouldShieldBash
bShouldReactToHit

Figure 11: Blackboard variables

AI sight is basically a cone extending from AI character (Figure 12). If an object of interest overlaps this cone we fire of an event. In our case if player overlaps this cone an event is fired which modifies blackboard's TargetActor value. TargetActor's value change leads to another event that ends AI's patrol event and starts left most sequence. This is due our decorator node, which aborts any lower priority nodes on TargetActor value change. Node priority is higher the more left the node is.



```
Pawn Name: BP_SkeletonSwordMan_5
Movement Mode: Walking, Base: Floor StaticMeshComponent0
NavData: BigAgent, Path following: Idle
Behavior: Running, Tree: BT_SkeletonSwordMan
Active task: BT_SkeletonSwordMan::Wait[37]
Gameplay
Montage:

[CATEGORY:
Brain Component: BTComponent
Behavior tree: BT_SkeletonSwordMan
 0. AI Root [Selector]
 4. Sequence
 15. Selector [Selector]
 18. Combat [Sequence]
 28. Selector [Selector]
 29. Sequence [Sequence]
 33. Selector [Selector]
 37. Wait [Wait: 0.5s], remaining: 0.174919s

[CATEGORY: Perception]
Sight: green rangeIN:green rangeOUT:neonpink
```

```
bShouldShieldBash: false
bTargetAtCloseDistance: true
bTargetAtCombatRange: true
bTargetAtFarDistance: false
bTargetAtSemiDistance: false
bTargetAtVeryCloseDistance: false
MoveToLocation: X=2877.565 Y=9376.547 Z=-43.000
SelfActor: BP_SkeletonSwordMan_5
TargetActor: BP_GuardRef_C_0
```

Figure 12: AI vision cone. Green indicates in vision range.

In Figure 13 we see Patrol behavior being the most right. This behavior happens when Blackboard Based Conditions are not both true – if either of them is false or both we are currently doing Patrol behavior. When Both conditions become true, event when enemy sees player, we begin combat start phase (Figure 14).

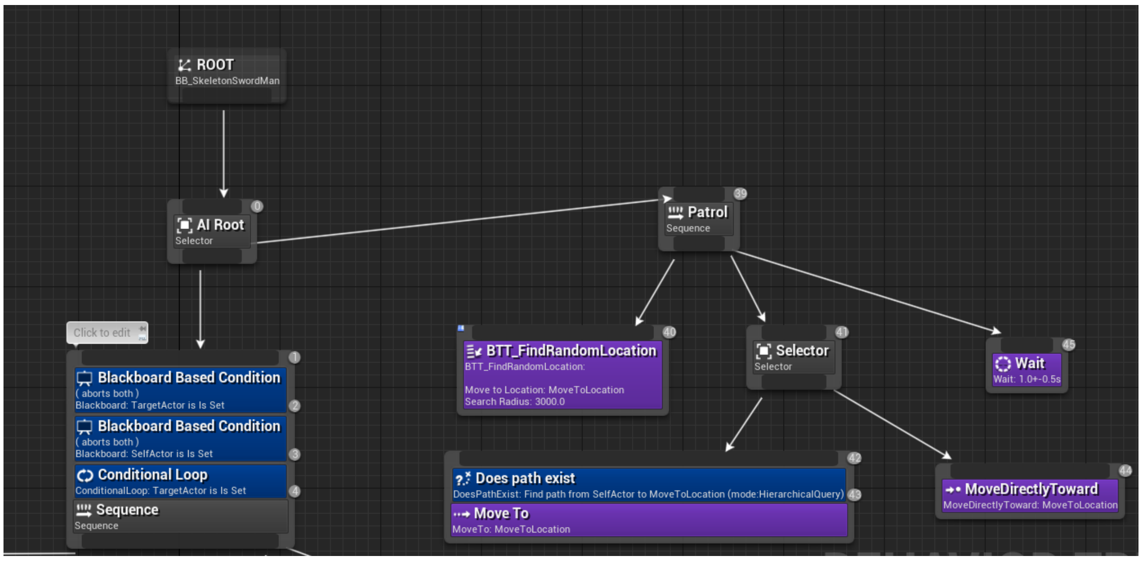

ROOT
BB_SkeletonSwordMan
AI Root
Selector
Click to edit
Blackboard Based Condition
(aborts both)
Blackboard: TargetActor is Is Set
Blackboard Based Condition
(aborts both)
Blackboard: SelfActor is Is Set
Conditional Loop
ConditionalLoop: TargetActor is Is Set
Sequence
Sequence
Patrol
Sequence
BTT_FindRandomLocation
BTT_FindRandomLocation:
Move to Location: MoveToLocation
Search Radius: 3000.0
Selector
Selector
Wait
Wait: 1.0+-0.5s
Does path exist
DoesPathExist: Find path from SelfActor to MoveToLocation (mode:HierarchicalQuery)
Move To
MoveTo: MoveToLocation
MoveDirectlyToward
MoveDirectlyToward: MoveToLocation

Figure 13: Patrol behavior.

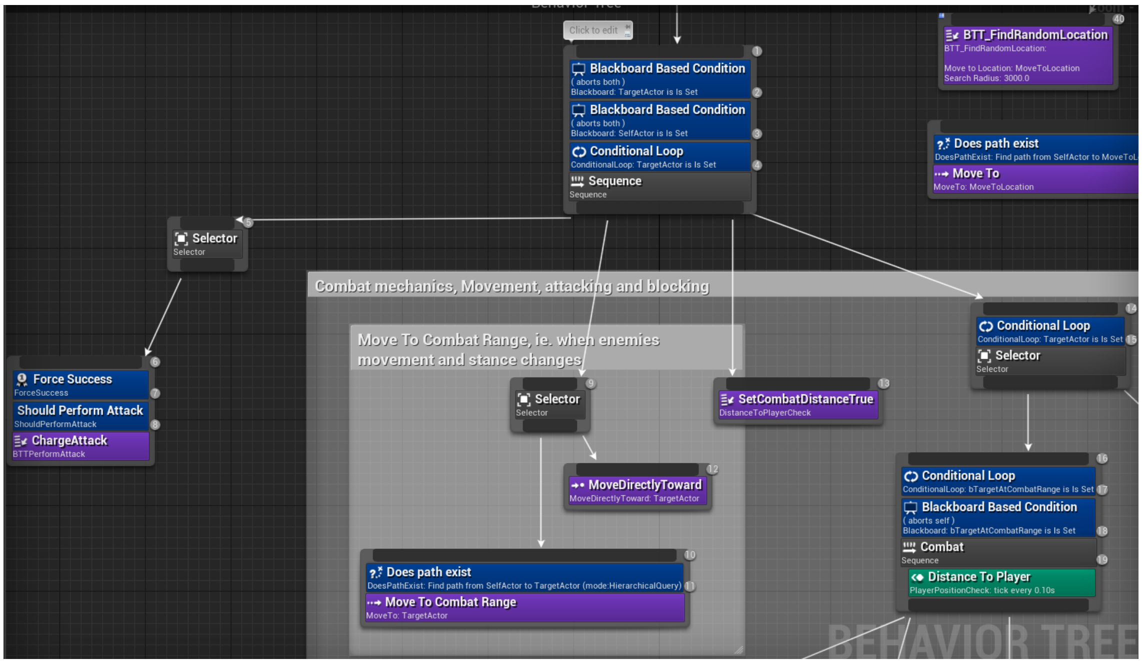

Click to edit
Blackboard Based Condition
(aborts both)
Blackboard: TargetActor is Is Set
Blackboard Based Condition
(aborts both)
Blackboard: SelfActor is Is Set
Conditional Loop
ConditionalLoop: TargetActor is Is Set
Sequence
Sequence
BTT_FindRandomLocation
BTT_FindRandomLocation:
Move to Location: MoveToLocation
Search Radius: 3000.0
Does path exist
DoesPathExist: Find path from SelfActor to MoveToL
Move To
MoveTo: MoveToLocation
Selector
Selector
Force Success
ForceSuccess
Should Perform Attack
ShouldPerformAttack
ChargeAttack
BTTPerformAttack
Combat mechanics, Movement, attacking and blocking
Move To Combat Range, ie. when enemies movement and stance changes
Selector
Selector
SetCombatDistanceTrue
DistanceToPlayerCheck
MoveDirectlyToward
MoveDirectlyToward: TargetActor
Does path exist
DoesPathExist: Find path from SelfActor to TargetActor (mode:HierarchicalQuery)
Move To Combat Range
MoveTo: TargetActor
Conditional Loop
ConditionalLoop: TargetActor is Is Set
Selector
Selector
Conditional Loop
ConditionalLoop: bTargetAtCombatRange is Is Set
Blackboard Based Condition
(aborts self)
Blackboard: bTargetAtCombatRange is Is Set
Combat
Sequence
Distance To Player
PlayerPositionCheck: tick every 0.10s
BEHAVIOR TREE

Figure 14: Combat start phase

Here enemy starts combat by charging towards player by initiating a task ”ChargeAttack”. Since there is a decorator in this node ”Force Success”, even if enemy fails the charge we proceed to do next things. Force Success is here to prevent any infinite loops in which would happen if, for some reason, ChargeAttack fails. After charging enemy moves closer towards player. When close enough we start our close combat phase (Figure 15).

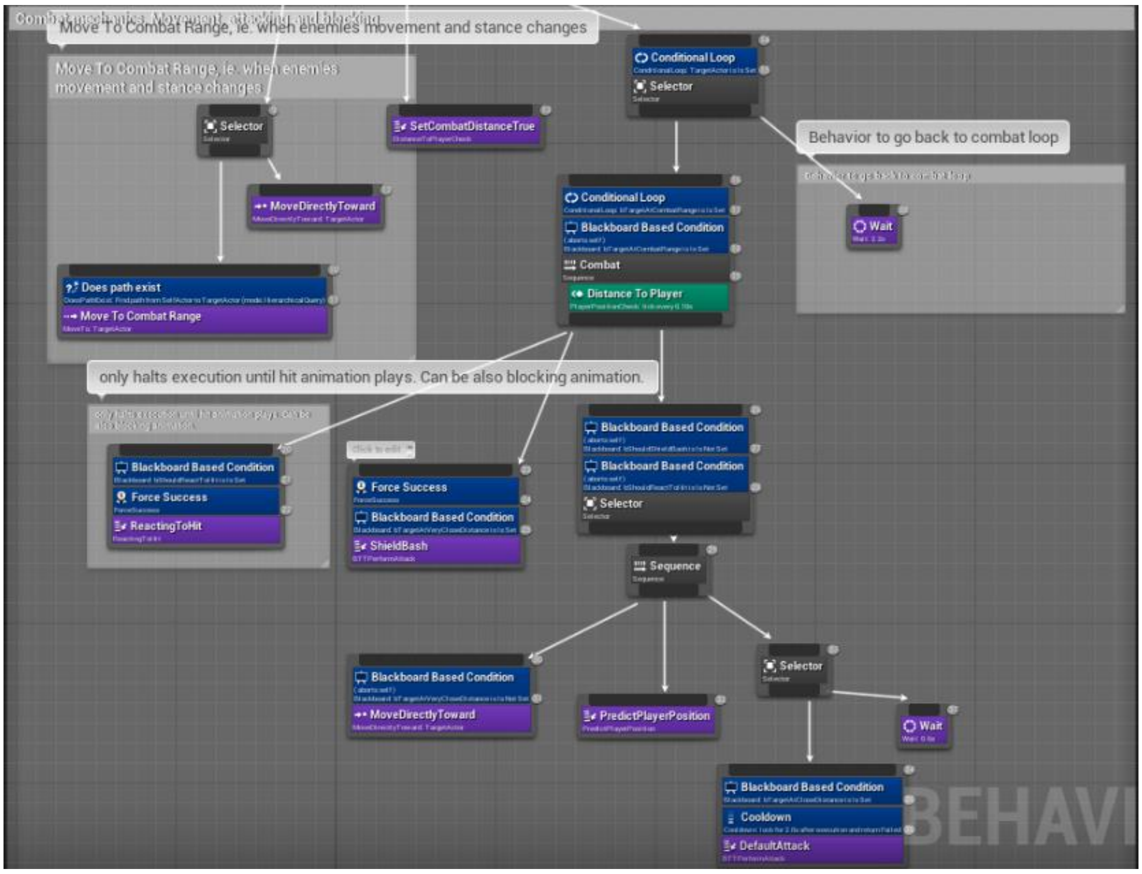

Move To Combat Range, ie. when enemies movement and stance changes
Selector
SetCombatDistanceTrue
MoveDirectlyToward
Does path exist
Move To Combat Range
Conditional Loop
Selector
Behavior to go back to combat loop
Wait
Conditional Loop
Blackboard Based Condition
Combat
Distance To Player
only halts execution until hit animation plays. Can be also blocking animation.
Blackboard Based Condition
Force Success
ReactingToHit
Force Success
Blackboard Based Condition
ShieldBash
Blackboard Based Condition
Blackboard Based Condition
Selector
Sequence
Blackboard Based Condition
MoveDirectlyToward
PredictPlayerPosition
Selector
Wait
Blackboard Based Condition
Cooldown
DefaultAttack
BEHAVI

Figure 15: Close combat phase.

In close combat phase (selector node name Combat), we start a combat loop. The most left ”ReactingToHit” is first priority. This node exists so enemy may abort everything AI agent is doing when it gets hit by the player. This plays also only when condition bShouldReactToHit is set to true. Next up is ”ShieldBash” which is aborts everything from right of this node. This is an attack to prevent player getting too close and fires when bTargetAtVeryCloseDistance is true.

Next, we get to the default behavior for combat when ReactingToHit and ShieldBash are not happening. This node group performs the basic attack first by moving closer to the player ”MovingDirectlyToward”. Then enemy predicts player position based on player’s velocity. If player is predicted to be too close, this node group get aborted and ShieldBash task is performed. Otherwise enemy performs its ”DefaultAttack” task.

There is a selector above DefaultAttack task node. There is also a conditional for cooldown. Cooldown prevents the enemy AI from spamming its attack. Selector node prevents infinite loop here to run wait task if we return failed from DefaultAttack task.

There further notes to consider. As one can see in Figures above, there are conditional loops sprinkled around some nodes. These loops are here to prevent behavior tree to start executing from root when tasks are finished. We save some performance by doing the loops. However, one can see that there are also some Force Success decorators sprinkled around as well. These are to prevent infinite looping, which may occur, if Sequence node has conditional loop and the first child node fails. This case Sequence node would just try to execute the failed child again and again. Force success prevents this completely.

5 Speculation and critique

This chapter focuses on the faults of this project. Some speculation about trade-offs between certain choices are discussed as well.

5.1 AI critique

Behavior tree structure could be better. Behavior tree's bad structure exists most likely because of my own skills and less likely due to limitations of behavior tree. The execution of the tree is a bit convoluted since there are multiple conditions of aborting nodes. The execution seems to jump all over the place and I get the feeling of spaghetti code. It would seem to be too hard for me to define complex behavior with behavior trees. Maybe complex behavior would be easier with a utility-based AI. Not to say that this project contains complex behavior.

Other critique is that the tree structure is not yet ready. There is a need of a new node that returns AI agent back to combat distance when TargetActor gets too far apart from our AI agent. For now, there is only "Wait" node set for this instance. This means our AI character will just stop in place if the distance to player gets too big, and after a while, when it hasn't seen player, it starts patrolling again. Since the tree structure is inherently wrong, I see no reason create additional nodes and behavior, because they may be obsolete in future as the tree's structure changes.

5.2 Attack critique

Attack shape traces are based on last location to current location. If this location change is big and the object moves in a curve-like-motion, we also lose accuracy, since shape traces cannot be rotated. We can do either shape trace with linear motion or rotation (last rotation to current rotation) but not both at the same time. We can help accuracy a bit by using sphere traces along the weapon. Sphere traces are better than line traces, since there can be big spaces between each line trace. We might miss thin objects. Obviously, this can be negated by a design choice of not having thin objects laying around.

Sphere traces do not solve the curve problem. The curve problem becomes more prevalent when fps is low. The curve gets cut off. We could use sub stepping in which movement and rotation gets divided to parts. By moving, let's say, half of the full shape trace length, we get more accuracy. Problem is still not solved. Sword location is wrong. This could be solved if we can get locations out of animation by specifying a bone or a socket and time. This is something I leave for the future to be solved.

Even though there are some accuracy problems. These only arises when fps is low. Some testing with 30fps is required. To relieve accuracy problems, we can give player a bit bigger shape traces that extends further than the visible part of the sword. For enemies we really don't care if their attack misses when it shouldn't. If player's attack misses when it shouldn't, it frustrates the player, but when enemy misses when it shouldn't, the player just thinks that they were lucky.

5.3 Animation critique

I consider animation part to be successful. This is due to the fact that this project's animations are not that complex. The most complicated part is physical animation, in which we blend keyframed animation with physical forces. The hardest part was blend from full ragdoll back to keyframed animation. In project this blending is done in a one frame. This can create some stuttering in the animation. This happens because before blending to keyframed animation from simulating physics we take a Pose Snapshot and we simulate physics for one frame. Then the Pose Snapshot gets applied, the animation goes back to position in which it was before the one frame of physical simulation. This problem may be prevented by two ways. If it is possible to update animation right after we have taken Pose Snapshot, then stuttering wouldn't be an issue anymore. Other possibility is to have blend from physics to animation span through multiple frames.

5.4 Personal growth

During this project, my understanding of C++ and UE4 architecture rose considerably. However, there is a lot to be learned still. At the start of this project, I had already familiarized myself with UE4 and C++, but there were parts which I didn't get before I started working on this project. The biggest one was polymorphism.

Polymorphism seemed like magic. How can one treat one type of an object like it is another type of an object. Why calling a virtual function on parent class, calls child class' function if virtual function is overridden? There were many questions. These questions lead to uncertainty about how the code should be organized. For example, my first take on creating a HUD system and inventory (omitted from this paper), I remember just thinking how it can even be possible to create with statically typed language. I thought, do I really have to create array for every type of item there exists? Turns out, just create base class for every item and store the pointers as the base class in an array. With polymorphism we can call the overridden functions in child classes.

Understanding polymorphism made also understand UE4 architecture better. Reading code got a bit easier. Still, it's C++ language. C++ just seems to be an endless swap and to be master at, it probably takes multiple years or even decades. Understanding compilers and computer architecture as a whole seems to give insight why C++ works the way it does. Maybe in future, to understand C++ better, there is a need to write a rudimentary compiler. Getting familiar with Assembly code could prove to be useful as well.

Being already familiar with UE4 graphical user interfaces, not much improvement was done regarding these things. Nevertheless, recapitulation of using these GUIs creates better memory footprint, therefore, I believe my efficiency in creating things with UE4 GUIs increased.

5.5 Future development

My future development should include some special effects. UE4 provides marvelous systems for creating visual effects. My knowledge of these systems is somewhat limited. I believe it is important in gaming industry to have a general idea of every functioning system that makes up a video game. Games just seems to need multiple interconnections between different systems to function properly. Understanding the underlying systems should help me as a developer to structure my code a bit better.

6 Conclusion

All in all, this project is not ready. Many things need further development as new strategies present themselves. The code part requires refactoring, as my understanding of C++ and UE4 increases. The project's main objective was to teach to me things about game development, UE4 architecture and C++ programming. This objective was reached. Now the next step is to refine and master these skills I've found.

References

Printed

Davis, A. 1995. 201 Principles of Software Development. New York: McGraw-Hill.

Martin, R. 2014. Agile Software Development Principles, Patterns and Practices. Essex: Person Education Limited.

Rabin, S. 2014. Game AI Pro. Collected Wisdom Of Game AI Professionals. New York: CRC Press.

Electronic

Actors. Unreal Engine 4 Documentation. Accessed on 26.09.2020.
https://docs.unrealengine.com/en-US/Programming/UnrealArchitecture/Actors/index.html

Adobe. 2018. Fuse Additional Terms of Use. Accessed on 22.09.2020.
https://wwwimages2.adobe.com/content/dam/acom/en/legal/servicetou/Fuse_Additional_Terms_en_US_20180605.pdf

AI Perception Component. Unreal Engine 4 Documentation. Accessed on 10.10.2020.
https://docs.unrealengine.com/en-US/Engine/ArtificialIntelligence/AIPerception/index.html

Animation Blueprints. Unreal Engine 4 Documentation. Accessed on 26.09.2020.
https://docs.unrealengine.com/en-US/Engine/Animation/AnimBlueprints/index.html

Animation Montage Overview. Unreal Engine 4 Documentation. Accessed on 19.09.2020.
https://docs.unrealengine.com/en-US/Engine/Animation/AnimMontage/Overview/index.html

Animation Pose Snapshot. Unreal Engine 4 Documentation. Accessed on 19.09.2020.
https://docs.unrealengine.com/en-US/Engine/Animation/PoseSnapshot/index.html

Behavior Tree Node Reference: Composites. Unreal Engine 4 Documentation. Accessed on 22.09.2020. https://docs.unrealengine.com/en-US/Engine/ArtificialIntelligence/BehaviorTrees/BehaviorTreeNodeReference/BehaviorTreeNodeReferenceComposites/index.html

Behavior Tree Node Reference: Decorators. Unreal Engine 4 Documentation. Accessed on 26.09.2020. https://docs.unrealengine.com/en-US/Engine/ArtificialIntelligence/BehaviorTrees/BehaviorTreeNodeReference/BehaviorTreeNodeReferenceDecorators/index.html

Behavior Tree Node Reference: Services. Unreal Engine 4 Documentation. Accessed on 26.09.2020. https://docs.unrealengine.com/en-

US/Engine/ArtificialIntelligence/BehaviorTrees/BehaviorTreeNodeReference/BehaviorTreeNodeReferenceServices/index.html

Behavior Tree Node Reference: Tasks. Unreal Engine 4 Documentation. Accessed on 26.09.2020. https://docs.unrealengine.com/en-US/Engine/ArtificialIntelligence/BehaviorTrees/BehaviorTreeNodeReference/BehaviorTreeNodeReferenceTasks/index.html

Behavior Tree Overview. Unreal Engine 4 Documentation. Accessed on 26.09.2020. https://docs.unrealengine.com/en-US/Engine/ArtificialIntelligence/BehaviorTrees/BehaviorTreesOverview/index.html

Blender. 2020. License. Accessed on 26.09.2020. https://www.blender.org/about/license/

Blueprints Visual Scripting. Unreal Engine 4 Documentation. Accessed on 26.09.2020. https://docs.unrealengine.com/en-US/Engine/Blueprints/index.html

Character. Unreal Engine 4 Documentation. Accessed on 26.09.2020. https://docs.unrealengine.com/en-US/Gameplay/Framework/Pawn/Character/index.html

Cleaning Up State Machine Spaghetti. UE4 AnswerHub. 2014. Accessed on 22.09.2020. https://answers.unrealengine.com/questions/142592/cleaning-up-state-machine-spaghetti.html

Controller. Unreal Engine 4 Documentation. Accessed on 22.09.2020. https://docs.unrealengine.com/en-US/Gameplay/Framework/Controller/index.html

Creative Blog Staff. 2019. Unity vs Unreal Engine: which game engine is for you? Accessed on 02.10.2020. https://www.creativebloq.com/advice/unity-vs-unreal-engine-which-game-engine-is-for-you#:~:text=One%20of%20the%20main%20differentiators,the%20same%20level%20as%20Unreal

Input. Unreal Engine 4 Documentation. Accessed on 22.09.2020. https://docs.unrealengine.com/en-US/Gameplay/Input/index.html

Matthew Palaje. 2017. Why I switched over to Unreal Engine 4 From Unity 5. Accessed on 02.10.2020. https://www.youtube.com/watch?v=UzCxWJxsskA

Movement Components. Unreal Engine 4 Documentation. Accessed on 26.09.2020. https://docs.unrealengine.com/en-US/Engine/Components/Movement/index.html

Overview of State Machines. Unreal Engine 4 Documentation. Accessed on 19.09.2020. https://docs.unrealengine.com/en-US/Engine/Animation/StateMachines/Overview/index.html

Pawn. Unreal Engine 4 Documentation. Accessed on 26.09.2020. https://docs.unrealengine.com/en-US/Gameplay/Framework/Pawn/index.html

Physics Simulation. Unreal Engine 4 Documentation. Accessed on 26.09.2020. https://docs.unrealengine.com/en-US/Engine/Physics/index.html

PontyPants. 2020. Unity VS Unreal – Which Engine Should You Choose? Accessed on 02.10.2020. https://sundaysundae.co/unity-vs-unreal/

Puissance-Zelda. 2013. Ocarina of Time – Glitch – Infinite Sword Glitch. Accessed on 11.10.2020. https://www.youtube.com/watch?v=GI1QLSnalnY

Prinke, M. 2019. Is Unreal Engine easy to learn? Accessed on 02.10.2020. https://www.quora.com/Is-Unreal-Engine-easy-to-learn

Skeletal Meshes. Unreal Engine 4 Documentation. Accessed on 19.09.2020. https://docs.unrealengine.com/en-US/Engine/Content/Types/SkeletalMeshes/index.html

State Machines. Unreal Engine 4 Documentation. Accessed on 19.09.2020. https://docs.unrealengine.com/en-US/Engine/Animation/StateMachines/index.html

Tracing. Unreal Engine 4 Documentation. Accessed on 26.09.2020. https://docs.unrealengine.com/en-US/Engine/Physics/Tracing/Overview/index.html

Unreal Engine Marketplace. Marketplace Distribution Agreement. Accessed on 22.09.2020. https://www.unrealengine.com/en-US/marketplace-distribution-agreement

Udemy 2020a. Search result for Unreal Engine. Accessed on 02.10.2020. https://www.udemy.com/courses/search/?src=ukw&q=Unreal+engine

Udemy 2020b. Search result for Unity. Accessed on 02.10.2020. https://www.udemy.com/courses/search/?src=ukw&q=Unity

UPrimitiveComponent. Unreal Engine 4 Documentation. Accessed on 19.09.2020. https://docs.unrealengine.com/en-US/API/Runtime/Engine/Components/UPrimitiveComponent/index.html

Volumes Reference. Unreal Engine 4 Documentation. Accessed on 19.09.2020. https://docs.unrealengine.com/en-US/Engine/Actors/Volumes/index.html

Figures