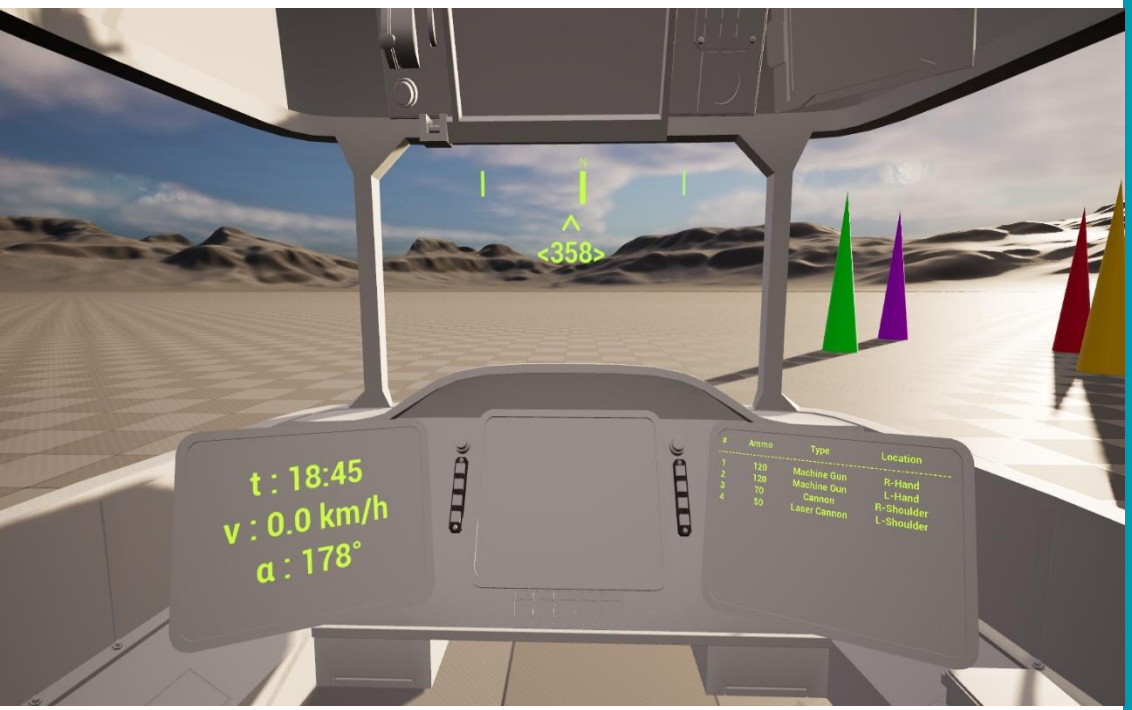
Taavi Korhonen

Diegeettisen HUD:in suunnittelu ja toteutus mecha-peliin Unreal Engine 5 -pelimoottorissa



Tradenomi

Koulutus

Syksy 2024

KAMK • University of Applied Sciences

Tiivistelmä

Tekijä: Korhonen Taavi

Työn nimi: Diegeettisen HUD:in suunnittelu ja toteutus mecha-peliin Unreal Engine 5 -pelimoottorissa

Tutkintonimike: Tradenomi, Tietojenkäsittely

Asiasanat: UI, UE5, Unreal Engine, Diegetic, Mecha

Opinnäytetyössä suunniteltiin ja toteutettiin diegeettinen HUD Unreal Engine 5 -pelimoottorin avulla. Diegeettinen UI tarkoittaa sitä, että UI-elementit on sijoitettuna pelin maailmaan, eikä pelaajan ruudulle.

Tässä työssä käytiin aluksi läpi erilaisia UI-tyyppejä, jotta ero diegeettisen ja muiden UI-tyyppien välillä selkeytyy. Lisäksi tarkasteltiin pelin genreä eli mecha-genreä. Genrestä käytiin läpi sen historiaa, miten genre toimii yleensä peleissä ja miten genre vaikuttaa UI:n suunnittelussa ja tekemisessä. Teoriaosuuden loppupuolella käytiin läpi erilaisia Unreal Engine 5 -pelimoottoriin liittyviä aiheita, jotka olivat työhön tärkeitä, kuten blueprinttien ja C++-ohjelmoinnin eroja, widget blueprint -luokkia ja hieman materiaaleja.

Työn tekovaiheessa käytiin askel kerrallaan läpi, kuinka luotiin diegeettinen kompassi käyttäen Unreal Enginen widget-blueprint-luokkia, jotka sitten tuotiin pelin maailmaan käyttäen blueprint-luokan widget-nimistä komponenttia. Työssä käytiin myös läpi, kuinka käyttäjä pystyy luomaan merkkejä kompassiin, joita voidaan käyttää apuna kertomaan pelaajalle tärkeiden asioiden sijainnista. Kompassin lisäksi työssä käydään nopeasti läpi, kuinka Unreal Engine -moottorilla voi tehdä modulaarisen ase- ja info-widgetin. Ase-widget kertoo pelaajalle tärkeää tietoa mechan aseista ja info-widget kertoo hyödyllistä lisäinfoa, kuten kellon ajan, nopeuden ja kulman, joka mechan ylä- ja alaosan välillä on.

Työn lopputuloksena syntyi mechalle diegeettinen HUD, joka sisältää paljon pelaajalle tärkeää tietoa. Lisäksi työn aikana opittiin, kuinka voidaan tuoda widget blueprint luokat pelin maailmaan käyttäen widget-nimistä komponenttia. Tämän työn tekeminen tulee jatkumaan vielä jatkossa, sillä projekti johon työ liittyy, vaatii muitakin UI-palasia tässä työssä käytyjen lisäksi. Varsinkin työn visuaalinen ilme tulee muuttumaan tulevaisuudessa.

Abstract

Author: Korhonen Taavi

Title of the Publication: Design & Making of Diegetic HUD for a Mecha Game in Unreal Engine 5

Degree Title: Bachelor of Business Administration, Business Information Technology

Keywords: UI, UE5, Unreal Engine, Diegetic, Mecha

The idea behind this thesis was to design and make a diegetic HUD using Unreal Engine 5 game engine. A diegetic HUD means that the pieces of HUD are located in the game's world and not on the player's screen as flat images.

At the beginning, the thesis went over different UI types to differentiate diegetic UI from the rest. After looking at the differences of the different UI types, the thesis studied the genre of the game which is mecha. The thesis looked into the history of the genre as well as how the genre is portrayed in games as well as how the genre affects the design and making of UI. At the end of theoretical part of this thesis, the game engine which was used in this project, Unreal Engine 5, was studied. For Unreal Engine, the thesis went over topics which are important to this thesis. These topics include differences between Unreal Engine's blueprint and C++ programming, widget blueprint classes and a bit about materials.

At the beginning of the practical part of the thesis, it was explained step by step how a diegetic compass is made using Unreal Engines widget blueprint classes which are then brought into the game world using a component called widget which is found in blueprint class. After making the compass, the thesis went over how to create markers that are able to be shown in the compass. These markers can be used to tell the player where some important objects are located using the compass. In addition to the compass, the thesis also showed how to make an ammunition and info widget in Unreal Engine game engine. The ammunition widget shows the player critical information about their weapons, and the info widget shows some additional information like the current time, speed and angle of the mecha.

A working diegetic HUD for a mecha game which includes a lot of important information for the player was the result of this thesis.

Sisällysluettelo

Symboliluettelo

UI	UI on laaja käsite ja tulee sanoista User Interface eli suomeksi käyttöliittymä ja se on osa pelejä, joissa pelaaja on vuorovaikutuksessa pelin kanssa
Diegeettinen	UI-elementti, joka on sisällytetty pelin maailmaan siten, että myös pelattava hahmo voi sen nähdä.
Ei Diegeettinen	UI-elementti, joka näytetään vain pelaajan ruudulla siten, että pelattava hahmo ei voi nähdä sitä.
Meta	Yleensä ei-diegeettisiä UI-elementtejä, jotka antavat pelaajalle lisää tietoa, mikä ei ole välttämätöntä pelaamisen kannalta.
Spatial	Pelimaailmaan sijoitettu UI-elementti, joka antaa pelaajalle lisätietoa rikkomatta pelikokemusta.
HUD	HUD tulee sanoista heads-up display. Se tarkoittaa videopelien yhteydessä yleensä näytöllä olevia 2-ulotteisia elementtejä, jotka kertovat pelaajalle tärkeää infoa
Mecha	suuri, yleensä sotakäyttöön käytetty robotti, jota ohjaa monesti ihminen.
Node	Yksi pala Unreal Enginen blueprint-systeemiä. Nämä voivat olla eventtejä, funktioita tai muuttujia.
Event	Eräänlainen funktio, joka voidaan kutsua, kun halutaan ajaa kyseiseen eventtiin kuuluva funktionaalisuus. (Eventit eivät voi palauttaa arvoja)

1 Johdanto

UI:n suunnitteleminen ja tekeminen voi monille tuntua tylsältä ja kenties jopa turhalta osalta pelin luomisessa, mutta jos UI:n saa tehtyä immersiivisesti ja peliin sopivasti, nostaa se pelin tunnelmaa todella huomattavasti. Unreal Engine 5 -pelimoottorissa UI:n tekeminen onnistuu todella helposti ja vaivattomasti käyttäen Unreal Enginen widget blueprint -luokkia.

Tässä opinnäytetyössä tavoitteena on luoda diegeettinen UI käyttäen Unreal Engine 5 -pelimoottoria ja dokumentoida sen tekemisen vaiheet tarkasti. Työssä ei ole ollut tilaajaa, vaan se on tehty oman tiedon ja taidon kasvattamiseksi.

Työn teoriaosuudessa tullaan käymään läpi useita työhön tärkeitä aiheita. Teoria aloitetaan kertomalla neljästä erilaisesta UI-elementtien tyylistä, mitä mikäkin niistä tarkoittaa ja mihin tarkoitukseen jokainen on tarkoitettu. Näihin kuuluvat diegeettiset, ei diegeettiset, meta-, ja spatiaaliset UI-elementit. Lisäksi tullaan avaamaan projektin genreä eli mecha-genreä ja käymään läpi, mitä mecha tarkoittaa ja miten genre vaikuttaa UI:n suunnitteluun ja tekemiseen. Teoriaosuuden lopuksi käydään läpi Unreal Engine 5 -pelimoottoria. Moottorista käydään läpi, mitä on blueprint-luokat ja C++-ohjelmointi ja käydään läpi eroja blueprint-luokkien ja C++-ohjelmoinnin välillä. Lisäksi Unreal Engine 5 -pelimoottorista käydään läpi myös, mitä on widget blueprintit ja mitä niillä voi tehdä ja miten ne voi tuoda pelimaailmaan. Lopuksi käydään läpi, mitä ovat materiaalit ja mitä niillä voi tehdä Unreal Engine 5 -pelimoottorissa.

Teoriaosuuden jälkeen työssä käydään läpi, kuinka tehdään kompassi-widget, siihen mahdollisia suuntaa antavia merkkejä, pieni info-widget ja aseiden infoa kertova widget. Lisäksi käydään läpi, kuinka edellä mainitut widgetit tuodaan pelimaailmaan käyttäen widget-nimistä komponenttia blueprint-luokissa.

2 Erilaiset UI-elementtien tyypit

Peleissä on monia erilaisia UI-tyyppejä. Yleisimpiä näistä ovat diegeettinen, ei-diegeettinen, meta ja spatiaalinen. Tässä kappaleessa kerrotaan näistä neljästä erilaisesta tyypistä. Mitä ne tarkoittavat ja milloin kyseisiä tyyppejä käytetään?

2.1 Diegeettiset elementit

Diegeettinen UI on yksi neljästä UI-tyylistä ja samalla tähän työhön tärkein, sillä myöhemmin tässä työssä tullaan kertomaan, kuinka tehdään diegeettinen UI Unreal Engine -pelimoottorilla. Diegeettinen UI on kaikista elementeistä immersiivisin UI-elementtien tyyppi, sillä se on sisällytettynä pelin maailmaan, eikä pelkästään 2-ulotteisina kuvina pelaajan ruudulla. Diegeettisestä UI:sta toimii hyvin esimerkkinä joidenkin ajopelien autojen mittaristot. Nykyajan ajopeleissä, joissa voi pelata ensimmäisen persoonan kameralla, on yleensä toimiva mittaristo, joka antaa pelaajalle tarvittavia tietoja. Tietoihin voi kuulua: auton nopeus, auton moottorin kierrosnopeus ja vaihde, jolla auto on [1.]. Kuvasta 1 näkee, kuinka auton mittaristo on luotu toimimaan diegeettisesti [Kuva 1].

Kuva 1. Need for Speed: Shift -peli [2], jossa näkyy auton kojelaudassa diegeettinen mittaristo [3].

2.2 Ei-diegeettinen

Ei-diegeettinen UI-tyyli on nimensä mukaan vastakohta diegeettiseen tyyppiin. Ei-diegeettinen UI-tyyli on yleisin. Tämä tyyli on helpoin tehdä ja myös pelaajien ymmärtää, sillä se on yksinkertainen. Monesti Ei-diegeettinen UI on vain 2-ulotteisia kuvia ja interaktiivisia objekteja ruudulla. Tällöin ne ovat kokonaan erillään pelin tarinasta ja pelattava hahmo ei näe kyseisiä elementtejä. [1.]

Hyvänä esimerkkinä tästä ovat pelien menut ja jotkin HUDit. HUD tulee sanoista heads-up display. Se tarkoittaa videopelien yhteydessä yleensä näytöllä olevia 2-ulotteisia elementtejä, jotka kertovat pelaajalle tärkeää infoa, kuten elämien määrän tai vaikka ammusten määrän aseessa. [4.] Kuvassa 2 näkyy, kuinka HUD on tehty ei-diegeettisesti [Kuva 2]. Joskus pelien kehittäjät haluavat saada heidän valikoistaan hieman immersiivisempiä ja elävämmän näköisiä ja tällöin he voivat tehdä heidän UI-elementeistä hieman 3-ulotteisen näköisiä. Jotkut kutsuvat tätä tyyliä hybridiksi. [1.]

Kuva2. Devil May Cry 5 -pelin [5] HUD joka on tehty ei-diegeettisesti [6].

2.3 Meta

Meta on ehkä vaikein neljästä pää-UI-tyylistä ymmärtää. Tämä on siksi, että meta-tyyliset UI-elementit pystyy helposti sekoittamaan ei-diegeettisiin UI-elementteihin, sillä meta-elementit käytännössä ovat ei-diegeettisiä UI-elementtejä, mutta meta-elementtien tarkoitus on luoda pelistä immersiivisempi kuin perinteisen ei-diegeettisen elementin. Meta-elementit eivät siis ole pakollisia elementtejä pelin pelattavuuden kannalta, mutta ne auttavat pelaajaa uppoutumaan pelin maailmaan. Yksi yleisimmistä tyyleistä käyttää meta-UI-tyyliä on tehdä erilaisia visuaalisia kuvauksia sille, kun pelaaja vahingoittuu. Näistä yleisimpiä on tehdä ruudusta hieman sumea tai verinen kuten kuvasta 3 näkee [Kuva 3]. Ruudun sumeaksi tai veriseksi saaminen on helpoin tapa kertoa pelaajalle siitä, että hän on vahingoittunut ilman, että hänen tarvitsisi katsoa jostakin omia elämäpisteitään. Tällöin pelaaja voi keskittyä paremmin peliinsä. [1.]

Kuva 3. Call of Duty Black Ops III -pelistä [7], jossa meta-elementtiä käyttämällä näytetään pelaajalle, että hän on vahingoittunut [6].

2.4 Spatial

Spatiaalinen UI-tyyli on hyvin informatiivinen UI:n tyyli. Spatiaalisella tyylillä tehdyt UI elementit ovat yleensä sijoitettu pelin maailmaan eikä pelkästään pelaajan ruudulle. Vaikka spatial-elementit ovat sijoitettuna pelin maailmaan, niin pelattava hahmo ei niitä näe, eli spatial-elementit ovat peleissä vain auttamassa pelaajaa. Yleensä spatial-elementit pyritään tekemään mahdollisimman immersiivisesti, mutta aina siihen ei ole mahdollisuutta. [1.]

Spatial-elementit antavat pelaajalle tietoa. Tieto voi kertoa, vaikka missä ja kuinka kaukana jokin tehtävä sijaitsee, vaikka laittamalla kuvan 4 tavalla kenttään merkin, jota pelaaja voi seurata haluttuun kohteeseen. Tai tieto voi kertoa pelaajalle, kuinka suurelle alueelle jokin räjähdys tulee vaikuttamaan laittamalla punaisen merkin vaara-alueelle, jotta pelaaja voi väistää sen ajoissa.

Kuva 4. Cyberpunk 2077 -pelistä [8], jossa näytetään kahdella erinäköisellä spatiaalisella elementillä pelaajalle minne mennä [6].

3 Mecha

Termi mecha on monelle tuntematon tai outo. Mecha on noin 1950-luvulla kehitetty manga- / animegenre. Mechalla yleensä tarkoitetaan suurta panssaroitua ja yleensä ihmisen ohjaamaa sotarobottia, mutta mechalla voidaan tarkoittaa myös muita mekaanisia asioita, kuten kyborgeja ja androideja. Yleensä kaikenlaiset mechat ovat silti hyvin ihmismäisiä. [9.]

Ensimmäisiä mecha genreen luokiteltavia teoksia olivat Osamu Tezukan vuonna 1952 julkaisema Tetsuwan Atom manga, josta myöhemmin tehtiin anime-versio, joka tunnetaan nimellä Astro Boy [10]. Astro Boyn lisäksi vuonna 1956 julkaistu Tetsujin 28-go on myös yksi ensimmäisistä mecha-genren teoksista [11]. Tetsujin 28-go sai vuonna 1963 animen, joka tunnetaan nimellä Gigantor [11].

Mecha-genre on kehittynyt huomattavasti ja se on levinnyt vahvasti myös länteen. Lännessä tunnetuimpia mecha-genreen tehtyjä teoksia ovat 1984 Hasbron Transformers [12] Ja Gainaxin tuottama animesarja Neon Genesis Evangelion [13].

3.1 Peleissä

Yleensä mecha-genren videopeleissä mechojen tarkoituksena on sotia toisia mechoja vastaan sodan välineinä. Mechojen koko ja nopeus monesti vaihtelevat pelien välillä. Esimerkiksi Titanfall-pelisarjassa [14] mechat ovat hyvin ketteriä ja pienikokoisia. Toisin kuin MechWarrior-pelisarjassa [15], jossa mechat ovat yleensä ihmiseen verrattuna todella valtavia ja liikkuvuudeltaan hyvin hitaita ja kömpelöitä. Näistä esimerkeistä voisi kuvitella, että mechan liikkuvuus riippuu mechan koosta, mutta Armored Core -pelisarjassa [16] ohjataan suurta mechaa, joka on kokoonsa nähden todella ketterä. Yleensä mechan liikkuminen on enemmän riippuvainen siitä, millainen tunnelma mechaan ja peliin halutaan.

Mecha-peleissä todella suuressa osassa on myös oman mechan muokkaaminen. Yleensä pelit antavat pelaajan valita, millaisia aseita ja panssaria he haluavat heidän mechalleen [Kuva 5]. Mechan muokkaaminen yleensä vaikuttaa myös mechan liikkuvuuteen. Jos mechoilla on vahvat panssarit ja järeät kanuunat, tulevat ne liikkumaan paljon hitaammin kuin sellaiset mechat, joilla on minimalistinen panssarointi ja pienet tykit. Mechojen aseistuksen muokkaamisen lisäksi mechojen visuaalinen muokkaaminen on myös suuressa osassa mecha-pelejä. Visuaalisen muokkaamisen avulla pelaaja voi paremmin uppoutua pelin maailmaan tai tehdä mechastaan vaikka hauskan näköisen.

Kuva 5. Armored Core VI -pelin [16] mechan aseiden muokkausvalikko [6].

3.2 Genren vaikutus UI:n suunnittelussa

UI on yleensä mecha-peleissä hyvin futuristisesti suunniteltu, sillä nämä pelit sijoittuvat monesti useita vuosia tulevaisuuteen. Koska mecha-pelit ovat yleensä todella monimutkaisia, niin tämän takia UI on useasti myös täynnä kaikenlaista infoa. Tähän infoon monesti tärkeimpänä kuuluu mechan elämäpisteet, jotka on myös monesti jaettu mechan eri osille.

Esimerkiksi mechan käsillä, keholla ja jaloilla voi olla eri elämäpisteet. Kun yhden osan elämäpisteet menevät tyhjiin, voi tämä osa lakata toimimasta ja kun mecha on ottanut tarpeeksi vahinkoa, tuhoutuu se kokonaan. Lisäksi mechan aseiden ammukset ovat toinen tärkeä asia pelaajan tietää. Lista eri aseiden ammuksista voi monesti viedä suurenkin osan pelaajan näytöstä, jos hänen mechallaan sattuu olemaan useampi ase, jotka kaikki käyttävät erilaisia ammuksia. Näiden tietojen lisäksi ruudulla voi olla myös muuta tarvittavaa tietoa, esimerkiksi kompassia, karttaa ja tehtävän tiedot, jolla mecha on.

Jossakin vaiheessa, kun mechan HUD on täytetty kaikenlaisella tiedolla voi uudella pelaajalla olla hankaluuksia ymmärtää kaikkea tietoa, mitä hänelle annetaan ja tämä voi monesti olla esteenä mahdollisille uusille pelaajille pelin aloittamisessa. Tästä esimerkkinä kuva 6, jossa näkyy Armored Core VI -pelin [16] HUD, joka on täynnä kaikenlaista tietoa. Tämän takia monet pelit rajoittavat tiedon määrää ruudulla, esim. tiivistämällä infoa, jotta se veisi pienemmän määrän tilaan HUD:ista. Tällä ilmiöllä voi joissakin tapauksissa olla päinvastainen vaikutus. Kun tietoa pyritään tiivistämään, niin silloin sen ymmärrettävyys voi kärsiä.

Kuva 6. Armored Core VI -pelin [16] HUD [6].

4 Pelimoottori

Tähän työhön pelimoottoriksi on valittu Unreal Engine 5.3 -pelimoottori. Unreal Engine on Epic Gamesin tekemä pelimoottori, mitä voi käyttää kaikenlaisen median tuottamiseen. Unreal on monelle tuttu videopeleistä, mutta sillä voi tehdä myös animaatioita tai sitä voi myös käyttää erilaisessa suunnittelutyössä. [17.] Unreal on monelle hyvä pelimoottorivalinta, sillä se on ilmainen ja Unreal Engineä on helppo ja nopea oppia käyttämään.

Unreal Enginessä on huonojakin puolia. Ainakin blueprinttien puolella dokumentointi on yleensä hyvin puutteellista, minkä takia monesti joutuu ongelmien tullessa mennä lueskelemaan erilaisia foorumeja ratkaisun toivossa. Lisäksi pelimoottori on täynnä kaikenlaisia valmiiksi tehtyjä asioita, jotka ovat helppouden takia tehty, mutta nämä nostavat moottorin kansion tilavuutta huomattavasti.

Unreal Enginessä voi tehdä funktionaalisuutta kahdella erilaisella tavalla. Monille helpompi, mutta myös toimivuudeltaan hitaampi Blueprint on yleisin, tapa tehdä asioita Unrealissa, toinen on C++-koodaus, joka on optimoidumpaa, mutta monimutkaisempaa.

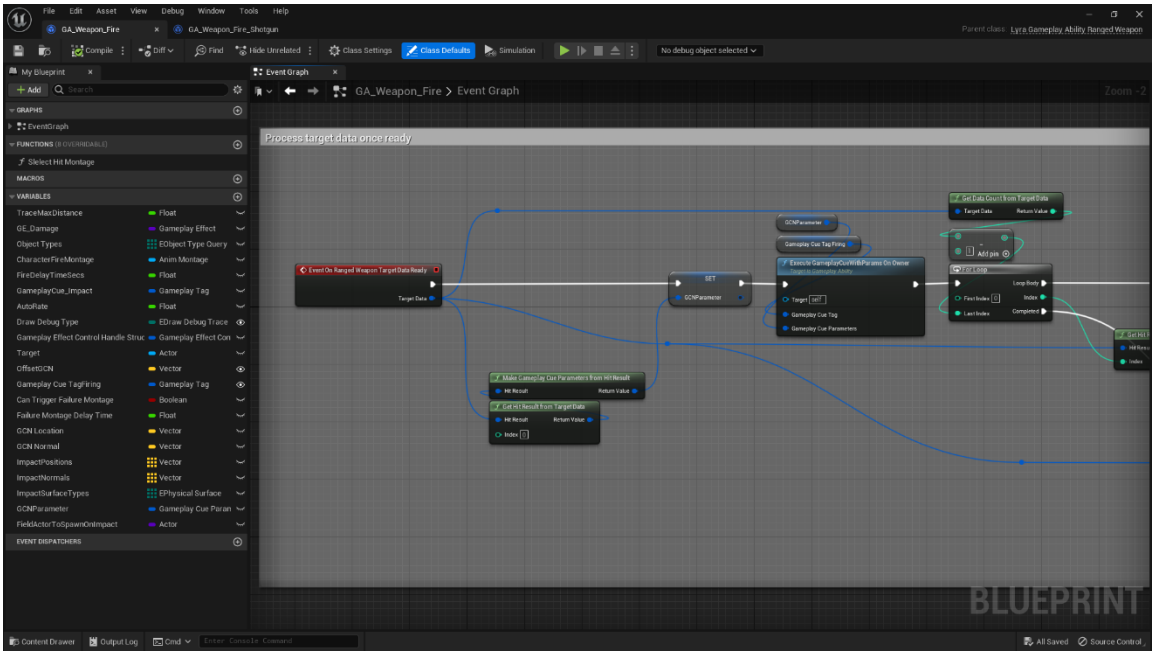
4.1 Blueprint Unreal Engine -moottorissa

Blueprintit ovat Unreal Enginen oma node-pohjainen ohjelmointikieli [19]. Koska blueprintit ovat node-pohjainen ohjelmointikieli, niin se toimii ihan eri tavalla normaaleihin koodauskieliin verrattuna.

Blueprintit sisältävät erilaisia nodeja eli laatikoita, joilla jokaisella on oma funktionaalisuus. Näitä nodeja voi yhdistellä niissä olevien sisään- ja ulostulotäppien avulla. Nodeja yhdistelemällä voi käyttäjä helposti luoda erilaisia blueprint-luokkia. [Kuva 7.]

Blueprintit ovat helppoja ja nopeita käyttää ja sen takia ne sopivat hyvin aloitteleville Unreal Engine -käyttäjille tai nopean prototyypin tekemiseen. Suurissa projekteissa blueprinttien pääsääntöinen käyttäminen ei ole suositeltavaa, sillä blueprintit ovat paljon vaativampia kuin suora C++-ohjelmointi, sillä kun blueprint-luokkaa ajetaan niin Unreal kääntää sen C++-koodiksi. Tämän takia on suositeltavaa, että suuremmissa projekteissa käytettäisiin mieluummin C++: aa. [18.]

Lisäksi blueprinttien avulla datan tallentaminen ja lataaminen on helpompaa, sillä blueprinttejä käyttäessä käyttäjällä ei ole niin suurta riskiä tehdä vahingossa jotakin väärin [19].



Kuva 7. Unreal Enginen blueprint-systeemistä. [18]

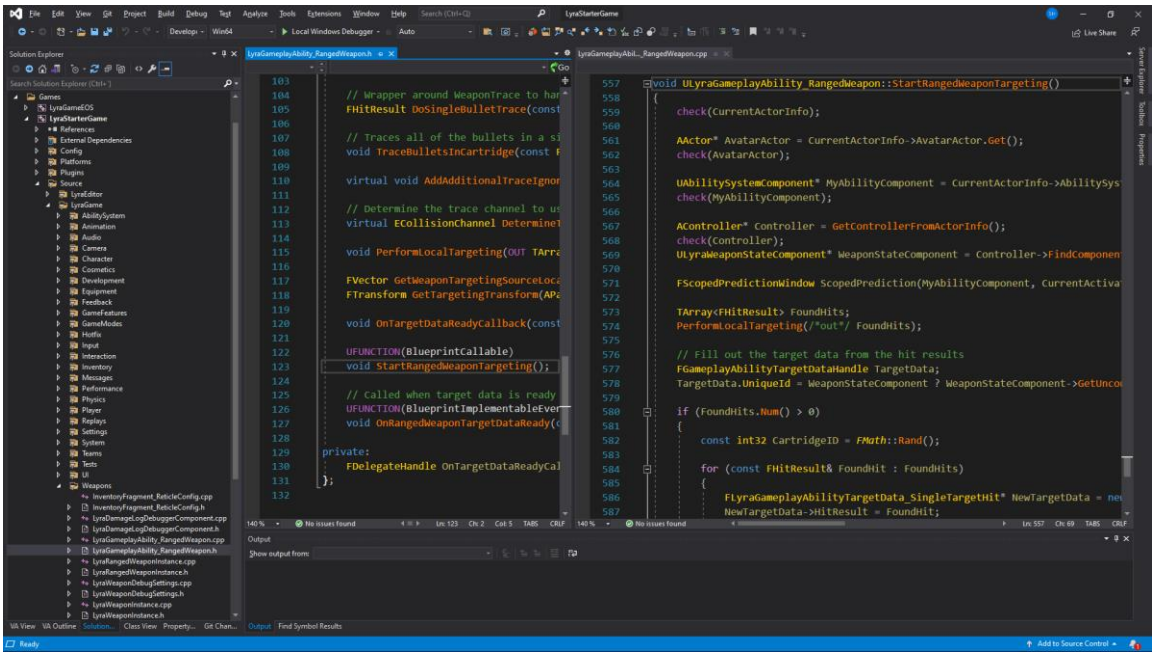
4.2 C++ Unreal Engine -moottorissa

C++:lla ohjelmointi on toinen tyyli tehdä funktionaalisuutta Unreal Engine -pelimoottorissa. C++:n avulla käyttäjä pääse syvemmin käsiksi siihen, mitä Unrealilla on käyttäjälle tarjota. Käyttäjä voi halutessaan mennä muuttamaan suoraan Unreal Enginen lähdekoodia, jos hän niin haluaa tehdä. [18.]

Vaikka C++ voi kuulostaa paljon paremmalle vaihtoehdolle, on se huomattavasti haastavampi ja hankalammin lähestyttävä tapa kuin blueprintit. Unrealin C++ ei ole suoraan C++:lla ohjelmointia, vaan Unrealin oma versio C++-koodikielestä, joka toimii lähes samalla tavalla kuin normaali C++, mutta siinä on pieniä eroja.

C++ on monesti paras vaihtoehto, jos käyttäjä haluaa tehdä monimutkaisia funktioita, sillä C++:n saa optimoitua paljon paremmin kuin perinteiset blueprint-luokat. Lisäksi jos on tekemässä verkon yli toimivaa peliä, tällöin on paljon parempi tehdä tärkeät funktiot C++:n avulla, sillä se takaa, että peliin ei tule mitään ylimääräisiä hidasteita, mitä voisi tulla, jos saman tekisi blueprinttien avulla. [18.]

Monesti projekteissa ei ole fiksua käyttää pelkästään C++-ohjelmointia tai blueprint-luokkia, vaan käyttäjän on hyvä oppia käyttämään molempia paikoissa, joissa toinen soveltuu tehtävään paremmin kuin toinen [18].



Kuva 8. Visual Studio -ohjelma, jossa on auki C++-tiedosto. [18]

4.3 Widget blueprintit

Widget-blueprintit tai widgetit ovat tapa tehdä UI:ta Unreal Enginessä. Kuten nimestä voi päätellä, widgetit toimivat Unrealin node-pohjaisella blueprint-systeemillä. Sillä UI sisältää yleensä vain 2-ulotteisia simppeleitä elementtejä on UI:n tekemiseen suotavaa ja jopa kannattavaa käyttää blueprinttejä.

Widgeteillä käyttäjä voi luoda kaikki haluamansa UI:t, kuten päävalikon, pelaajan HUD:in tai vaikka kaupan, josta pelaaja voi ostaa esineitä. Käytännössä voisi sanoa, että widgetit ovat vain 2-ulotteisia blueprint-luokkia, kun taas perinteiset blueprint-luokat toimivat 3 ulottuvuudessa, mutta tämä ei tarkoita sitä, että widgettejä ei voisi tuoda kolmanteen ulottuvuuteen.

Blueprint-luokan alla on olemassa komponentti nimeltä ”Widget Component”. Tämä komponentti mahdollistaa perinteisten 2-ulotteisten UI-elementtien tuomisen pelin 3-ulotteiseen maailmaan [20]. Widget Component on tärkeä komponentti varsinkin silloin, kun käyttäjä haluaa tuoda vaikka syvyyttä UI:hinsa.

4.4 Materiaalit

Materiaalit määrittelevät sen, miten pelin sisäisten esineiden pinnat käyttäytyvät. Materiaalien avulla käyttäjä voi määrittää esimerkiksi esineen pinnan värin, heijastavuuden, töyssyisyyden tai vaikka läpinäkyvyyden. [21.]

Materiaalien muokkaus toimii Unreal Enginen blueprint-systeemillä. Materiaalien muokkaus mahdollistaa monien erilaisten materiaalien luomisen. Monen samankaltaisen materiaalin luominen ja muokkaus vievät paljon aikaa. Unreal Engine mahdollistaa ”Material Instancing” -ominaisuuden, joka mahdollistaa ”Parentin” eli emomateriaalin, jota voi käyttää pohjana ”Childreneissä” eli lapsimateriaaleissa. Nämä niin sanotut lapsimateriaalit perivät emomateriaalin ominaisuudet, joita voi sitten muokata lapsimateriaalissa. [22.]

5 Tekovaihe

Tässä luvussa käydään läpi suunnitelma diegeettiselle UI:lle ja luodaan se vaihe vaiheelta Unreal Engine 5 -pelimoottorissa.

5.1 Suunnitelma

Tarkoituksena on tehdä diegeettinen HUD valtavalle sota-mechalle. Inspiraatiota mechan hudille otettiin nykyaikaisten hävittäjälentokoneiden HUD:eista [Kuva 9]. Tärkeää oli kumminkin muistaa se, että osa siitä, mitä hävittäjälentäjä tarvitsee lentäessään, on aivan turhaa mechakuskille. Lisäksi tarkoituksena oli tehdä jokainen elementti siten, että sen voi sijoittaa mechan ”tuulilasiin” käytännössä mihin käyttäjä sen haluaa. Tämä pieni lisäys lisää huomattavasti pelaajan immersiota varsinkin, jos hän pelaa käyttäen VR-laseja.

Kuva 9. Simuloitu kuva F/A18C-hävittäjän HUDista [23]

Kuva 9 on hyvä lähtökohta siitä, mitä hävittäjälentäjän HUDissa näkyy. Lisäksi nettisivulla, josta kuvan otin, on hyvin kerrottu, mitä mikäkin numeroiduista kohdista tarkoittaa. Näistä ehkä tärkeimpänä on kuvassa 9 numerolla 3 merkitty kompassi.

Kompassi on monessa pelissä tärkeä UI-elementti, sillä sen avulla pelaaja tietää tarkalleen, mihin suuntaan hän on katsomassa. Lisäksi kompassiin voidaan liittää erilaisia merkkejä, jotka voivat osoittaa, vaikka tehtävän seuraavan osan sijainnin tai kertoa mistäpäin viholliset ovat ampumassa.

Kuvassa 9 numerolla 2 & 9 on merkitty eri tavalla hävittäjän nopeus. Tämä on hyvä tieto ohjaajan tietää varsinkin mechoissa, joissa ohjaaminen voi toimia siten, että yksi vipu vastaa nopeudesta, jolloin nopeuden voi asettaa ja se voi joissakin tapauksissa unohtua.

Lisäksi tärkeänä osana, mitä ei kuvassa 9 ole mainittu, on ammusten määrä. Tämä on todella tärkeä tieto varsinkin, kun on kyse sota-mechasta. Mechalla voi olla useita erilaisia aseita, joiden ammustilannetta on hankala pitää muistissa varsinkin, jos eri aseet käyttävät erilaisia ammustyyppejä.

5.2 HUDin suunnittelu

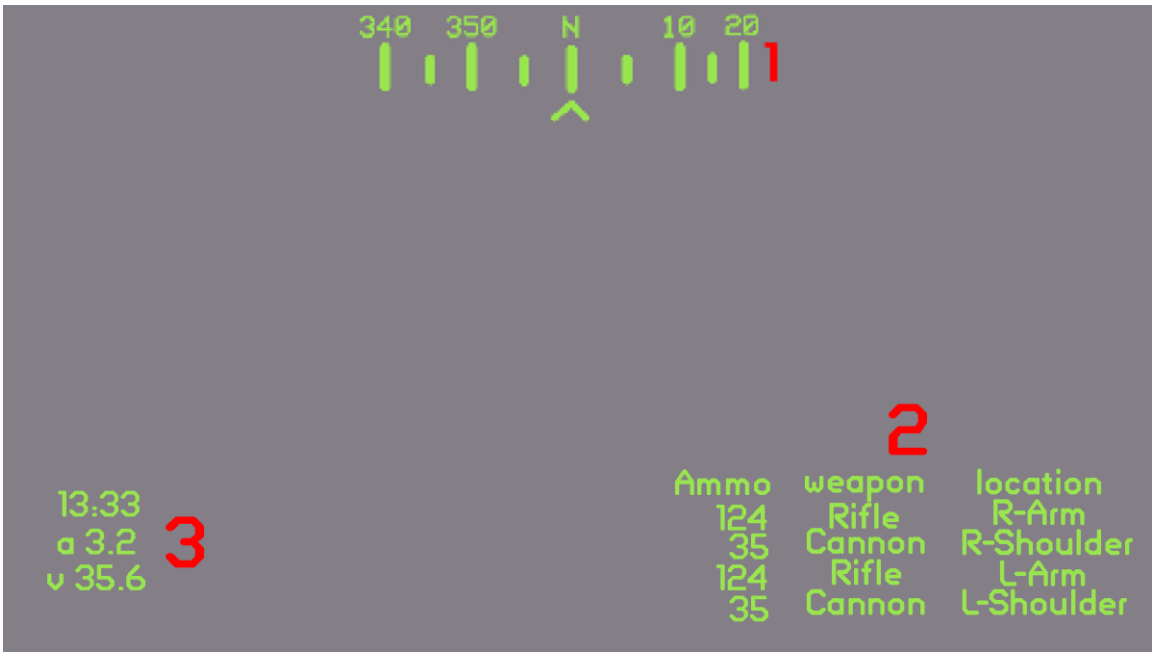
Kuvassa 10 näkyy suunnitelma siitä, mitä HUD tulisi alustavasti sisältämään. Kuvassa 10 on merkitty elementit, jotka tässä työssä tehdään punaisilla numeroilla [Kuva 10].

Kuvassa 10 punaisella numerolla 1 on kuvattu kompassi. Kompassin olisi tarkoitus näyttää, mihin suuntaan mechan yläosa katsoo maailmaan nähden. Tähän kompassiin on myös tarkoitus tehdä merkkejä, jotka voivat näyttää pelaajalle tärkeitä kohteita. [Kuva 10.]

Kuvassa 10 punaisella numerolla 2 on kuvattu mechan aseiden antamat infot. Tässä kerrotaan, mitä aseita mechalla on, paljonko ammuksia missäkin aseessa on ja missä kyseinen ase sijaitsee mechassa. [Kuva 10.]

Viimeisenä kuvassa 10 punaisella numerolla 2 on kuvattu 3 kappaletta lisäinfoa kertovia elementtejä. Nämä ovat ylhäältä alas.

- Kello, joka voi kertoa joko pelaajan laitteen ajan tai sitten tehtävään liitetyn ajan.
- Mechan kulma kertoo mechan ylä- ja alaosan välisen kulman.
- Mechan nopeus on luku, joka kertoo pelaajalle, kuinka nopeasti hänen mechansa liikkuu.



kuva 10. Aikainen suunnitelmakuva HUD:ista

5.3 Kompassin tekeminen

Suunnitelmani kompassin tekemiselle on seuraava: Aion aluksi luoda väliaikaiset kuvat kompassiin. Nämä tullaan mahdollisesti vaihtamaan projektin myöhemmissä vaiheissa. Kunhan kuvat ovat valmiita, teen niitä käyttämällä materiaalin. Aion käyttää materiaalia widgetissä ja laittaa sen liikkumaan. Kunhan kompassi itsessään on valmis, aion tehdä merkkiluokan, jonka avulla voi lisätä useita merkkejä kompassiin näyttämään suuntaa. Lopuksi tuon nämä vielä ruudulta pelimaailmaan käyttäen blueprint-luokan widget-komponenttia.

5.3.1 Kompassi-materiaalin tekeminen

Kompassiin tarvittavat kuvat aion tehdä käyttämällä Aseprite-nimistä [24] sovellusta, sillä se on minulle tutuin sovellus kuvien tekemiseen ja saan tarvittavat kuvat tehtyä sen avulla nopeasti.

Kuvassa 11 on tekemäni väliaikainen kuva kompassista [Kuva 11]. Tässä kuvassa on merkitty viivat jokaisen 15 asteen välillä ja lisäksi olen merkinnyt pääilmansuunnat. Kuva 12 on liukuväristä, jonka olen tehnyt saavuttaakseni häivytysefektin kompassin reunoille [Kuva 12].

Kuva 11. tekemäni väliaikainen kuva kompassille.

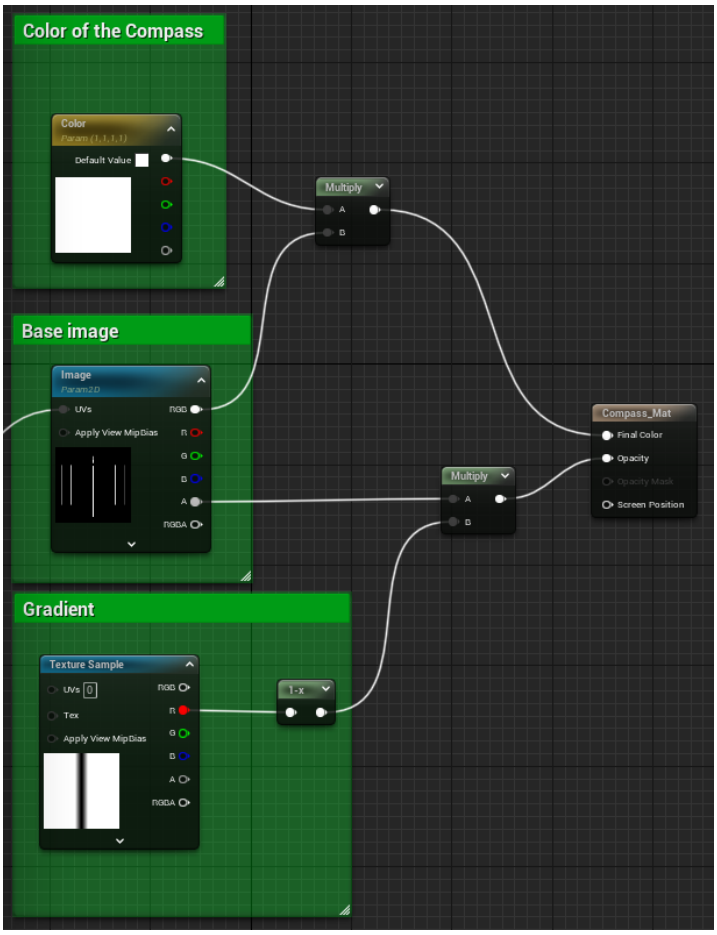
Kuva 12. Kompassin haalistumisliukukuva.

Kuvassa 13 olen tehnyt materiaalin tekemästäni kompassi kuvasta [Kuva 11] ja olen lisäksi tuonut häivytysliukukuvan [Kuva 12]. Olen ottanut liukukuvan punaisen kanavan ja lisännyt siihen noden nimeltä ”OneMinus”, joka on node, jossa lukee ”1-x”. Tämä node kääntää vain gradientin valkoisen ja mustan värin toisin päin. ”OneMinus”-nodea tarvitaan siksi, että Unreal Enginen läpinäkyvyys toimii siten, että valkoinen näytetään ja musta ei ja tekemäni kuva [Kuva 12] on tehty juurin päinvastoin. [Kuva 13.]

Olen sitten yhdistänyt ”OneMinus”-noden kompassi kuvan ”Alpha”-kanavan eli läpinäkyvyys kanavan kanssa ”Multiply”-nimiseen nodeen. ”Multiply”-node mahdollistaa kompassin kuvan [Kuva 11] läpinäkyvyys kanavan ja ”OneMinus”-noden yhdistämisen yhdeksi arvoksi. Lopuksi vielä yhdistin ”Multiply”-noden antaman arvon materiaalin ”Opacityä” eli opasiteetti kohtaan. [Kuva 13]

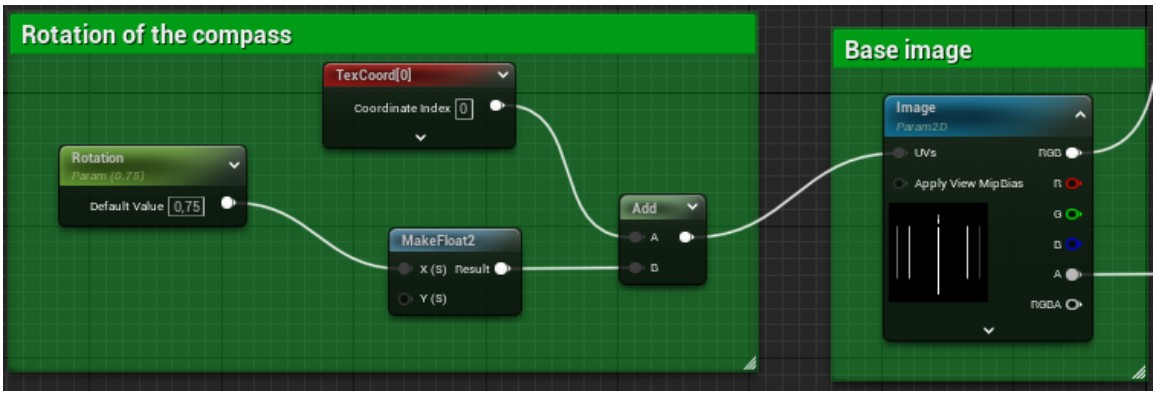
Materiaalin ”Opacity” eli opasiteetti vastaanottaa arvoja, jotka kertovat materiaalille, missä kohtaa materiaali on läpinäkyvä. Nämä arvot ovat numeraalisesti asteikolla 0–1, joista 0 on täysin läpinäkyvä ja 1 täysin näkyvä. Arvot 0–1 voidaan antaa myös mustina ja valkoisina arvoina, joista musta on täysin läpinäkyvä ja valkoinen täysin näkyvä. [Kuva 13]

Kompassin värin muuttamiseksi olen ottanut kompassin kuvan [Kuva 11] RGB-arvon ja liittänyt sen ”Multiply”-nodeen ”Color”-nimisen vektoriparametrin kanssa. Koska kompassin kuva [Kuva 11] on täysin valkoinen ja musta, sen väri voidaan muuttaa kertomalla siihen vektori 3:n arvo. Ja koska ”Color” on vektoriparametri niin sitä voi muokata materiaalin ulkopuolelta helposti, joten kompassin värin muokkaaminen tulevaisuudessa blueprint-luokan sisältä on mahdollista. [Kuva 13]



Kuva 13. Kompassimateriaalin alkuvaihe

Kuvassa 14 olen ottanut ”TexCoord”-noden. Tämä node kertoo, missä kohtaa tekstuuri on materiaalissa. Olen lisännyt ”TexCoord”-nodeen uuden tekemäni float2-arvon. float2 tai Vector2 on muuttuja, joka sisältää 2 eri float eli liukulukumuuttujaa. Olen antanut float2 x -muuttujaan ”Rotation”-nimisen liukulukuparametrin. Muuttamalla ”Rotation”-parametrin arvoja voin liikuttaa tekstuurin sijaintia x-asteikolla. Lopuksi olen antanut float2:n ja ”TexCoordin” yhdistetyn arvon kompassikuvan [Kuva 11] UV-sisääntuloon. [Kuva 14]

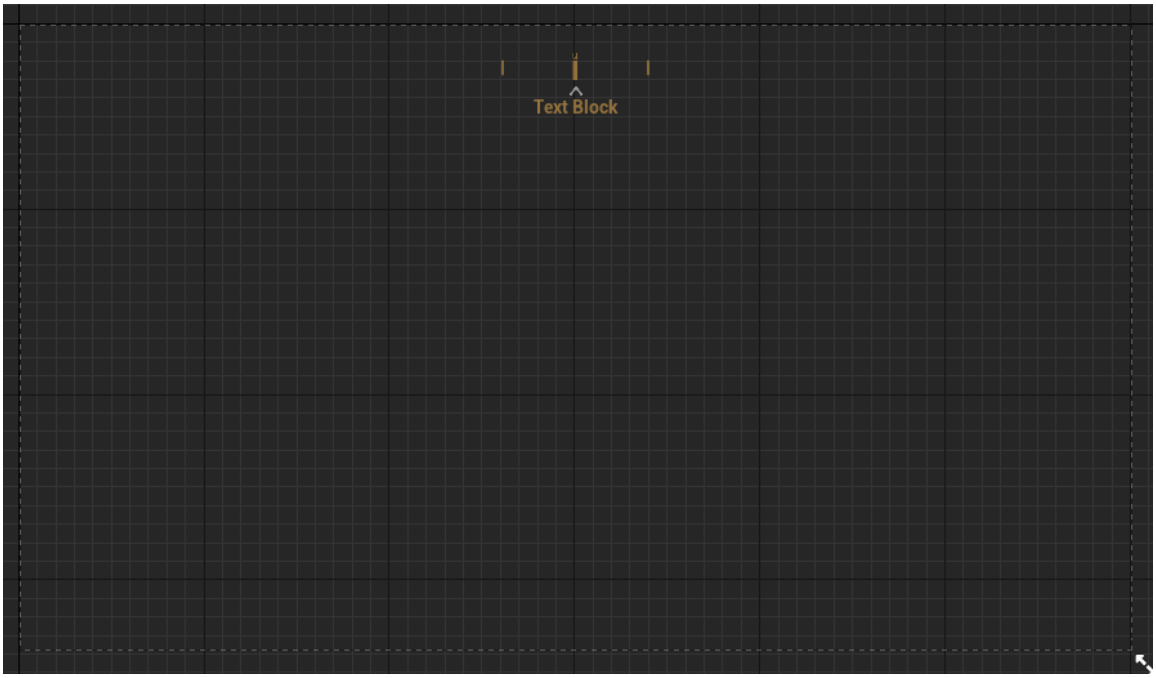


Kuva 14. Kompassimateriaalin liikuttaminen

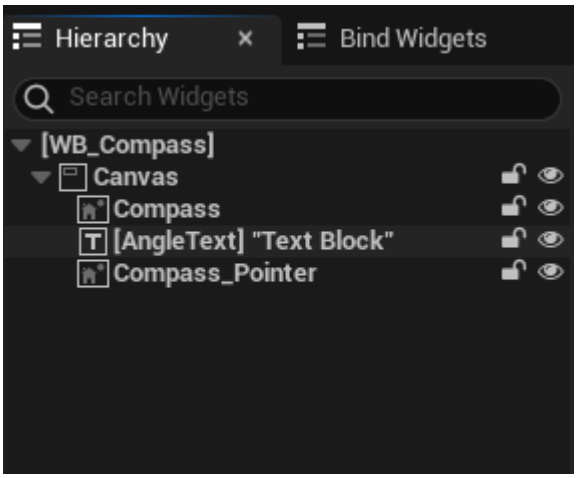
5.3.2 Kompassi widgetin tekeminen

Kompassin materiaalin tekemisen jälkeen loin kompassille uuden widget blueprint -luokan. Mitä widget blueprintit on ja mitä ne tekevät, käytiin läpi kappaleessa 4.3.

Kun olin luonut widgetin kompassille, lisäsin siihen 2 kuvaobjektia ja 1 tekstiobjektin, sijoitin ne haluamallani tavalla [Kuva 15] ja nimesin ne kuvan 16 mukaan [Kuva 16]. Ylimpään kuvaobjektiin laitoin kiinni tekemäni materiaalin kompassista. Toiseen kuva objektiin lisäsin materiaalin ylöspäin osoittavasta nuolesta. Näiden kahden kuva objektin alle laitoin luomani tekstiobjektin. Tekstiobjektin tarkoituksena on kertoa pelaajalle asteen tarkkuudella, mihin suuntaan mechan yläosa katsoo.



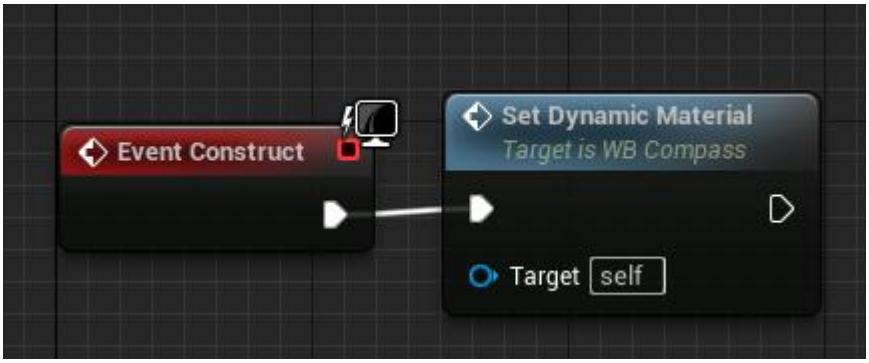
Kuva 15. Kompassi-Widget



Kuva 16. Kompassi-widgetin ”Hierarchy” eli hierarkia.

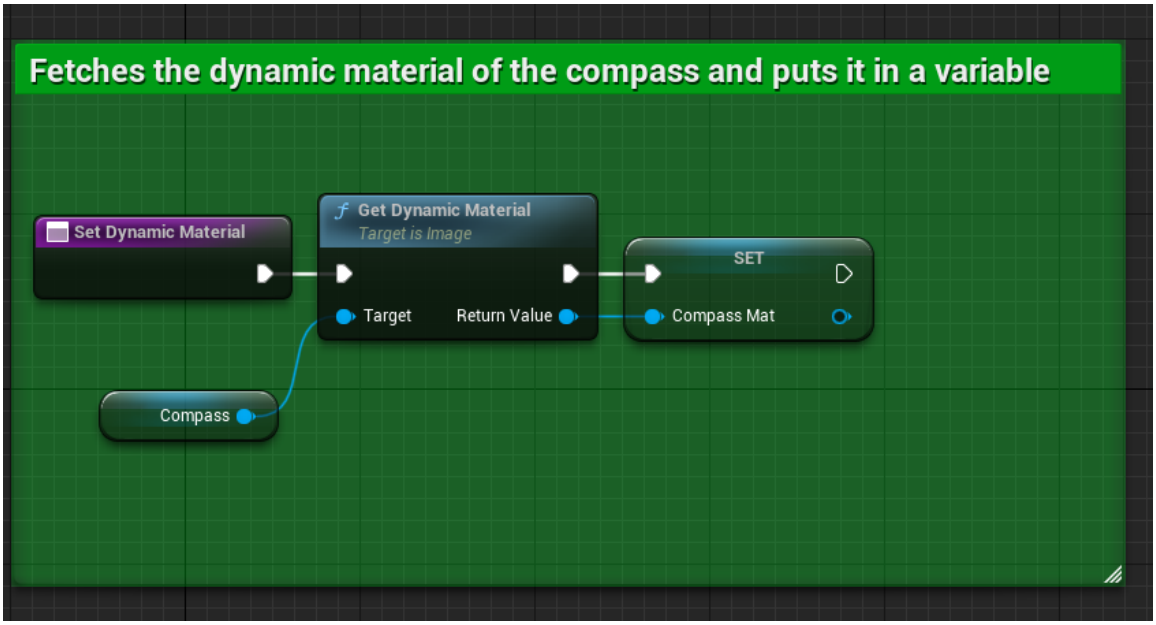
Kompassi-widgetin ”Graph” eli kaaviopuolella olen liittänyt ”Event Construct”-nimiseen nodeen oman tekemäni ”Set Dynamic Material”-funktion. [Kuva 17]

Unreal Enginessä widget-blueprinteissä “Event Construct” on node, johon voidaan kiinnittää muita nodeja siinä järjestyksessä, jossa kyseiset nodet halutaan ajaa suorituksen aikana. ”Event Construct” on node, joka ajetaan silloin, kun kyseinen widget käynnistetään pelimuodossa ensimmäistä kertaa. Blueprint-luokissa ”Event Construct”-nodea vastaava node on ”Event BeginPlay” ja Unityssä vastaava on ”Start”-funktio.



Kuva 17. Kompassi-widgetin ”Event Construct”-metodi.

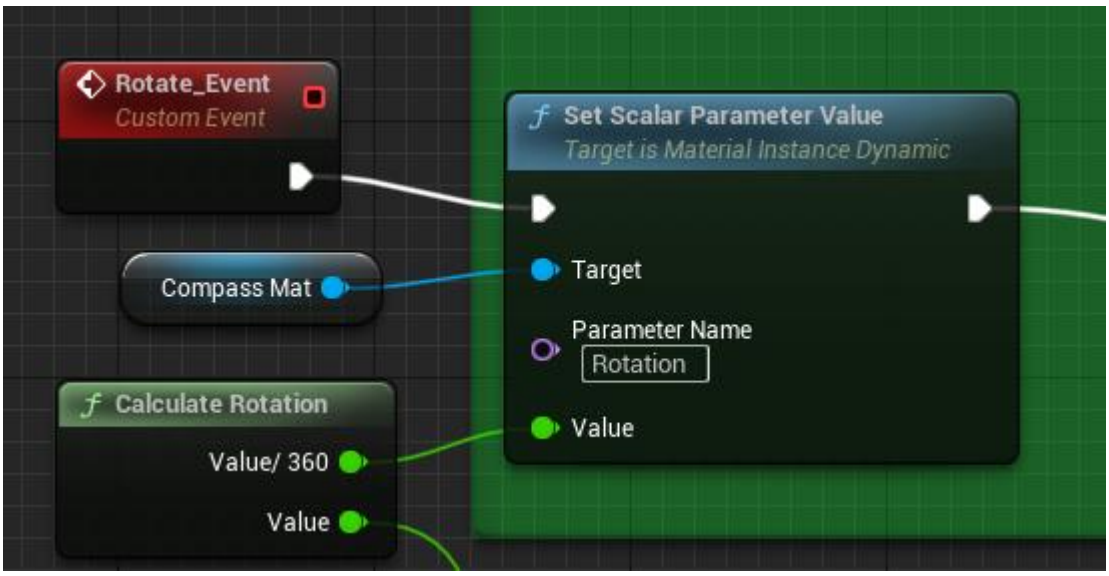
Kuvassa 18 olen tehnyt aiemmin näyttämäni [Kuva 17] ”Set Dynamic Material”-funktion. Se ”Set Dynamic Material” -funktio sisältää viittauksen widgettiin tekemästäni kuvaobjektista ”Compass”, joka sisältää tekemäni kompassimateriaalin. Tästä viitteestä voin hakea ”Get Dynamic Material”-nimisen noden. ”Get Dynamic Material”-node mahdollistaa viittauksen saamisen tekemääni kompassin materiaalin. Otan viittauksen talteen muuttujaan nimeltä ”Compass Mat” käyttäen kyseisen muuttujan ”Set”-nodea. Tämä viittauksen tallentaminen muuttujaan materiaalissa olevien parametrien löytämisen ja niiden muokkaamisen. [Kuva 18.]



Kuva 18. ”Set Dynamic Material”-funktio

Kuvassa 19 olen luonut uuden ”Custom Event” eli itse luomani eventin kompassin widgetin sisälle. Unreal Enginessä käyttäjän itse luomat eventit eroavat esimerkiksi ”Event Tick”- tai ”Event BeginPlay”-eventeistä siten, että moottori ei kutsu itse tehtyjä eventtejä itse, ellei sitä ole käsketty erikseen. Itse tehdyt eventit on käteviä silloin, kun käyttäjä haluaa ajaa osan koodia, jonka ei tarvitse palauttaa mitään arvoa. [Kuva 19.]

Olen kiinnittänyt ”Rotate”-eventnodeen ”Set Scalar Parameter Value”-noden. ”Set Scalar Parameter Value”-noden avulla voin muokata parametrejä, joita dynaaminen materiaali mahdollisesti sisältää. Tässä tapauksessa muokkaan kompassimateriaaliin tekemääni ”Rotation”- eli pyörähdysparametriä [Kuva 14]. Annan parametrille arvoksi liukulukuarvon, jonka lasken itse tekemälläni ”Calculate Rotation”-nimisellä funktiolla. [Kuva 19.]



Kuva 19. Kompassi-widgetin ”Rotate”-event.

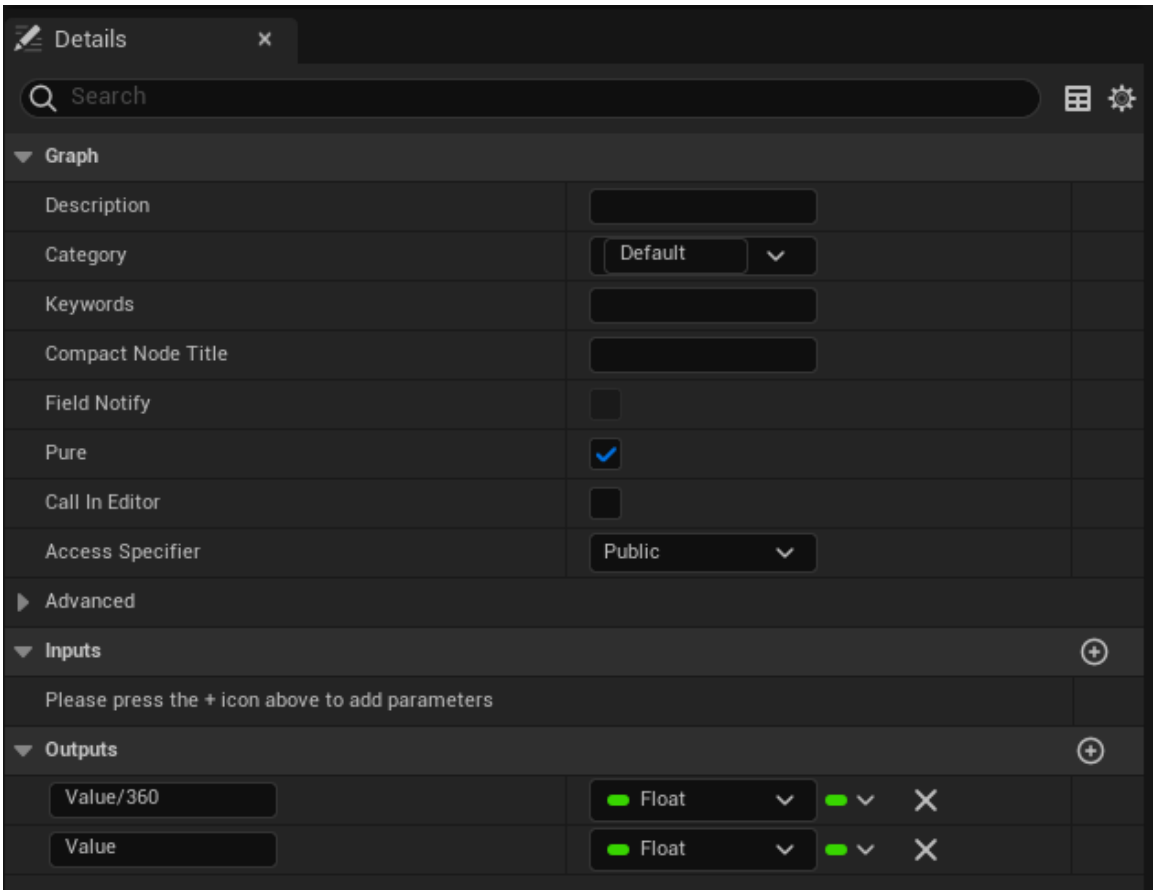
Kuvassa 20 olen tehnyt funktion, jonka avulla haen pelaajan pyörähdyskulma Z-akselilla. Kulman pelaajasta saa hakemalla ”Get Player Controller”-noden. Tämä node palauttaa viitteen kyseisen kentän pelaajaa ohjaavasta luokasta. Tämä pelaajaa ohjaava luokka sisältää ”Get Control Rotaion”-noden, joka antaa pyörähdyksen ulos Unreal Enginen omalla ”Rotator”-muuttujalla. Mutta koska tarvitsen vain pyörähdyksen Z-akselilla liukulukuna, niin painoin ”Rotator”-muuttuja ulostuloa hiiren oikealla näppäimellä ja valitsin ”Split Struct Pin”-vaihtoehdon. ”Split Struct Pin” vaihtaa ”Rotator”-ulostulon kolmeksi liukuluvuksi, joista yksi on haluamani Z-akselin pyörähdys. [Kuva 20.]

Koska ”Get Control Rotation”-node antaa arvon asteina 0–360, mutta materiaali käyttää asteikkoa 0–1. Päästääkseni 0–1 asteikolle olen jakanut Z-akselin pyörähdyksen luvulla 360. Tämän jälkeen olen funktion ”Return”-nodeen antanut kaksi ulostuloa. Toinen näistä on laskemani pyörähdys 0–1 asteikolla ja toinen on pyörähdys 0–360 asteikolla. [Kuva 20.]

Huomioon otettava asia tässä on se, että otan pyörähdyskulman vielä pelaajasta enkä mechasta, sillä projektin mecha ei ole vielä siinä kunnossa, että siitä voitaisiin katsoa kulma.

Kuva 20. ”Calculate Rotation”-funktio.

Jotta olen saanut ”Calculate Rotation” funktion toimimaan ilman ”Exec”- eli suorituslinjaa [Kuva 19], olen tehnyt funktiosta niin sanotusti puhtaan ”Pure”-funktion. ”Pure”-ominaisuuden olen laittanut funktiossa päälle funktion ”Details”-osiosta. [Kuva 21] Funktion muuttaminen ”Pure”-funktioksi ei ole pakollista sen toiminnan kannalta, mutta ”Pure”-muoto on paljon siistimpi lähestymistapa kuin normaali funktio. Unreal Engine osaa ajaa ”Pure”-funktiot silloin, kun niitä tarvitaan.

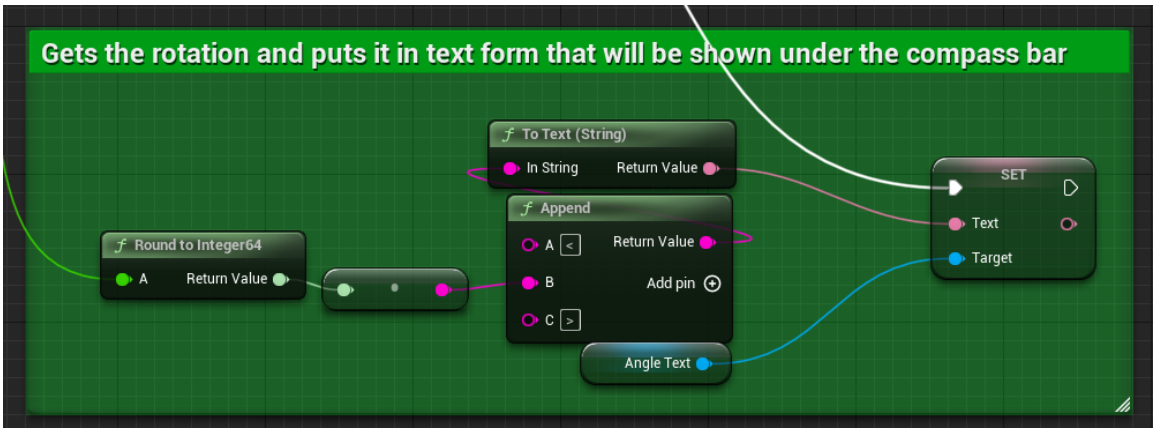


Kuva 21. ”Calculate Rotation”-funktion ”Details”-osio.

Kuvassa 22 otan ”Calculate Rotation”-funktion palauttaman ”Value”-arvon eli kulman asteikolla 0–360 ja muutan sen tekstiksi. Tämä teksti tulee lopuksi näkymään kompassi-widgettiin tekemässäni tekstiobjektissa [Kuva 16].

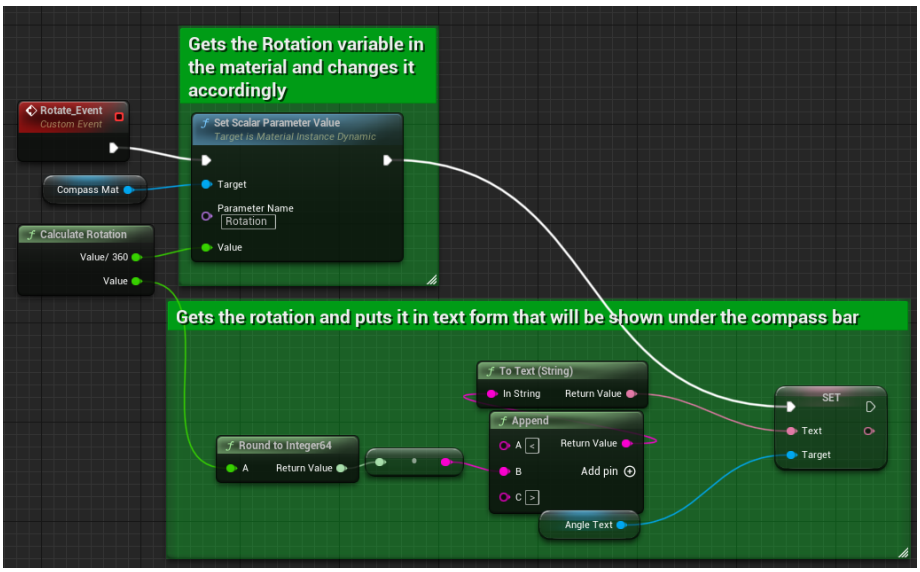
Aluksi olen pyöristänyt ”Calculate Rotation”-funktion antaman liukuluvun kokonaisluvuksi käyttämällä ”Round to Integer64”-nodea. Tämän jälkeen olen luonut ”Append”-noden. ”Append”-node mahdollistaa useiden merkkijonomuuttujien ”String”-muuttujien yhdistämisen yhdeksi merkkijonomuuttujaksi. Olen vetänyt ”Round to Integer64”-noden ulostulon ”Append”-noden

toiseen sisääntuloon. Unreal Engine osaa laittaa automaattisesti noden, joka muuntaa kokonaisluvut merkkijonoksi. Olen lisännyt ”Append”-noden ensimmäiseen ja kolmanteen sisääntuloon ”<” ja ”>” merkit koristeeksi. Lopuksi olen hakenut viitteen tekstiobjektiin [Kuva 16], josta olen hakenut ”Set Text”-nimisen noden. ”Set Text” mahdollistaa tekstiobjektien tekstikentän sisällön muokkaamisen blueprint-luokan sisällä. Olen liittänyt ”Set Text”-noden ”Text”-sisääntuloon ”Append”-noden ulostulon, johon Unreal Engine osaa automaattisesti laittaa noden ”To Text (String)”. ”To Text (String)”-node muuntaa merkkijonon Unreal Enginen ”Text”-muuttujaksi. [Kuva 22.]



Kuva 22. ”Calculate Rotation”-funktion palauttaman kulman tekstiksi muuntaminen kompassi-widgetissä.

Kuvassa 23 näkyy kompassi-widgetin ”Rotate”-event ja kuinka olen yhdistänyt kaikki nodet kokonaisuudessa toisiinsa [Kuva 23].

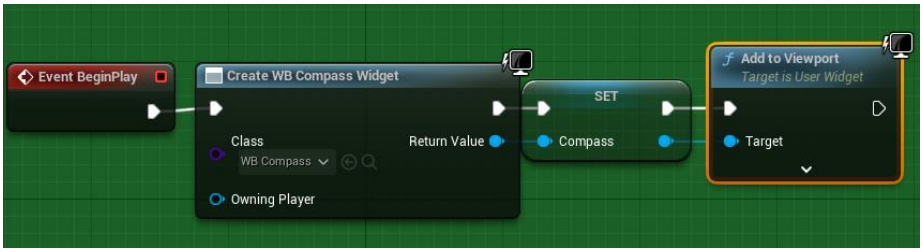


Kuva 23. Kompassi-widgetin koko ”Rotate”-Event

5.3.3 Kompassin testaaminen

Kun olin saanut kompassin widgetin siihen vaiheeseen, että se olisi valmis testattavaksi, loin uuden blueprint-luokan, jonka avulla pystyn testaamaan kompassia ei-diegeettisesti. Vaikka tulen tekemään diegeettistä kompassia, niin testaaminen ei-diegeettisesti on helpompaa ja nopeampaa. Kompassin toiminnallisuus ei-diegeettisenä ja diegeettisenä on lähes identtinen.

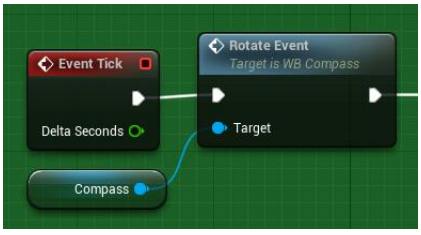
Widgettien tuominen ruudulle onnistuu Unreal Enginessä todella helposti. Olen kuvan 24 mukaan ottanut testaus-blueprint-luokan ”Event BeginPlay”-metodin ja liittänyt siihen ”Create Widget”-noden. ”Create Widget”-nodea käytetään widgettien luomiseen. Tässä tilanteessa käytin sitä kompassin widgetin luomiseen. ”Create Widget” palauttaa viitteen noden luomaan widgettiin ”Return Value”-ulostulosta, joten otin sen talteen ”Compass”-nimiseen muuttujaan. Lopuksi olen lisännyt ”Add to Viewport”-noden, joka on se node, joka tuo widgetin ruudulle. [Kuva 24.]



Kuva 24. HUD:in testaus-blueprint.

Kuvassa 25 näkyy, kuinka olen yhdistänyt ”Rotate Event”-noden ”Event Tick”-nodeen. ”Event Tick”-node mahdollistaa testauksen ennen, kuin ”Rotate Event” voidaan liittää suoraan mechan pyörähdykseen kiinni. [Kuva 25.]

”Event Tick”-node ei ole optimaalisin tapa liittää asioita, sillä ”Event Tick” ajetaan joka kerta kun pelimuodossa ruutuun päivitetään uudet pikselit. Suuremmissa luokissa tämä voi olla todella epäoptimaalinen tapa päivittää asioita, mutta tässä tilanteessa se toimii, koska sitä käytetään vain testauksessa. Lopullisessa projektissa ”Event Tick”-metodin käyttäminen on tässä tilanteessa suotavaa.



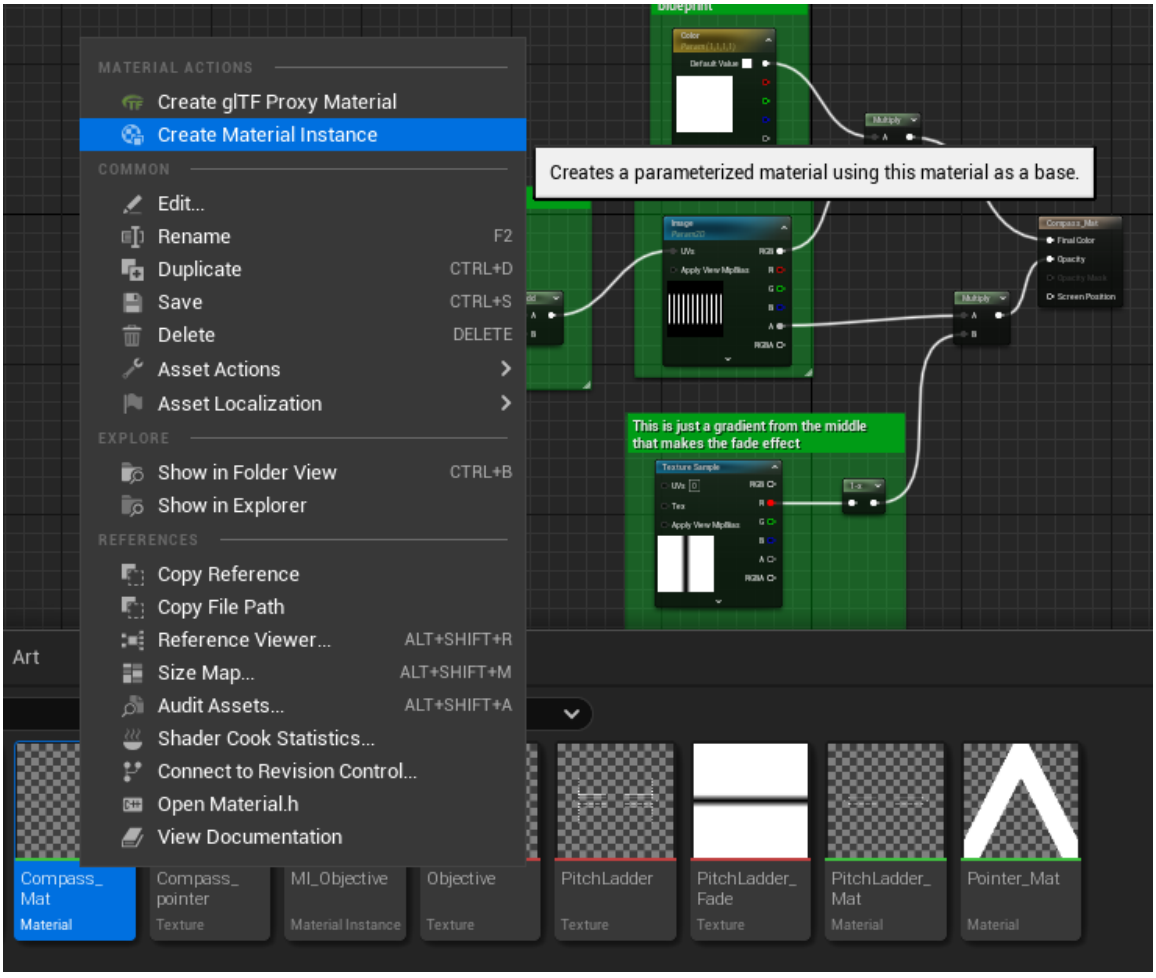
Kuva 25. HUD:in testaus-blueprint pyörimisen testausosio.

5.4 Kompassimerkki

Tässä kappaleessa käyn läpi, kuinka teen merkit kompassiin, mitä voidaan käyttää näyttämään pelaajalle erilaisten asioiden sijainteja kompassin avulla.

5.4.1 Merkkimateriaalin luominen

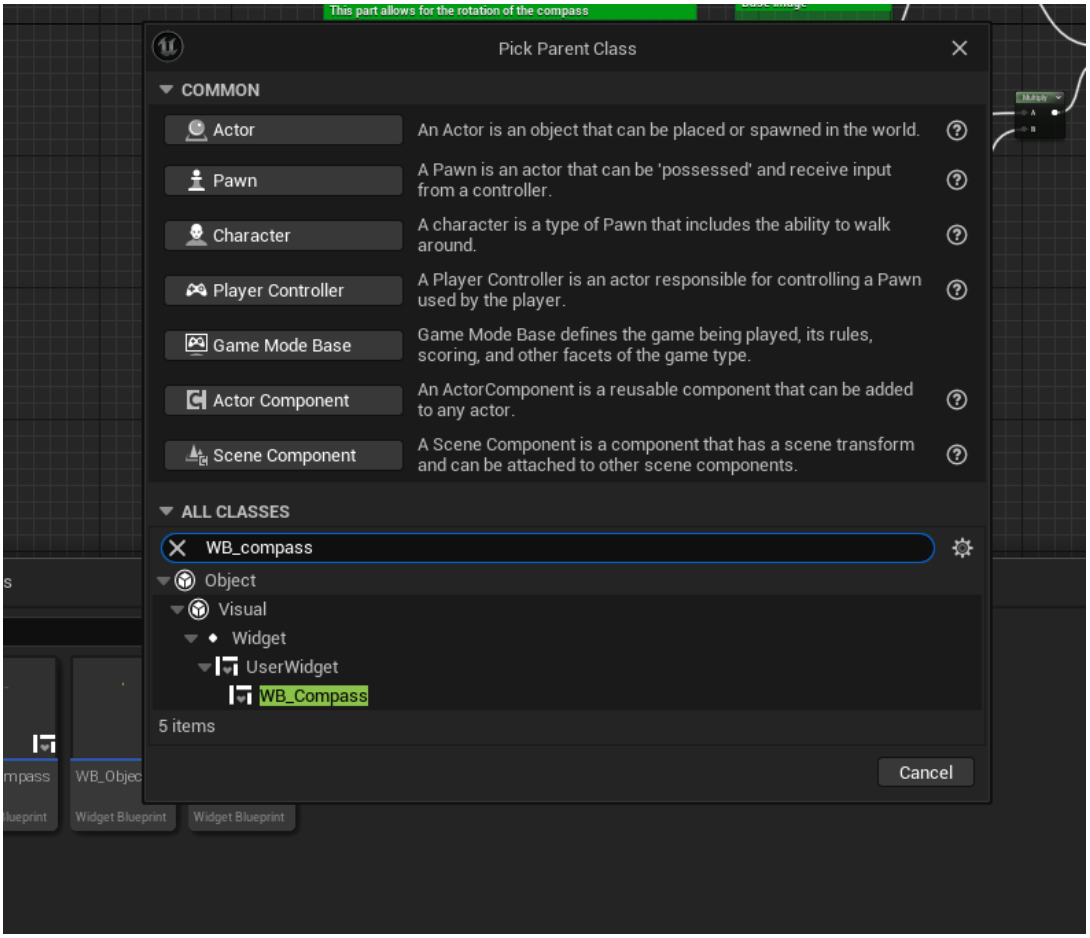
Ensimmäisenä kompassin merkkiä tehdessä tein kappaleessa 5.3.1 tekemästäni kompassin materiaalista ”Material Instance”:n. ”Material Instance”:n avulla minun ei tarvitse erikseen tehdä kompassin materiaalista toista identtistä materiaalia. ”Material Instance” mahdollistaa myös sen, että jos teen muutoksia kompassin materiaaliin samat muutokset tapahtuvat kyseisen materiaalin kaikilla ”Instanceille”. Nimesin luomani ”Material Instancen” nimellä ”MI_Objective”, mutta tulen tässä työssä kutsumaan sitä nimellä merkin materiaali. [Kuva 26.]



Kuva 26. ”Material Instancen”-luominen Unreal Engine 5 -pelimoottorissa.

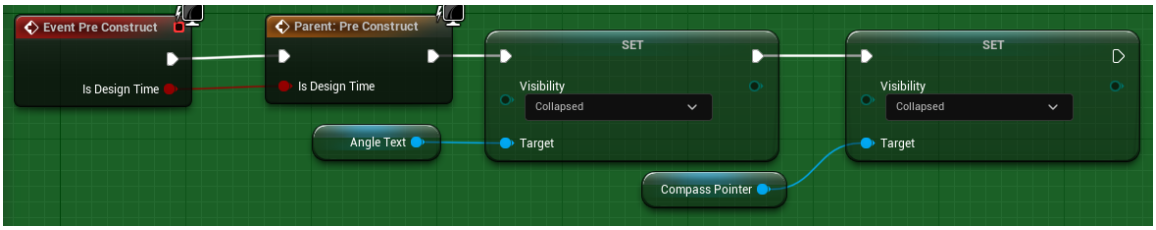
5.4.2 Merkki-widgetin luominen

Merkin materiaalin luomisen jälkeen loin kappaleessa 5.3.2 tekemästäni kompassin widgetistä uuden lapsi-widgetin. [Kuva 27] Luominen onnistuu lähes samalla tavalla kuin normaalin blueprint-luokan luominen, mutta luokka, joka halutaan periä, haetaan ”All Classes”-valikon alta. Tässä tapauksessa halusin tehdä lapsiluokan kompassi-widgetistäni niin etsin ”WB_Compass” joka on luomani kompassi-widget ja valitsin sen.



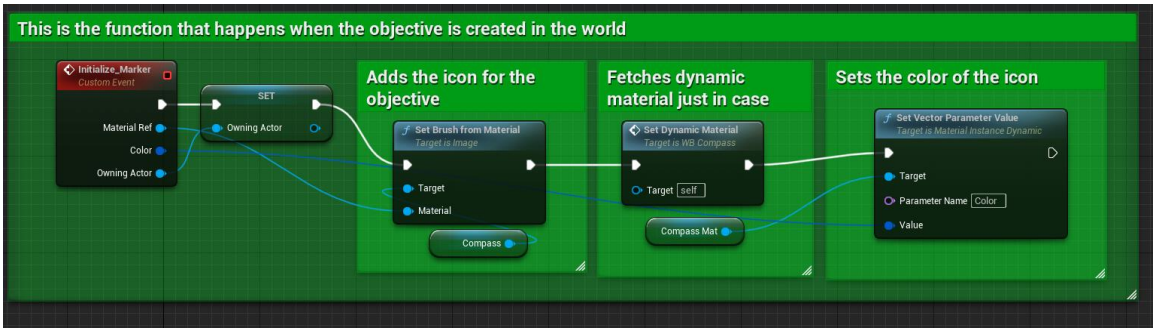
Kuva 27. Lapsiluokan luominen Unreal Engine 5 -pelimoottorissa.

Sillä luomani merkki-widget on lapsi kompassi-widgetistä, perii merkki-widget kaiken, mitä kompassi-widget sisältää. Tämän takia minun on piilotettava osa kompassin sisältämistä asioista, joita en merkeissä tule tarvitsemaan. Asiat, jotka minun on piilotettava merkki-widgetissä ovat kompassin sisältämä teksti, joka kertoo kompassin kulman ja nuoli, joka näkyy kompassin alapuolella. Tekstin ja nuolen piilottaminen on onneksi helppoa ja sain sen tehtyä käyttämällä kahta ”Set Visibility”-nodea. [Kuva 28.] Koska merkki widget on kompassi-widgetin lapsiluokka, pääse merkki-widget käsiksi ”Angle Text”- ja ”Compass Pointer”-komponentteihin ja nämä ovat juuri ne, jotka haluan piilottaa.



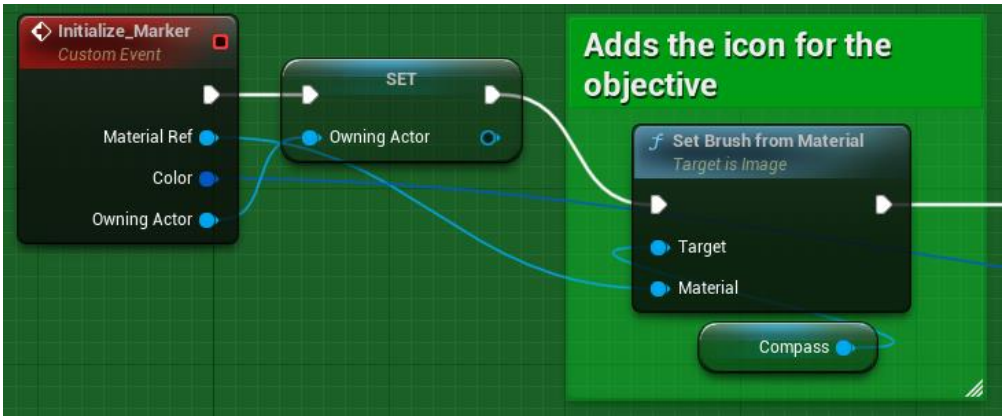
Kuva 28. Merkki-widgetin ”Event Pre Construct”-metodi

Kun olin saanut piilotettua ylimääräiset komponentit, loin merkki-widgettiin uuden eventin nimeltä ”Initialize_Marker”. Tämän eventin tarkoitus on mahdollistaa merkin kuvakkeen ja värin muokkaus silloin, kun merkki tullaan luomaan peliin. [Kuva 29.]



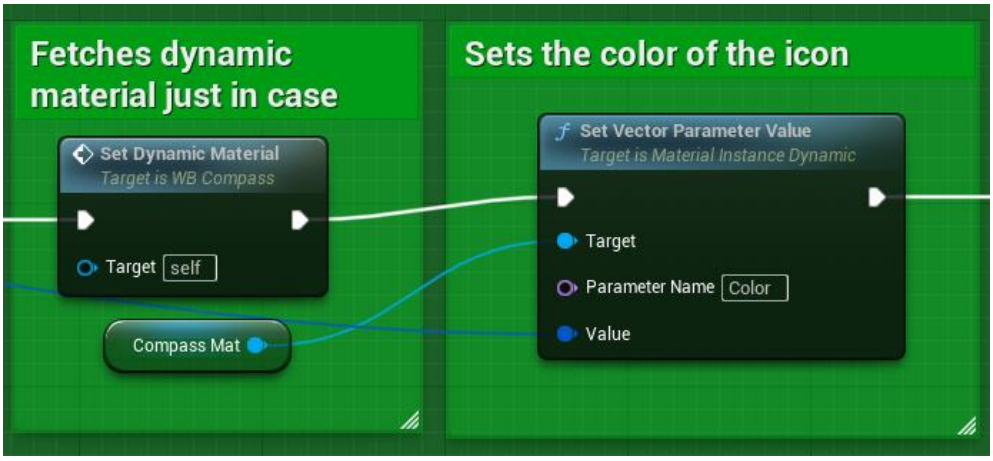
Kuva 29. Merkki-widgetin ”Initialize_Marker”-event

Luomani ”Initialize_Marker”-event sisältää sisääntuloina ”Material Ref”-, ”Color”- ja ”Owning Actor”-nimiset muuttujat. Näistä ”Material Ref” sisältää kuvakkeen, joka halutaan antaa merkille, ”Color” taas sisältää merkille tulevan värin ja ”Owing Actor” taas halutaan vain talteen muuttujaan, jotta voidaan päästään tarvittaessa käsiksi siihen objektiin, mikä on vastuussa kyseisestä merkistä. Otan kyseisen objektin talteen ”Owining Actor”-nimiseen muuttujan kyseisen muuttujan ”Set”-nodea käyttäen. Lisäksi mahdollistan merkin kuvan vaihtamisen käyttämällä ”Set Brush from Material”-nodea ja antamalla sille ”Material Refin” sisääntulon arvon. [Kuva 30.]



Kuva 30. Kuvakkeen muokkaaminen merkki-widgetissä.

Kuvasta 30 ”Set Brush from Material”-nodesta jatketaan kuvan 31 ”Set Dynamic Material”-nodeen. ”Set Dynamic Material”-noden avulla varmistan vain, että materiaali on asetettu ja se ei anna virheitä, jos se olisi vahingossa jäänyt asettamatta. Materiaalin tarkistuksen jälkeen käyn läpi ”Set Vector Parameter Value”-noden. ”Set Vector Parameter Value”-noden avulla vaihdan materiaalin ”Color”-parametria ja annan sille arvoksi ”Initialize_Marker”-eventistä tulleen ”Colorin” sisääntuloarvon. [Kuva 31.]



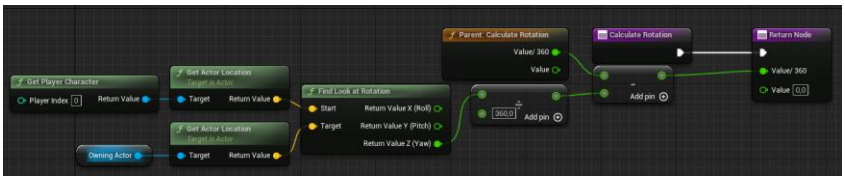
Kuva 31. Kuvakkeen värin muokkaaminen merkki-widgetissä

Kuvassa 32 olen tehnyt väliaikaisen lisän ”Initialize Marker”-eventtiin. Tämä lisä etsii tekemäni ”Hud Test”-blueprint-luokan ja lisää merkin ”Hud Test”-blueprintissä olevaan jonoon. Tämä jono on luotu, jotta voin testata merkkien toimivuutta helposti. [Kuva 32.]

Kuva 32. Merkin lisääminen ”Hud Test”-blueprintin jonoon.

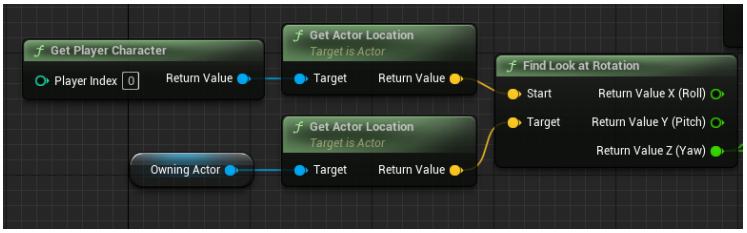
Seuraavaksi olen halunnut saada merkit pyörimään oikein, ja tämän olen saanut tehtyä käyttämällä ”Override”-funktiota kompassin-widgettiin kappaleessa 5.3.2 tekemästäni ”Calculate Rotation”-funktiosta. [Kuva 33.]

”Override”-funktiot ovat funktiontyyli, jossa jo valmiin funktion toiminnallisuus voidaan yliajaa eli muokata mitenkä funktio toimii.



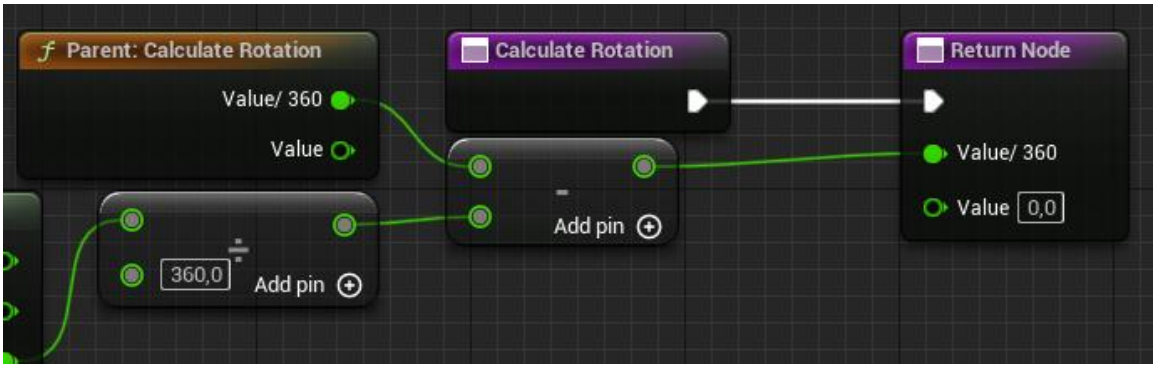
Kuva 33. Merkki-widget ”Calculate Rotaion”-Override funktio

Aluksi merkki-widgetin ”Calculate Rotation”-funktiossa haluan saada kulman merkin ja pelaajan välillä. Tämän olen saanut hakemalla sijainnin merkin omistavasta blueprintistä ja pelaajasta käyttämällä nodea ”Get Actor Location”. Tämä node hakee blueprintin sijainnin pelin maailmassa ja palauttaa sen arvon ”Return Value”-kohdasta. Käyttämällä pelaajan ja merkin sijainteja olen yhdistänyt nämä ”Find Look at Rotation”-nodeen. ”Find Look at Rotation”-node palauttaa kulman tässä tapauksessa pelaajan ja merkin omistavan blueprintin välillä. [Kuva 34.]



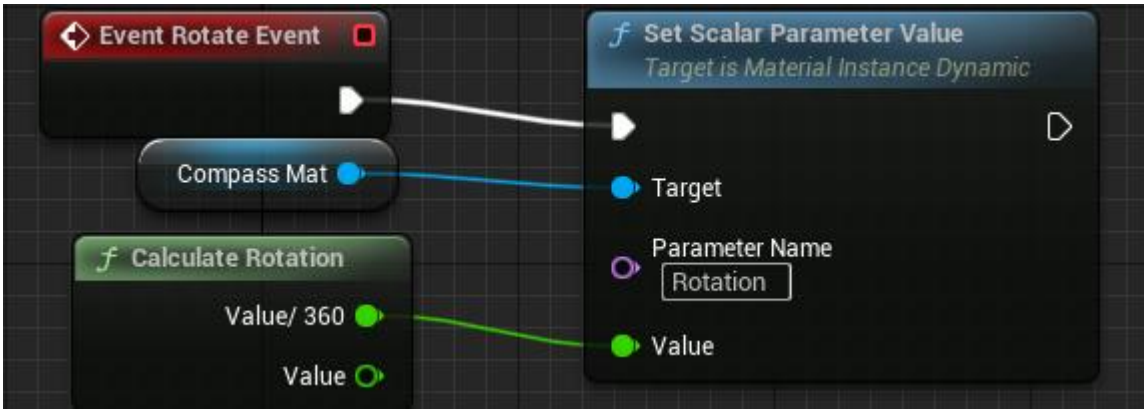
Kuva 34. ”Calculate Rotation Override”-funktion ensimmäinen osa.

Olen tämän jälkeen jakanut pelaajan ja merkin välisen kulman luvulla 360, jotta saan luvun lukujen 0–1 välille. Tämän jälkeen olen vähentänyt pelaajan ja merkin välisen kulman pelaajan pyörähdyskulmasta käyttäen ”Subtract”-nimistä nodea. Tämä vähentäminen siirtää merkkiä kompassissa siten, että merkki näkyy oikeassa paikassa kompassiin nähden. [Kuva 35.]



Kuva 35. ”Calculate Rotation Override”-funktion toinen osa.

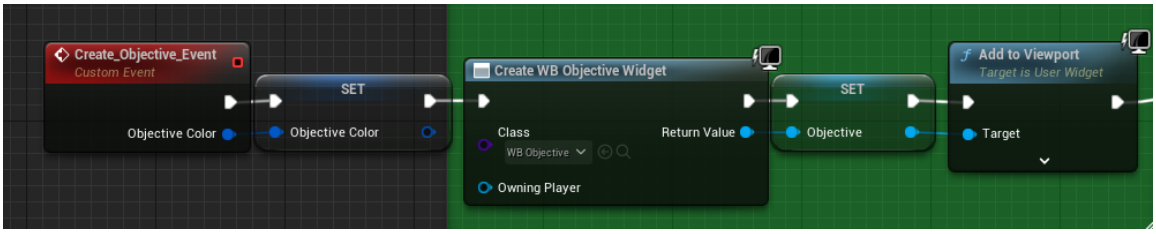
”Get Rotation Override”-funktion luomisen jälkeen olen luonut merkki-widgettiin ”Rotate”-nimisen eventin. [Kuva 36]. Olen tehnyt merkki-widgetin ”Rotate”-eventistä lähes identtisen kompassi-widgetin ”Rotate”-eventin kanssa [Kuva 19].



Kuva 36. merkki-widgetin ”Rotate”-event.

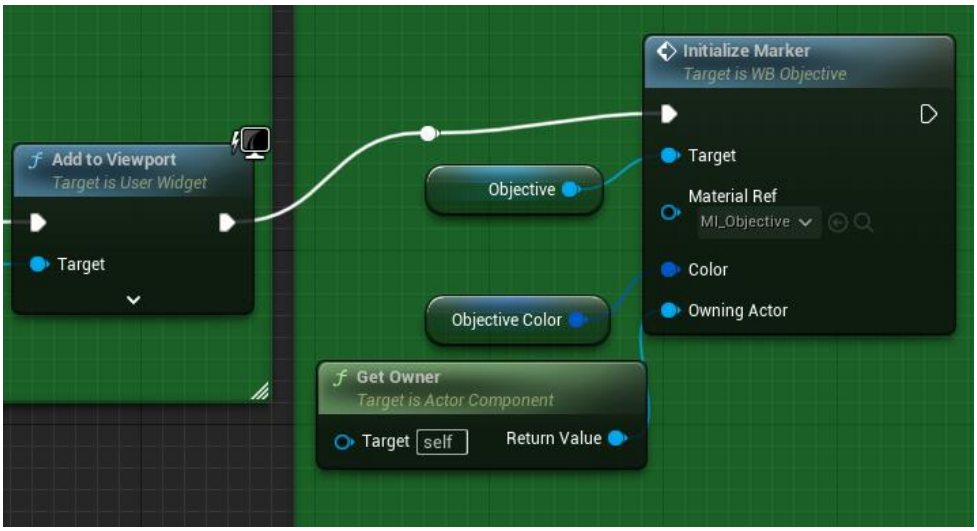
5.4.3 Merkki-blueprint-komponentin luominen

Kun olin saanut merkki-widgetin valmiiksi loin uuden ”Actor Component”-blueprint-luokan. Tätä luokkaa voi käyttää eri blueprint-luokissa, jotta saa merkin kyseiselle blueprintille näkymään kompassissa. Tähän merkkikomponentin ”Graph”-osiossa olen luonut uuden ”Create Objective”-eventin ja lisännyt siihen ”Objective Color”-sisääntulon ja ottanut sen talteen samannimiseen muuttujaan. Olen tämän jälkeen lisännyt ”Create Widget”- ja ”Add to Viewport”-nodet. [Kuva 37.] Nämä kaksi nodea mahdollistavat merkin testaamisen suoraan ruudulla kompassin tapaan. Olen myös ottanut luodusta widgetistä viittauksen talteen muuttujaan nimeltä ”Objective” käyttäen kyseisen muuttujan ”Set”-nodea.



Kuva 37 Merkkikomponentin ”Create Objective”-eventin ensimmäinen puolisko

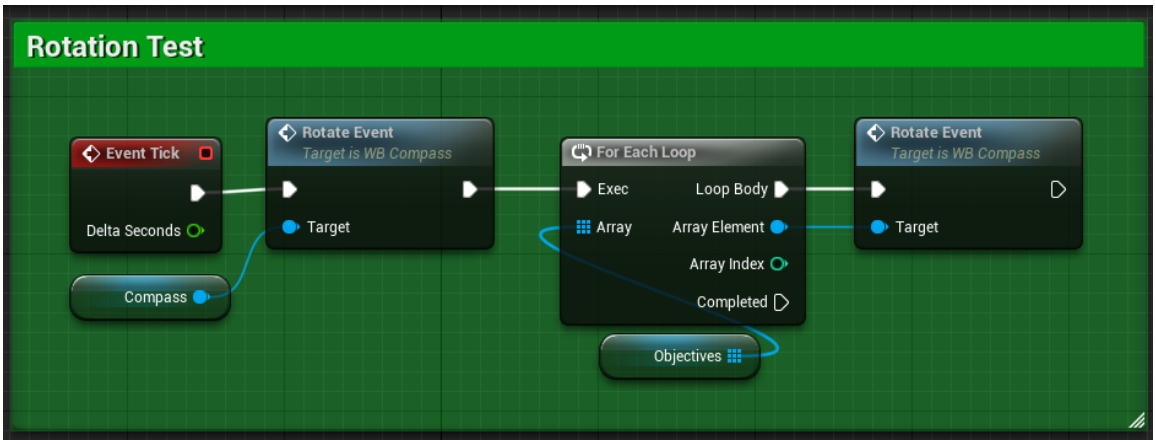
Merkki-komponentin ”Create Objective”-eventin lopuksi käytän kappaleessa 5.4.2 luomaani ”Initialize Marker”-eventtiä. Annan tässä merkille kuvaksi aiemmin tekemäni merkkimateriaalin ”MI_Objective” ja väriksi annan muuttujan nimeltä ”Objective Color” tein tämän siksi, että voin vapaasti valita merkin värin. Lopuksi annoin ”Owning Actorin” kohtaan ”Get Owner”-noden palauttaman arvon. [Kuva 38.]



Kuva 38. Merkkikomponentin ”Create Objective”-eventin toinen puolisko

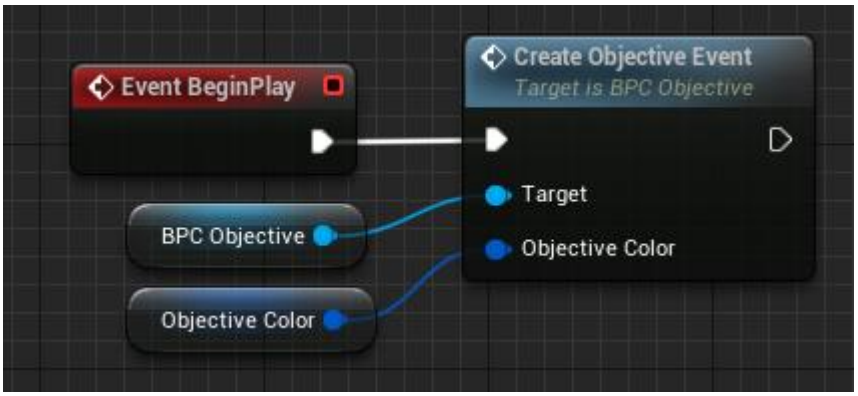
5.4.4 Merkin testaaminen

Olen lisännyt ”HUD Test”-blueprint-luokan ”Event Tick”-osioon ”For Each Loop”-noden. ”For Each Loop”-noden tarkoitus on tässä tilanteessa käydä kaikki ”Objectives”-jonossa olevat merkit läpi ja ajaa jokaisen niiden ”Rotate”-event. [Kuva 39.]



Kuva 39. HUD:in testaus-blueprintin ”Event Tick”-metodi

Olen myös luonut uuden blueprint-luokan, johon olen liittänyt kappaleessa 5.4.3 tekemäni merkkikomponentin. Olen tästä komponentista hakenut tekemäni ”Create Objective”-eventin ja liittänyt siihen ”Objective Color”-nimisen värimuuttujan, jonka avulla voin valita jokaiselle merkille oman värin kompassissa. [Kuva 40.]



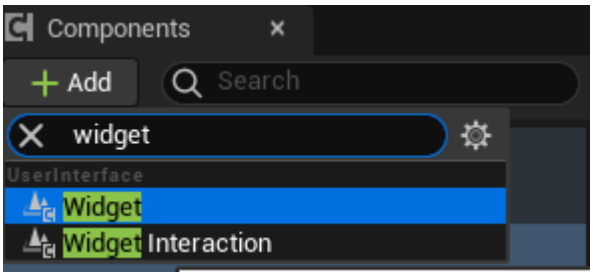
Kuva 40 Merkki-blueprint-luokan ”Event BeginPlay”-funktio.

5.5 Kompassi ja merkki diegeettiseksi

Tässä kappaleessa tulen kertomaan siitä, kuinka olen tehnyt kappaleissa 5.3 ja 5.4 tekemistäni kompassista ja kompassin merkistä widgeteistä yhden blueprintin, jota voidaan käyttää diegeettisenä kompassina mechan ohjaamossa.

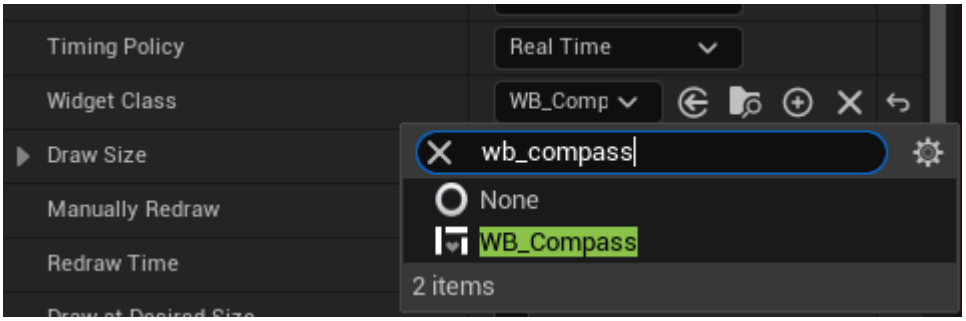
5.5.1 Blueprint-luokan luominen

Aluksi loin uuden blueprint-luokan kompassille. Tähän blueprint-luokkaan lisäsin komponentin nimeltä ”Widget” [Kuva 41]. Tätä ”Widget”-komponenttia käyttämällä voin tuoda luomani kompassi-widgetin tähän kyseiseen blueprint-luokkaan.



Kuva 41. ”Widget”-komponentin luominen.

Jotta saan widget-komponentin näyttämään kappaleessa 5.3 tekemäni kompassin widgetin, olen käynyt widget-komponentin ”Details”-osiosta valitsemassa ”Widget Class”-valikosta luomani kompassin widget-luokan [Kuva 42].



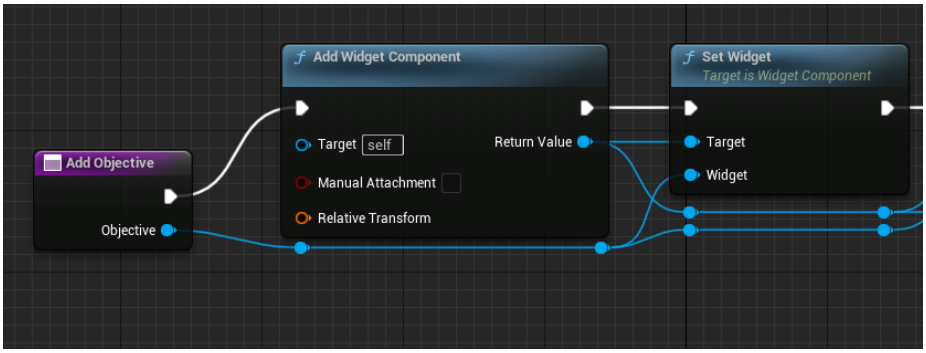
Kuva 42. Widget-komponentin ”Widget Class” antaminen.

5.5.2 Merkkien lisääminen

Merkkien lisäämiseksi kompassin blueprinttiin minun on luotava funktio, jota merkit voivat kutsua lisätäkseen blueprinttiin uuden widget-komponentin ja antamaan kyseiselle komponentille viittauksen itseensä.

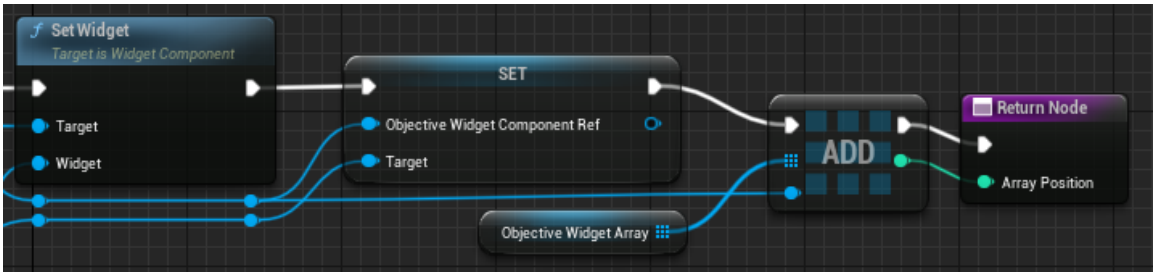
Kuvassa 43 näkyy, kuinka olen luonut uuden funktion nimeltä ”Add Objective”. Olen antanut funktiolle yhden sisääntulon, joka ottaa sisään vain merkki-widget luokka-arvoja. [Kuva 43.]

Ensimmäisenä funktioon olen laittanut ”Add Widget Component”-noden. Tämä node luo blueprint-luokkaan uuden widget-komponentin. ”Add Widget Component”-noden olen liittänyt ”Set Widget”-nodeen. ”Set Widget”-node laittaa juuri luomaamme widget-komponenttiin widget-viittauksen, jotta se osaa käyttäytyä oikein. Widget-viittauksena toimii merkki-widgetin sisääntulo. [Kuva 43.]



Kuva 43. ”Add Objective”-funktion ensimmäinen osa.

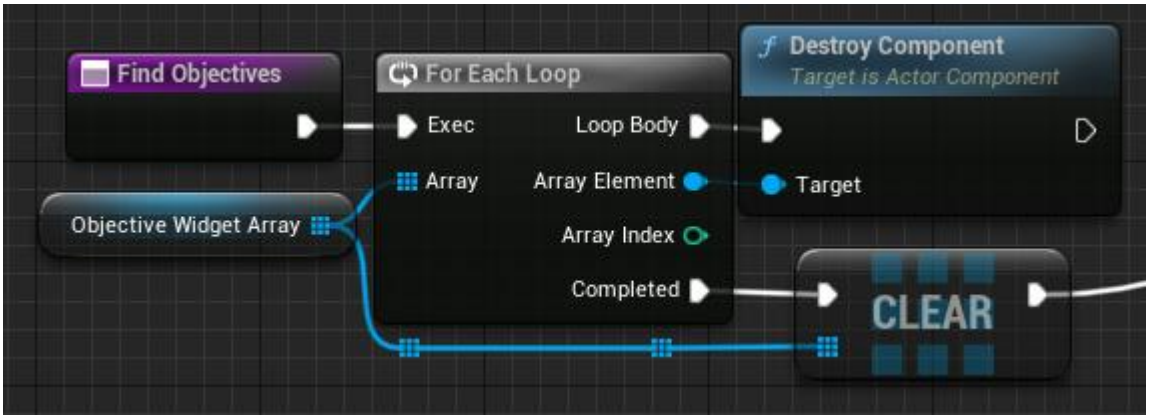
”Set Widget” noden jälkeen halusin ottaa merkki-widgettiin talteen referenssin widget-komponenttiin. Tämän jälkeen lisäsin widget-komponentin listaan nimeltä ”Objective Widget Array”, josta palautan komponentin sijainnin jonossa tarvittavaa testausta varten. [Kuva 44.]



Kuva 44 ”Add Objective” funktion toinen osa.

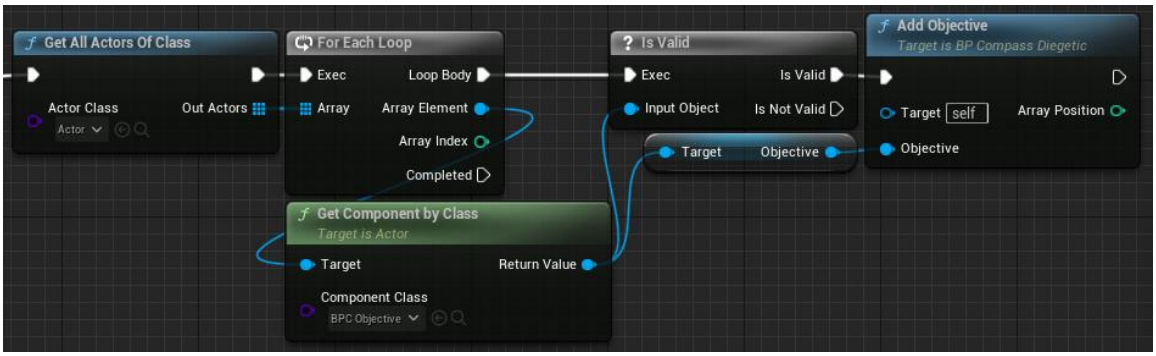
Loin kompassi-blueprinttiin myös ”Find Objectives”-nimisen funktion. Tämän funktion tarkoitus on löytää kentästä kaikki mahdolliset merkit ja lisätä ne kompassiin [Kuva 45].

Ensimmäisenä ”Find Objectives”-funktiossa olen ottanut minun ”Objective Widget Array”-muuttujajonon. Tämä jono sisältää viittauksen kaikkiin näytettävien merkkien komponentteihin. Olen ottanut ”Objective Widget Array”-muuttujajonosta ”For Each Loop”-noden ja poistan sen avulla jokaisen widget-komponentin, mitä mahdollisesti ”Objective Widget Array” sisältää. Kun kaikki mahdolliset komponentit on tuhottu, niin lopuksi vielä tyhjennän listan, jos sinne on vahingossa jäänyt tyhjiä viittauksia. [Kuva 45.]



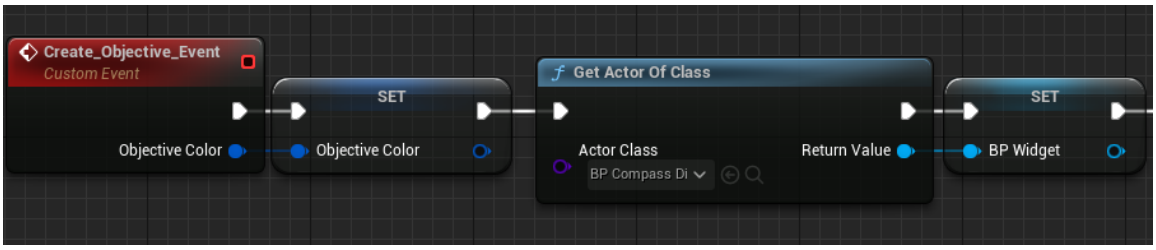
Kuva 45. ”Find Objectives”-funktion ensimmäinen osa

Kun olen poistanut kaikki widget-komponentit mitä mahdollisesti kompassissa on ollut ja tyhjentänyt ”Objective Widget Array”-listan, haen kentästä kaikki merkit käyttäen ”Get All Actors Of Class”-nodea. ”Get All Actors Of Class”-node tässä palauttaa kaikki ”Actor”-luokat, mitä se löytää kentästä. Tämän jälkeen käyn kaikki ”Actor”-luokat läpi ”For Each Loop”-noden avulla ja testaan, sisältävätkö ne tekemääni merkkikomponenttia. Jos komponentti löytyy, pääsee se ”Is Valid”-noden läpi ja asettaa uuden merkin kompassiin. [Kuva 46.]



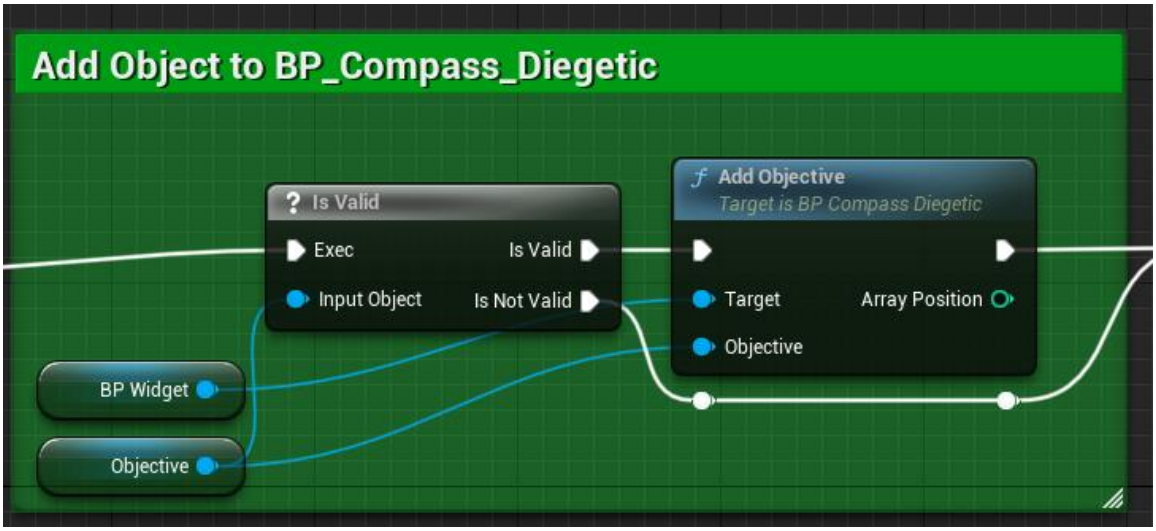
Kuva 46. ”Find Objectives”-funktion toinen osa.

Jotta saan merkit näkymään diegeettisessä kompassissa, on minun lisättävä merkit kompassiin käyttämällä juuri luomaani funktiota. Jotta pystyn kutsumaan funktiota, on minun löydettävä viittaus kompassi-blueprinttiin. Tämä onnistuu käyttäen ”Get Actor Of Class”-nodea. ”Get Actor Of Class”-node etsii kyseisestä kentästä kyseisen luokan objektin ja palauttaa ensimmäisen, minkä se löytää. Tässä tilanteessa, missä kompasseja tulee olemaan vain yksi, tämän noden käyttäminen on ihan ok, mutta jos kompasseja olisi useampia ei tämä node toimisi. Otan vielä ”Get Actor Of Class”-nodesta sen palauttaman arvon talteen ”BP Widget”-nimiseen muuttujaan. [Kuva 47.]



Kuva 47. Merkki komponentin ”Create Objective”-eventin alkupään muutoksia

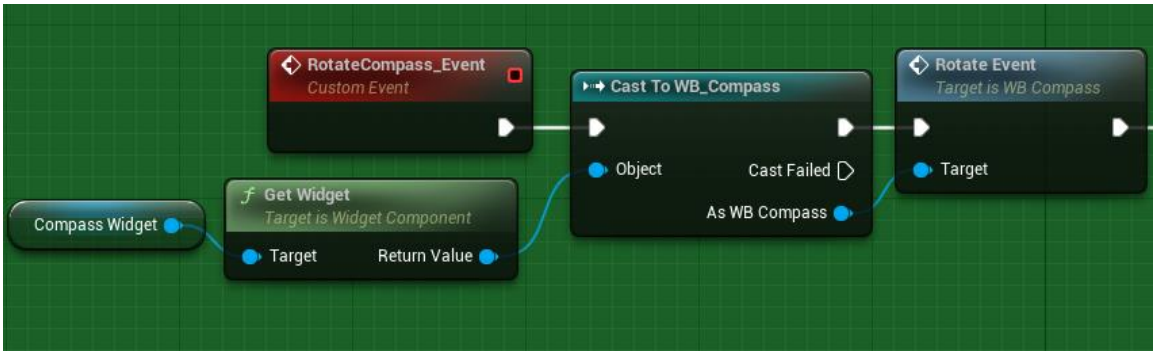
Merkkikomponentin ”Create Objective” loppupuolella ennen kuin ajan ”Initialize Marker”-eventtiä testaan aluksi, onko aiemmin tekemäni haku kompassi-blueprintille onnistunut käyttäen ”Is Valid”-nodea. ”Is Valid”-node tarkistaa muuttujan ja katsoo, sisältääkö se sopivan arvon. Tässä tilanteessa, jos sopiva arvo löytyy, ajetaan aiemmin tekemäni ”Add Objective”-funktio ja jos taas sopivaa arvoa ei löydy ohitetaan ”Add Objective”-funktio kokonaan. [Kuva 48.]



Kuva 48. Merkkikomponentin ”Create Objective”-eventin loppupään muutoksia.

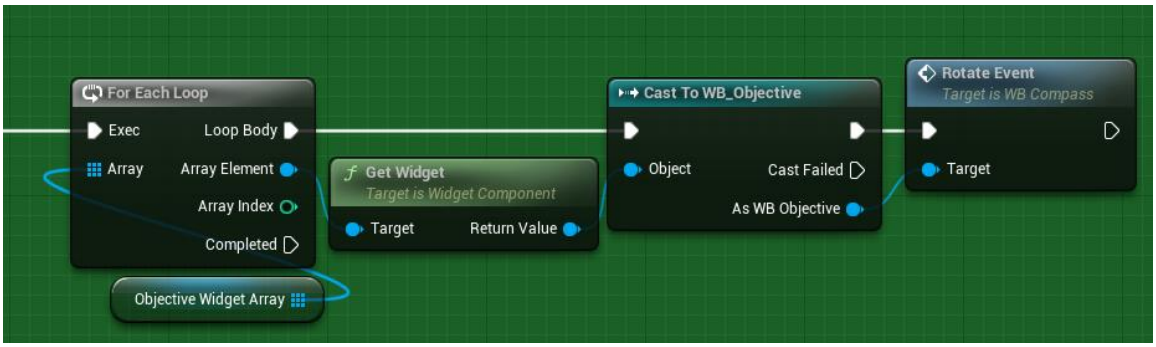
5.5.3 Kompassin ja merkkien pyöriminen

Jotta saan kompassin pyörimään, olen luonut uuden eventin kompassin blueprintin sisälle ja kutsun sitä nimellä ”RotateCompass”. Tähän eventtiin olen hakenut kompassi-widget-komponentin, josta olen hakenut itse kompassi-widgetin käyttämällä nodea ”Get Widget”. ”Get Widget”-node toimii päinvastoin kun käyttämäni ”Set Widget”-node ja palauttaa widget komponentissa olevan widgetin. Kun minulla oli viittaus kompassin widgettiin, pystyin ”Cast”-noden avulla kutsumaan ”Rotate”-eventin. [Kuva 49.]



Kuva 49. Kompassi-blueprintin ”RotateCompass”-eventin kompassin pyöräytysosa.

Jotta saan kaikki merkit pyörimään kompassin kanssa, olen ottanut aiemmin luomani ”Objective Widget Array”-jonon, joka sisältää kaikki kompassissa olevat merkit ja olen liittänyt sen ”For Each Loop”-nodeen. ”For Each Loop”-nodessa haen widgetin komponentista ja ”Cast”-noden avulla haen ”Rotate”-eventin samalla tavalla kuin kompassissa. [Kuva 50.]



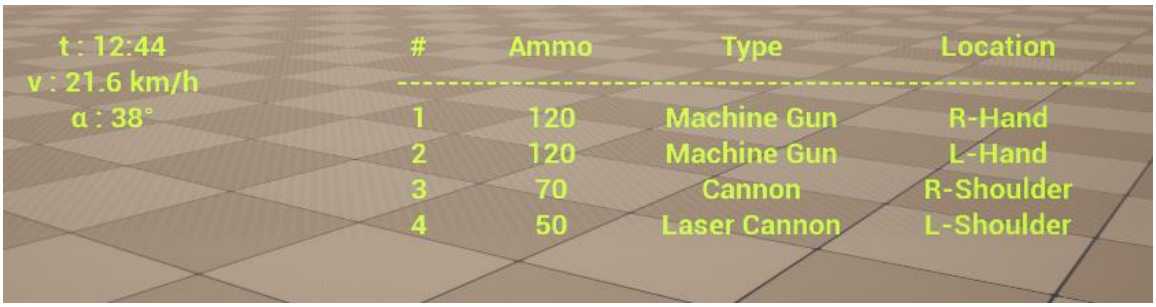
Kuva 50. Kompassi-blueprintin ”RotateCompass”-merkkien pyöräytysosa.

5.6 Ase - & Info-widgettien tekeminen

Olen tehnyt suunnitelmani mukaan [Kuva 10] diegeettiset UI-widgetit aseista ja infosta. [Kuva 51] Tein ensiksi info-widgetin, joka antaa pelaajalle nimensä mukaan vähän pientä infoa. Ylimpänä lisäinfo osiosta on aika eli t. Haen ajan käyttämällä Unreal Enginen ”Now”-nimistä nodea, joka hakee käyttäjän systeemin ajan ja antaa ne ulos. ”Now”-nodesta olen ottanut vain tunnin ja minuutin ja lisäsin siihen oman pienen funktion, joka lisää nollan luvun eteen, jos se on alle 10. Toiseksi olen hakenut pelaajan nopeuden. Pelaajan nopeus on haettu käyttäen ”Get Actor Velocity”-nodea ja siitä ottaen ”Get Vector Length”-nodella vektorin pituuden. Vektorin pituus antaa tässä vaiheessa antaa nopeuden metreinä sekunnissa, mutta sen saa muutettua kilometriä tunnissa kertomalla luku 0.036. Ja lopuksi olen tehnyt arvon, joka vertaa kahden ”Actor”-blueprintin pyörähdystä ja antaa niiden eron arvona. Tätä arvoa olisi tarkoitus käyttää kertomaan pelaajalle mechan ala- ja yläosan pyörähdyksen ero.

Info-widgetin jälkeen loin ase-widgetin. Ase-widget nimensä mukaan kertoo pelaajalle tietoa hänen mechansa käyttämistä aseista, esim. paljonko ammuksia aseessa on, mikä ase on kyseessä ja missä ase sijaitsee. Tein ase-widgetin siten, että siihen voidaan luoda uusia sarakkeita riippuen siitä, montako asetta on käytössä. Esim. jos mechalla on käytössään vain 2 asetta niin sarakkeita näytetään vain 2 kappaletta. Lisäksi ase-widget sisältää funktion, jota kutsumalla voi muokata pelkästään sarakkeen ammuslukua.

Kun olin saanut molemmat widgetit valmiiksi, tein niistä diegeettiset blueprintit aivan kuten tein aiemmin kompassista kappaleessa 5.5.



Kuva 51. Diegeettiset ammus- ja lisäinfo-blueprint-luokat.

6 Yhteenveto

Tässä työssä oli tavoitteena suunnitella ja luoda diegeettinen HUD käyttäen Unreal Engine 5 -pelimoottoria. Tämän HUD:in luominen onnistui loppujen lopuksi helposti käyttämällä widget-blueprint-luokkia ja sitten tuomalla ne pelin maailmaan käyttämällä blueprint-luokkien widget-nimistä komponenttia.

HUD:in osista käytiin kompassin ja siihen kuuluvien merkkien tekeminen hyvin vaihe vaiheelta. Aluksi kerrottiin, kuinka kompassin materiaali luotiin. Tämän jälkeen käytiin läpi itse kompassin widgetin tekeminen ja lopuksi tuotiin widget peliin tekemällä sille oma blueprint-luokka. Kaikki nämä sitten toistettiin merkille, sillä se käytännössä toimii omanlaisena kompassina. Kaikki tämä tehtiin, jotta tulisi hyvin selväksi, mitä on missäkin vaiheessa tehty ja jotta tätä työtä voidaan käyttää jonkinlaisena ohjeena tulevaisuudessa.

Kompassin lisäksi työssä käytiin läpi kaksi muutakin HUD:in palasta, Ammus- ja info-osiot. Näistä osioista kirjoitettiin hieman tiiviimmin, kun perusidea oli näissäkin sama kuin kompassissa. Näistä kahdesta widgetistä kerrottiin tiiviisti, miten mikäkin osio on tehty, mutta ei ole ihan niin selkeitä ohjeita, kun kompassin tekovaiheessa, joten näiden kahden tekeminen tätä ohjeena käyttäen ei ole ihan niin helppoa, mutta mahdollista sen pitäisi olla.

Tässä vaiheessa tätä työtä voidaan jo kokeilla itse projektissa, mutta ei se ole julkaisuvalmiissa kunnossa. Julkaisuvalmiiseen versioon pitäisi päivittää tekstuurit, jotka on työhön tehty ja mahdollisesti vaihtaa fontti projektiin sopivammaksi.

Lähteet

1. Game UI Design: Everything you need to know. Careerfoundy. [Internet]. [viitattu 19.5.2024]. Saatavilla: https://careerfoundry.com/en/blog/ui-design/game-ui-design/
2. Slightly Mad Studios. Need For Speed Shift. [Videopeli]. 2009. Electronic Arts.
3. Need for Speed: Shift. Steam [Internet]. [viitattu 19.5.2024]. Saatavilla: https://store.steampowered.com/app/24870/Need_for_Speed_Shift/
4. Head-Up Display. Scholarly Community Encyclopedia. [Internet]. [Viitattu 8.6.2024]. Saatavilla: https://encyclopedia.pub/entry/32105
5. Capcom. Devil May Cry 5. [Videopeli]. 2019. Capcom.
6. Game UI Database. Game UI Database. [Internet]. [viitattu 19.5.2024]. Saatavilla: https://www.gameuidatabase.com/
7. Treyarch. Call of Duty Black Ops III. [Videopeli]. 2015. Activision.
8. CD Projekt RED. Cyberpunk 2077. [Videopeli]. 2020. CD Projekt RED.
9. A Beginner's guide to mecha. New York Public Library. [Internet]. [viitattu 19.5.2024]. Saatavilla: https://www.nypl.org/blog/2019/04/04/beginners-guide-mecha-manga-anime
10. Osamu Tezuka. Tetsuwan Atom (Astro Boy). [Manga]. 1952.
11. Mitsuteru Yokoyama. Tetsujin 28-go (Gigantor). [Manga]. 1956.
12. Hasbro & Takara Tomy. Transformers. [Media]. 1984.
13. Gainax. Neon Genesis Evangelion. [Anime]. 1995.
14. Respawn Entertainment. Titanfall. [Videopeli]. 2014. Electronic Arts.
15. Piranha Games Inc. MechWarrior 5: Mercenaries. [Videopeli]. 2021. Piranha Games Inc.
16. FromSoftware. Armored Core VI: Fires of Rubicon. [Videopeli]. 2023. Bandai Namco.

17. Unreal Engine: The most powerful real-time 3D creation tool. Epic Games. [Internet]. [viitattu 14.7.2024]. Saatavilla: https://www.unrealengine.com/en-US

18. Comparing blueprints and c++ use cases. Epic Games. [Internet]. [viitattu 19.5.2024]. Saatavilla: https://dev.epicgames.com/community/learning/tutorials/qM2K/unreal-engine-comparing-blueprints-and-c-use-cases

19. Balancing Blueprint and C++. Epic Games. [Internet]. [Viitattu 31.7.2024]. Saatavilla: https://dev.epicgames.com/documentation/en-us/unreal-engine/balancing-blueprint-and-cplusplus?application_version=4.27

20. Widget Component. Epic Games. [Internet]. [viitattu 17.7.2024]. Saatavilla: https://dev.epicgames.com/documentation/en-us/unreal-engine/widget-components-in-unreal-engine

21. Materials. Epic Games. [Internet]. [Viitattu 18.7.2024]. Saatavilla: https://dev.epicgames.com/documentation/en-us/unreal-engine/unreal-engine-materials

22. Creating and Using Material Instances. Epic Games. [Internet]. [Viitattu 18.7.2024]. Saatavilla https://dev.epicgames.com/documentation/en-us/unreal-engine/creating-and-using-material-instances-in-unreal-engine

23. F/A-18C. [Internet]. [viitattu 19.5.2024]. Saatavilla: https://wiki.hoggit-world.com/view/F/A-18C

24. David Capello. Aseprite. [Sovellus]. 2016. Igara Studio