Miro Salminen

C++:n ja Blueprintin yhteiskäyttö Unreal-pelimoottorissa

Tradenomi
Tietojenkäsittely
Syksy 2019

Tiivistelmä

Tekijä: Miro Salminen

Työn nimi: C++:n ja Blueprintin yhteiskäyttö Unreal-pelimoottorissa

Tutkintonimike: tradenomi, tietojenkäsittely

Asiasanat: Unreal, Blueprint, C++, yhteiskäyttö

Opinnäytetyön tavoitteena oli selkeyttää, miten C++-koodi ja Blueprintit toimivat yhdessä Unreal-pelimoottorissa ja miten valinta erilaisissa tilanteissa näiden välillä tehdään. Yhteiskäytön avulla voidaan saavuttaa sujuva rinnakkain tekeminen, jotta jokainen tiimin jäsen voisi keskittyä omaan osaamisalueeseensa.

Unrealin koodikielien yhteiskäytöstä on vain vähän selkeästi koottua tietoa. Opinnäytetyön tarkoituksena oli selvittää, miten saadaan koko tiimin työpanos pelin logiikoiden tekoon vaikkei tiimissä olisikaan montaa koodaustaitoista. Jos tiimi olisikin täynnä koodaustaitoisia, onko silti järkevää tehdä kaikki pelkkään C++-koodiin?

Blueprintien suoritusteho on heikompi C++-koodiin verrattuna, mutta suorituskyky ei ole kaikki kaikessa. Blueprintien muokattavuuden helppous on monikertaisesti parempi C++-koodiin verrattuna. Tämä mahdollistaa kaikkien tiimin jäsenten osallistumisen projektin tekoon. Kummalla koodikielellä projekti tehdään, ei ole merkitystä, kunhan se soveltuu tekijöiden osaamiseen.

Opinnäytetyössä tuotiin esille kaksi vaihtoehtoista työtapaa yhdistää C++-koodi ja Blueprintit. Esimerkkityön avulla havainnollistettiin, miten voidaan yhdistää C++-koodin suorituskyky ja Blueprintien muokattavuus. Ensimmäisessä versiossa painotettiin enemmän C++:aa ja toisessa Blueprintejä. Molemmissa esimerkkiversioissa käytettiin hyväksi C++:n perinnöllisyyttä. Lopuksi näitä työtapoja pohdittiin tehokkuuden, suorituskyvyn ja työryhmän näkökulmasta.

Prosessin aikana todettiin C++-painotteisen version olevan ihannetilanne Unreal-pelimoottorin käyttämisestä. Ohjelmoijat saavat keskittyä luomaan koodia, kun taas artisti tai suunnittelija voivat keskittyä visuaaliseen tekemiseen. C++-painotteinen versio yhdistää erinomaisesti C++:n suorituskyvyn ja Blueprintien muokattavuuden. C++:n ja Blueprintien tehokas yhteiskäyttö vaatii laajan tietämyksen Unreal-pelimoottorista ja sen toiminnasta.

Tarkoituksena oli selvittää, miten käyttää C++-koodia ja Blueprinttejä yhdessä. Selkeää vastausta tähän kysymykseen ei ole vaan kysymykseen pitää aina vastata tiimin sisällä, sillä jokainen projekti ja tiimi on erilainen. Työn tuloksena voidaan todeta, että projektia ei kannata tehdä pelkästään joko C++-koodilla tai Blueprinteillä. Jos projektia ei saada valmiiksi osaamattomuuden vuoksi tai jos puolet tiimistä odottaa asian lisäämistä C++-koodiin, silloin työtavan valinta on ollut väärä suhteessa tiimin kokoonpanoon.

Abstract

Author: Miro Salminen

Title of the Publication: Collaboration between C++ and Blueprint in Unreal Game Engine

Degree Title: Bachelor's Degree in Business Information Technology

Keywords: Unreal, Blueprint, C++, Collaboration

The aim of this thesis is to clarify how C++ code and Blueprints work together in Unreal Engine and how to choose between them in different situations. Collaboration aims to seamless cooperation so that each team member can focus on their own area of expertise.

There is little clear information about collaboration of coding languages in Unreal Engine. A further goal of this thesis was to find out how to get the whole team to contribute to games logics, even if the team doesn't have many members with coding skills. If the team is full of members with coding skills, is it still relevant to do everything in C++ alone?

Blueprints have lower performance than C++, but performance is not everything. Blueprints are many times easier than C++ code, and it allows all team members to participate in making the project. Which code language the project should be using is an irrelevant question, because coding language should be chosen based on the skills of the makers.

The thesis presented two alternative ways of combining C++ code and Blueprints. The case study illustrated how to combine the performance of C++ code with the configurability of Blueprints. The first version focused more on C++ and the second version focused on Blueprint. Both versions utilized inheritance of C++. These working methods were compared from the point of view of efficiency, performance and team.

The statement of this thesis was that the C++ based version is the ideal situation for using the Unreal Engine. Programmers can focus on creating code, while artists and designers can focus on visual creation. The C++ version combines C++ performance and Blueprint customizability perfectly. Efficient collaboration between C++ and Blueprints requires extensive knowledge of the Unreal Engine.

The purpose of this thesis was to find out how to use C++ code and Blueprints together. There is no clear answer to this question, the question should always be answered within the team, because each project and team is different. It is not worth doing the project with either C++ code or Blueprint alone. If the project is not completed because of incompetence, or if half of the team is waiting for something to be added to the C++ code, then the working method has been wrong for the team composition.

Sisällys

SYMBOLILUETTELO

Frame	Näytölle renderöity kuva, joka lasketaan useasti sekunnissa.
Funktio	Itsenäinen ohjelman osa-alue, joka suorittaa tietyn toiminnon.
Logiikka	Peliin ohjelmoitava toiminnallisuus.
Makro	Automaattinen syöttösarja, joka matkii näppäimistön tai hiiren toimintoja.
Olio	Luokasta luotu ilmentymä. Tietorakenne, johon kuuluvat sitä käsittelevät toiminnot.
Peliolio	Pelimaailmaan asetettava olio.

1 Johdanto

Opinnäytetyön tavoitteena on selkeyttää, miten C++-koodi ja Blueprintit toimivat yhdessä Unreal-pelimoottorissa ja miten valinta erilaisissa tilanteissa näiden välillä tehdään. Blueprintit ovat Unreal-pelimoottorin oma visuaalinen solmupohjainen koodikieli, jota voidaan käyttää yhdessä perinteisen C++-koodin kanssa. Yhteiskäytön avulla halutaan mahdollistaa sujuva rinnakkain tekeminen, jotta jokainen tiimin jäsen voi keskittyä omaan osaamisalueeseensa.

Unrealin koodikielien yhteiskäytöstä on vain vähän selkeästi koottua tietoa. Unreal-pelimoottori on vielä melko uusi kuluttajien käytössä, sillä se julkaistiin ilmaisversiona vasta vuonna 2015. Samanlaista kommuunin luomaa tietomäärää pelimoottorin käytöstä ei ole vielä ehtinyt syntyä kuin esimerkiksi täysin ilmaisen Unity-pelimoottorin kanssa. C++-koodi on myös haastava kielenä, minkä vuoksi suurin osa dokumentaatiosta on vain Blueprinteista. Opinnäytetyön tarkoituksena on selvittää, miten saadaan koko tiimin työpanos pelin logiikoiden tekoon, vaikkei tiimissä olisikaan montaa koodaustaitoista. Jos tiimi olisikin täynnä koodaustaitoisia, onko silti järkevää tehdä kaikki pelkkään C++-koodiin?

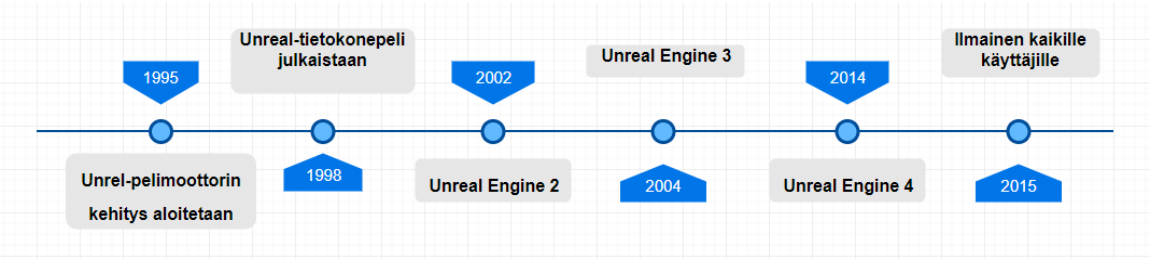
Opinnäytetyön alkuosassa tutkitaan C++-koodin ja Blueprintien eroja ja miten näitä koodaustapoja voidaan yhdistää yhtenäiseksi toimintamalliksi. Toimintatapojen yhdistämisen edellytyksenä on käyttää erilaisia makroja ja näiden lisämääritteitä, joista tärkeimmät esitellään tässä opinnäytetyössä. Tämän pohjalta laaditaan kaksi havainnollistavaa esimerkkiä. Molemmat esimerkit on tarkoitettu täysin erilaisille tiimeille niiden osa-alueiden osaamisen perusteella. Lopuksi vertaillaan työtapojen toimivuutta erilaisista näkökulmista: miten tehokkuuteen voidaan vaikuttaa ja miten tiimin kokoonpano vaikuttaa toimintamallin valintaan.

2 Unreal-pelimoottori

Unreal-pelimoottori on kehitystyökalu, jonka työkaluilla voidaan luoda sovelluksia ja pelejä PC:lle, Macille, Linuxille sekä suurimmalle osalle verkkoselaimista. Se tukee VR- ja AR-laitteita, kuin myös Android- ja iOS-mobiililaitteita. [1.]

Kuva 1. Unreal-tietokonepeli [1.]

Unreal-pelimoottori on alun perin kehitetty Unreal-tietokonepeliä (kuva 1) varten, ja pelimoottorin kehitys alkoi vuonna 1995 (kuva 2). Nykyään kehittäjien käytössä oleva pelimoottori on versio 4, joka ilmestyi vuonna 2014. [1.] Unreal Engine 4:n julkaisun jälkeen pelimoottorin tekijät ilmoittivat sen olevan ilmainen kaikille käyttäjille, toisin kuin sen aikaisemmat versiot. Ilmainen käyttö on lisännyt pelimoottorin suosiota myös harrastajien joukossa. [2.]



Kuva 2. Unrealin versioiden aikajana

Versio 4 siirtyi myös pois Unrealin omasta ohjelmointikielestä Unreal Scriptistä ja otti tilalle C++-ohjelmointikielen. Muutos tehtiin siksi, koska Unreal Script jäi kehityksessä jälkeen. C++ sisälsi valmiiksi kaikki Unreal Scriptiin tarvittavat muutokset. Unreal 4 -pelimoottori perustuukin nykyään pitkälti C++-luokkien perimiseen. [3.] Unreal Engine 4:n tekijöiden tavoitteena on ollut luoda pelimoottori, jossa ohjelmoijat voivat keskittyä täysin koodin luontiin C++:lla, ja artistit sekä suunnittelijat voivat keskittyä visuaalisen ohjelmointikielen eli Blueprintien käyttöön. [4.]

2.1 C++ Unreal-pelimoottorissa

C++ on paranneltu ohjelmointikieli C-kielestä. Yleisesti C++-kieltä käytetään olio-ohjelmoinnin kaltaiseen ohjelmointiin. C++ on yli 30 vuotta vanha ohjelmointikieli, joten se on yksi vanhimmista ohjelmointikielistä, joka on edelleen yleisessä käytössä nykypäivänä. Pelimaailmassa C++:aa suositaan sen hyvän suorituskyvyn takia. C++:a pidetään keskitason ohjelmointikielenä, koska sillä on mahdollisuus ohjelmoida matalan ja korkean tason ominaisuuksia. [5.]

Unreal-pelimoottori käyttää standardin mukaista C++-koodia, ja se sallii tämän käytön täysin pelimoottorin sisällä. Unrealin C++:aa kuvaillaan sanoilla ”C++ assistentin kanssa” moottorin sisäisten ominaisuuksien ja kirjastojen takia. Unrealin C++ ei kuitenkaan muilta osin eroa normaalista C++:sta. [4.]

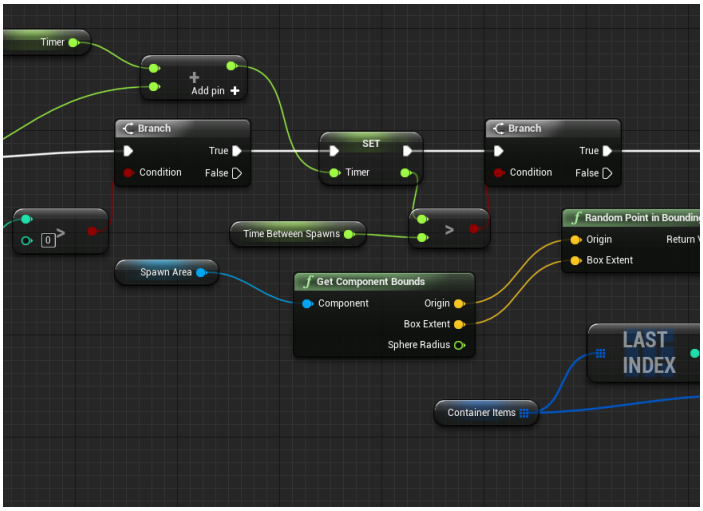
Unrealin kanssa työskentely perustuu pitkälti C++-luokkien perimiseen [2]. Luokkaa voidaan kuvailla esineen pohjapiirustukseksi, joka sisältää kaikki esineen tiedot ja ominaisuudet. Perimisellä tarkoitetaan luokan perimistä. Perittäessä jostain luokasta uusi luokka saa käyttöönsä kaikki yläluokkien tiedot ja ominaisuudet, joita ei ole yläluokissa estetty. Luotua alaluokkaa voidaan käsitellä yläluokan erikoistapauksena, koska se sisältää logiikoita ja dataa, joita ei löydy yläluokasta. Luokasta voidaan luoda pelimaailmaan esineitä eli *olioita* ja muuttaa luokan tiedot ja ominaisuudet halutun kaltaisiksi, näin luoden aina uudenlaisen olion luokasta. Tästä peritystä luokasta luotua oliota voitaisiin ajatella päivitetyksi olioksi. [6.]

Unrealissa luokkaa luodessa se useimmiten peritään jo olemassa olevasta Unrealin pääluokasta. Unrealin pääluokat sisältävät valmiiksi tärkeitä ominaisuuksia ohjelmoijan käytettäväksi. Unreal sisältää neljä pääluokkatyyppiä, joista lähes kaikki peliin lisättävät ominaisuudet peritään. Pääluokat ovat UObject, AActor, UActorComponent ja UStruct. Unreal sallii luoda myös luokkia, jotka eivät peri pääluokista, mutta silloin pelimoottorin ominaisuudet ja kirjastot ovat näiden luokkien

ulkopuolella. Esimerkki tällaisesta tilanteesta on kolmannen osapuolen kirjastot. [4.] Pääluokista kerrotaan tarkemmin luvussa 2.4.1 UCLASS().

2.2 Blueprint

Blueprintit (kuva 3) ovat Unreal-pelimoottorin visuaalinen skriptikieli, joka käyttää solmupohjaista käyttöliittymää toiminnallisuuden eli *logiikan* luomiseen. Blueprintit käyttävät samoja ominaisuuksia ja kirjastoja kuin C++. Tämän vuoksi myös suunnittelijoiden ja artistien on helppo luoda sisältöä peliin koskematta C++-koodiin. [3.]

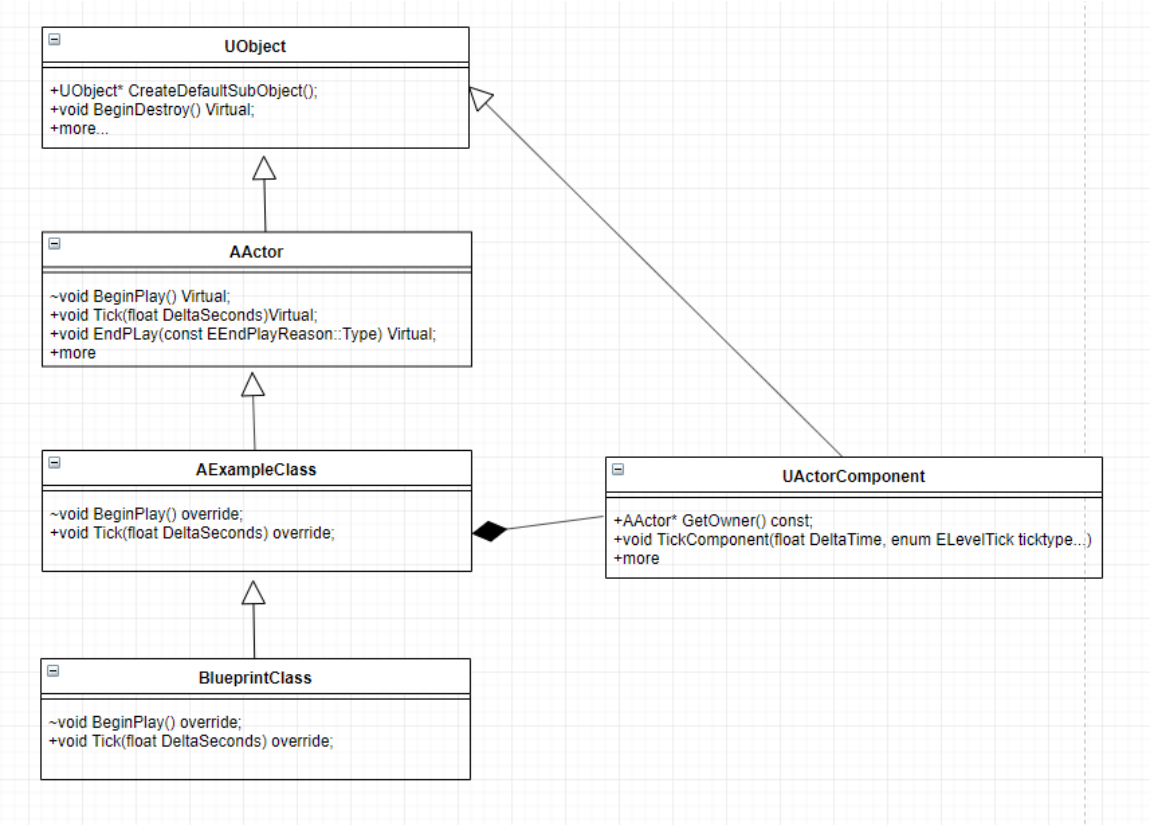


Kuva 3. Malli Blueprintilla luodusta koodista.

Yleisimpiä Blueprint-tyyppejä ovat Level Blueprint ja Blueprint Class. Kun puhutaan Unrealin Blueprinteista, yleisesti sillä tarkoitetaan Blueprint Classia. Blueprint Class on tarkoitettu peliolioiden logiikan ja datan luomiseen. Level Blueprint on tarkoitettu levelin omiin tarpeisiin. Levelillä tarkoitetaan peliolioiden kokoelmaa, josta muodostuu pelimaailma. Jokaisella levelillä onkin oma Level Blueprintinsa. [3.] Tässä opinnäytetyössä Blueprintiin viitatessa tarkoitetaan Blueprint Classia.

Blueprint Class peritään aina jostain C++-luokasta, joka voidaan asettaa pelimaailmaan. Periminen mahdollistaa pääluokkien sekä ohjelmoijan C++-koodissa luomien ominaisuuksien käytön Blueprintin sisällä. Yleistä onkin, että Blueprint Class peritään C++-luokasta, joka perii Unrealin pääluokasta (kuva 4). Luokka on myös mahdollista periä toisesta Blueprint Classista. Vaikka välissä

olisikin monta perintää, ylimpänä oleva luokka on kuitenkin aina jokin Unrealin pääluokista. Blueprint Classien vastuulle jää logiikoiden luominen ja pelin toiminnallisuuden hallitseminen yhdessä C++-koodin kanssa. [7.]

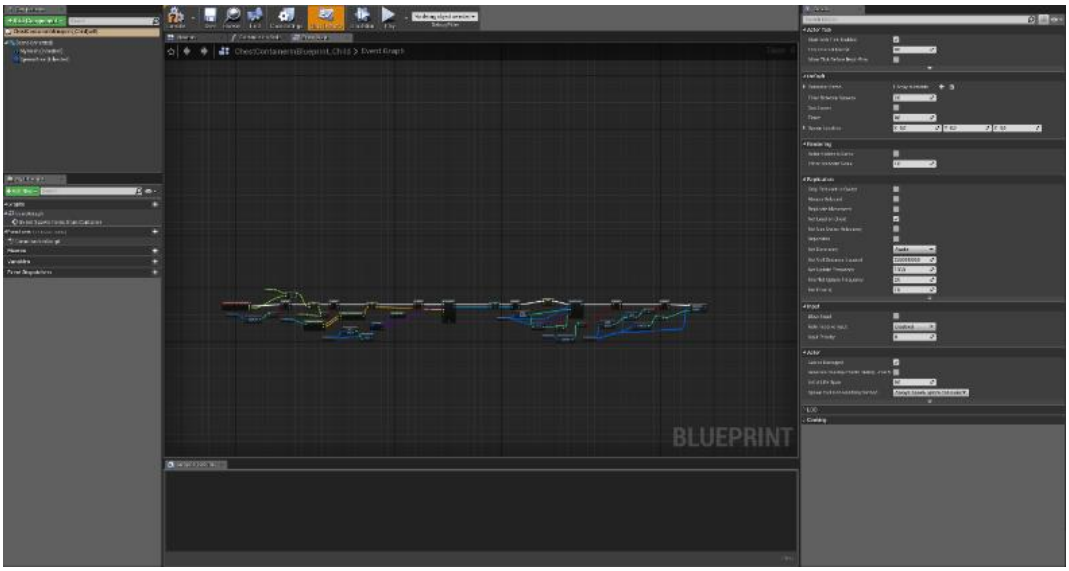


Kuva 4. Blueprint Classin perimisjärjestys

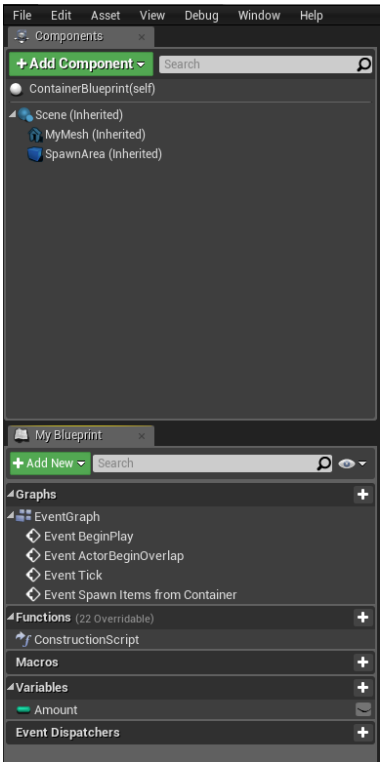
Ohjelmoijien on mahdollista laajentaa omaa koodiaan niin, että se on Blueprintin käyttäjän käytettävissä. Blueprintin tulisi olla aina jatke C++-koodille, vaikka pelin luominen ainoastaan Blueprinteillä onkin mahdollista. [3.]

2.3 Unreal-pelimoottorin Blueprint Editor

Blueprint-editori (kuva 5) on Unreal-pelimoottorin sisällä oleva työkalu, jonka avulla on mahdollista luoda pelin logiikat visuaalisia suoritettavia solmukohtia käyttäen. Blueprintit luodaan visuaalisesti Unrealin sisällä, eikä koodia tarvitse erikseen kirjoittaa. [8.]

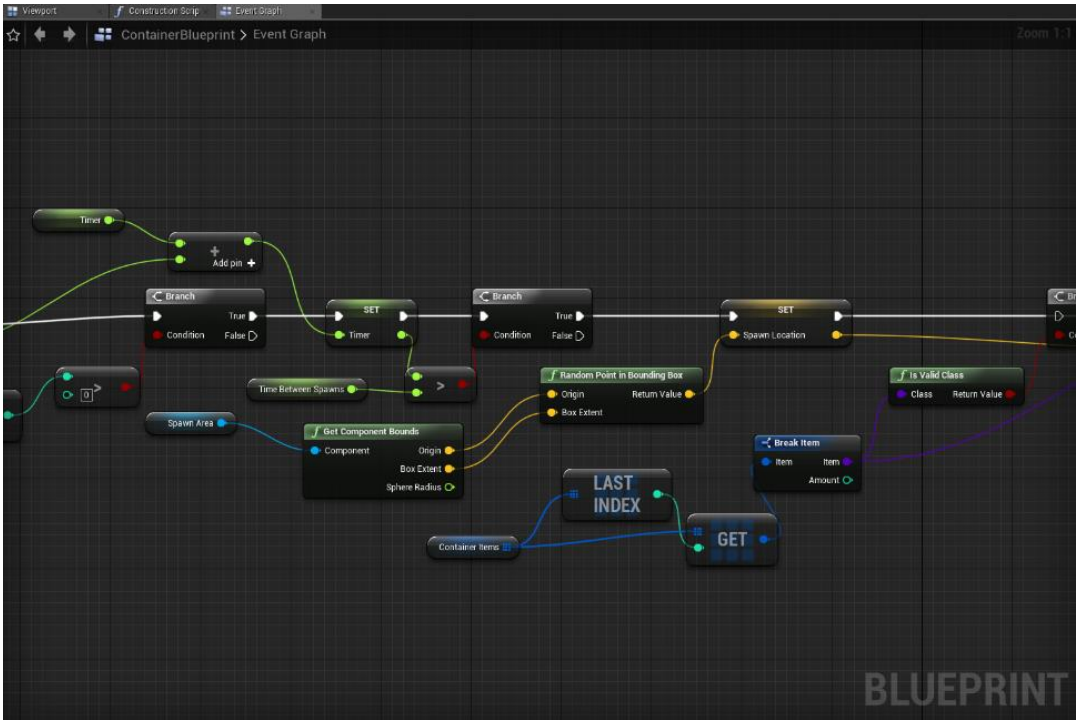


Kuva 5. Unreal-pelimoottorin Blueprint-editori.



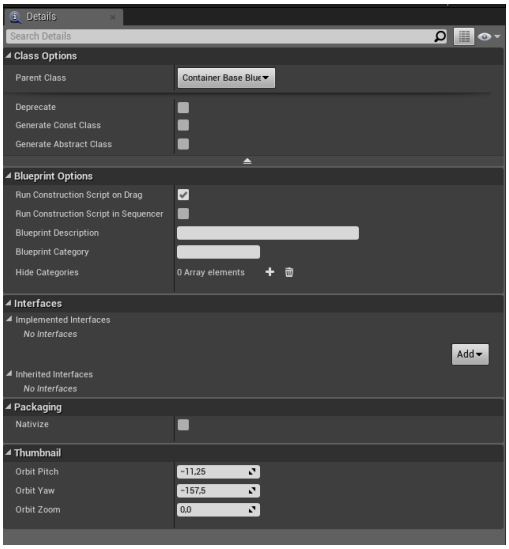
Kuva 6. Blueprint-editorin Components- ja MyBlueprint-paneelit.

Components-paneelin (kuva 6) avulla voidaan lisätä Blueprintiin erilaisia komponentteja. Esimerkkejä komponenteista ovat erilaiset primitiiviset muodot tai pelissä liikkumiseen tarkoitetut komponentit. Lisättäviä komponentteja pystyy luomaan myös itse. Components-paneelin alareunassa olevasta MyBlueprint-paneelista voi luoda, hakea ja muokata Blueprintissä toteutettavia muuttujia ja funktioita. [9.]



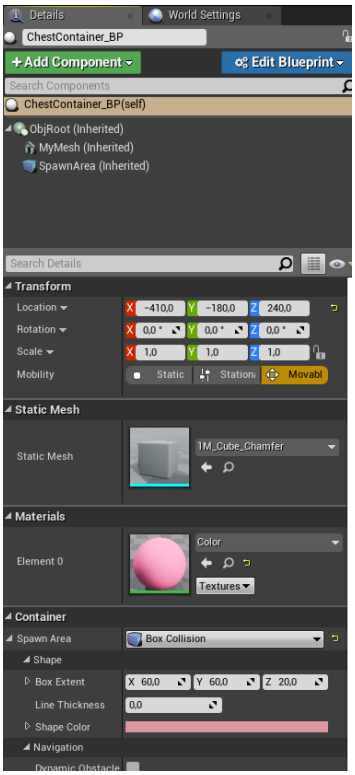
Kuva 7. Blueprint-editorin Graph-editori.

Graph-editori (kuva 7) on Blueprintin ydin, johon Blueprinteihin tehtävä logiikka luodaan. Lisäämällä ja yhdistämällä solmukohtien suoritusrivejä muodostuu suoritettavaa koodia, joka yhdistyy logiikaksi. C++-koodissa voidaan luoda Blueprintiin tulevia solmukohtia, joita kutsutaan Graph-editorin kautta. [10.]



Kuva 8. Blueprintin Details-paneeli.

Blueprintin Details-paneeli (kuva 8) on alue, joka tarjoaa muokkausoikeuden valittujen kohteiden ominaisuuksiin editorissa. C++-koodista pystyy luomaan muuttujia, joiden muokkaus mahdollistetaan Details-paneelissa käyttämällä koodissa erilaisia makroja ja niiden lisämääritteitä. [11.]



Kuva 9. Pelimaailman Details-paneeli.

Pelimaailman Details-paneeli (kuva 9) antaa tietoa sillä hetkellä pelimaailmassa eli levelissä valitusta pelioliosta. C++-koodissa pystytään erilaisia makroja ja lisämääritteitä käyttämällä luomaan samalla tavalla muokattavia ja seurattavia muuttujia pelimaailman Details-paneelissa kuin Blueprintin Details-paneelissa. [12.]

2.4 Reflection System -ominaisuus

Unreal-pelimoottorin Reflection System on ominaisuus, joka suorittaa Unrealin itsensä reaaliaikaisen tutkimisen päällä ollessaan. Ominaisuus kerää ja käsittelee Unrealin omien järjestelmien, luokkien, muuttujien ja funktioiden tietoa. C++:ssa ei ole natiivisti minkäänlaista tukea tällaiselle järjestelmälle, ja sen vuoksi Unreal-pelimoottoriin on kehitetty tämä ominaisuus. Unrealin Reflection System mahdollistaa Blueprintin ja C++-koodin keskinäisen viestinnän. [13.]

Reflection System ei luo automaattisesti Blueprintin ja C++-koodin välisiä yhteyksiä, vaan ohjelmoijan tulee luoda nämä yhteydet itse C++-koodissa. Sen jälkeen Unreal löytää yhteydet ja luo halutut ominaisuudet näkyväksi Blueprintin puolelle. [13.]

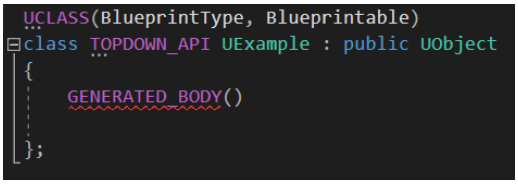
Käytännössä Reflection System tarkoittaa makrojen käyttöä. *Makro* on automaattinen syöttösarja, joka matkii näppäimistön tai hiiren toimintoja. Makroja käyttämällä voidaan minimoida toistuvan koodin kirjoittamiseen menevä aika. Halutulla syöttösarjalla on koodissa nimi, jota kutsuttaessa haluttu määrä koodia suoriutuu. [14.] Reflection Systemin kautta käytettävissä olevat makrot ovat UENUM(), UCLASS(), USTRUCT(), UFUNCTION() ja UPROPERTY() [13.].

Reflection Systemin makroja käytettäessä niille on mahdollista antaa tarkentavia lisämääritteitä parametreinä kyseisen yhteyden tarkoituksesta. Lisämääritteet määrittelevät sen, miten makro käyttäytyy Blueprintin kanssa. [3.] Lisämääritteitä on runsaasti eri tarkoituksiin, mutta tässä opinnäytetyössä esitellään niistä yleisimmin käytössä olevat.

2.4.1 UCLASS()

UCLASS()-makroa käytetään merkitsemään UObjectista perittyjä luokkia (kuva 10), jotta UObject-luokassa sijaitseva käsittelyjärjestelmä olisi tietoinen luodusta luokasta. Luokan tulee periä itsensä UObjectista saadakseen käyttöönsä tarvittavat moottorin ominaisuudet merkitsemisen onnistumiseksi. Sillä ei ole merkitystä, missä kohtaa luotu luokka on perimisjärjestyksessä, kunhan

UObject on perimisjärjestyksen ylin luokka. [15.] Luokalle mahdollisia lisämääritteitä on kymmeniä, joista tärkeimmät ovat BlueprintType ja Blueprintable [16.]. UCLASS()-merkinnän lisämäärite on periytyvä [15.].



Kuva 10. UObject-luokasta peritty luokka.

Unreal-pelimoottori sisältää neljä pääluokkaa, joista pääosa peliin luotavista C++-luokista peritään. Luokkia voidaan periä ja luoda myös Unrealin pääluokkien ulkopuolelta, mutta tällöin pelimoottorin ominaisuudet ovat näiden luokkien ulottumattomissa. [4.]

UObject

UObject on Unrealin pääluokista ylimpänä oleva luokka ja näin perusta kaikille muille luokille. UObject tarjoaa kaikki tärkeimmät pelimoottorin toiminnallisuudet, joita kehittäjän ei tarvitse useinkaan muokata. UObject-luokka toimii pohjana kaikelle, mitä peliolio tekee elämänkaarensa aikana. Suurin osa pelin logiikasta ei peri suoraan mitään UObject-luokasta, vaan AActor- tai UActorComponent-luokista, joista suurin osa pelin logiikasta peritään ja muovataan. UObjectista perityillä luokilla voi olla minkä tyyppisiä luokkia ja funktioita tahansa. Ne kuitenkin pitää olla merkitty erikseen makroilla, jotta niitä on mahdollista käyttää Unreal-pelimoottorin Blueprinteissä. [4.]

AActor

AActor perii itsensä UObject-luokasta ja on tarkoitettu peliin luotavien ja asetettavien peliolioiden luomiseen. Kaikki peliin asetettavat tai luotavat pelioliot perivät vähintään tästä luokasta. [4.]

Tärkeimmät AActorista perittävät funktiot:

- BeginPlay: Kutsutaan, kun AActorista peritty peliolio syntyy pelimaailmassa [4.].
- Tick: Funktio, joka suoritetaan tasaisin väliajoin, yleensä kerran framessa [4.].
- EndPlay: Kutsutaan, kun AActorista peritty peliolio poistuu pelimaailmasta [4.].

UActorComponent

UActorComponent on komponentti, joka lisätään AActoriin tekemään itsenäistä toiminnallisuutta. UActorComponentia käytetään usein itsenäisiin toiminnallisuuksiin tukemaan ylemmän prioriteetin ominaisuuksia AActorissa. UActorComponent perii itsensä UObjectista ja on tarkoitettu sellaisen toiminnallisuuden luomiseen, jota voidaan käyttää useissa AActoreissa. [4.]

Tärkeimmät UActorComponentista perittävät funktiot:

- GetOwner: Hakee hierarkiassa ylimmän AActorin, joka omistaa komponentin [4.].
- TickComponent: Käyttää samaa Tick-funktiota kuin hierarkiassa ylimpänä oleva AActor [4.].

UStruct

Struktit mahdollistavat mukautettujen muuttujien luonnin. Struktia käytetäänkin yleensä järjestelmällisen koodin luontiin. Kun käytetään UStructia, sitä ei tarvitse periä mistään tietystä luokasta. Luotu strukti tulee vain merkitä USTRUCT() makrolla ja Unreal-pelimoottori tekee pohjatyön. USTRUCT()-makro ei saa olla minkään muun luokan sisällä. Makro luo struktista globaalisti käytössä olevan luokan, johon muilla luokilla on pääsy. [17.]

UCLASS()-lisämääritteet

BlueprintType Määrite mahdollistaa luokan käyttämisen muuttujan tapaan Blueprintissä [3.].

Blueprintable Määrite mahdollistaa Blueprintin perimisen tästä luokasta, oletusarvolta perimінen ei ole mahdollista. Määrite on periytyvä. [3.]

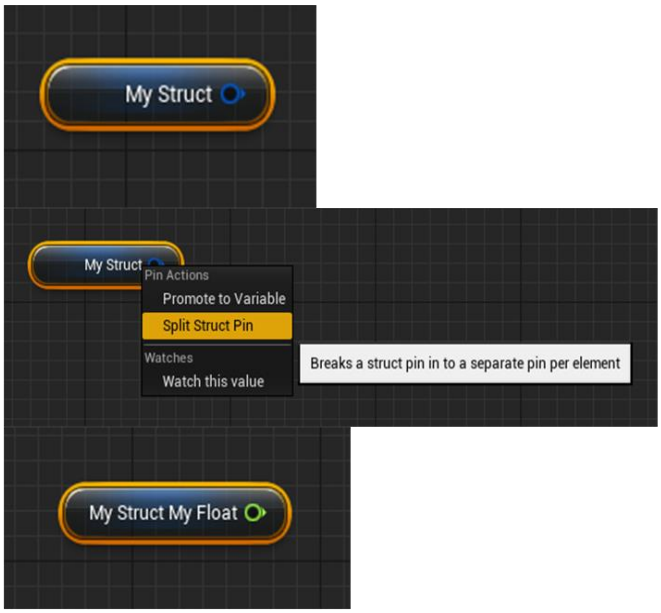
2.4.2 USTRUCT()

USTRUCT()-makro toimii samalla tavalla kuin UCLASS(). Se lisää Struct-luokan Reflection Systemille seurattavaksi. [3.] USTRUCT()in lisämääritteitä on vain muutama, joista BlueprintType on tärkein. Kun USTRUCT()-makrolle annetaan lisämäärite BlueprintType, silloin Struct-luokka (kuva 11) saadaan käytettäväksi Blueprint-muuttujan tapaan. [18.]

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

public:

    UPROPERTY(BlueprintReadWrite)
    float MyFloat;

};
```

Kuva 11. Blueprintissa käytettävän Struct-luokan luonti C++-koodissa.

C++-koodissa voidaan myös luoda Struct-luokasta olio ja merkitä se UPROPERTY()-makrolla ja antaa sille tarvittavat lisämääritteet, jotta se olisi käytettävissä C++-muuttujana Blueprintissa (kuva 12). Struktin sisällä olevat UPROPERTY()-muuttujat pitää vähintään merkitä lisämääritteellä BlueprintReadWrite, jotta ne ovat käytössä Blueprintistä käsin. [17.]



Kuva 12. C++-koodissa luotu Struct-muuttuja Blueprintissä.

Jotta struktista luodun olion sisällä olevia muuttujia olisi helpompi muuttaa Blueprintistä käsin, merkataan C++-koodissa oleva Struct-muuttuja silloin UPROPERTY()-makrolla ja sille annetaan lisämääritteet BlueprintReadWrite ja EditAnywhere. Lisämääritteet luovat struktista osion editorin

Details-paneeliin (kuva 13), jossa sitä on helppo muuttaa. Muitakin lisämääritteitä voidaan käyttää yhdessä BlueprintReadWrite-määritteen kanssa struktin sisällä. Nämä toimivat UPROPERTY()-makron lisämääritteiden tapaan. [17.]

Kuva 13. Esimerkki C++-koodissa olevasta Struct-muuttujasta editorin Details-paneelissa.

2.4.3 UPROPERTY()

UPROPERTY()-makro (kuva 14) mahdollistaa C++-muuttujan käytön Blueprintistä käsin. Muuttujan UPROPERTY()-makro estää myös Unrealin automaattista roskienkerääjää poistamasta muuttujaa niin kauan, kun jollain on viittaus omistavaan luokkaan. UPROPERTY()-makroa käytettäessä tulee omistavan luokan tai struktin olla merkitty lisämääritteellä BlueprintType, jotta luokasta voidaan luoda Blueprint-luokka. Luodun Blueprint-luokan tulee olla peritty UObjectista, koska muutoin makrot eivät ole käytettävissä. [3.]

```
UPROPERTY()
int32 Amount;
```

Kuva 14. C++-muuttuja UPROPERTY()-makrolla.

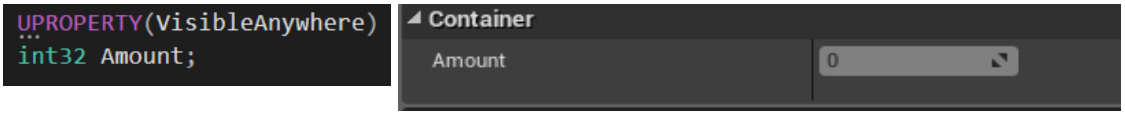
Myös UPROPERTY()-makrolle on runsaasti lisämääritteitä. Makron lisämäärite ei estä muuttujien käyttämistä C++-koodissa. [19.]

UPROPERTY()-lisämääritteet

VisibleAnywhere

Lisämäärite VisibleAnywhere on tarkoitettu C++-muuttujien arvojen seuraamiseen Unreal-pelimoottorin Details-paneelissa. Lisämäärite asettaa C++-muuttujan näkyväksi sekä Blueprintin Details-paneelissa (kuva 15) että pelimaailman Details-paneelissa sen jälkeen, kun peliolio on ase-

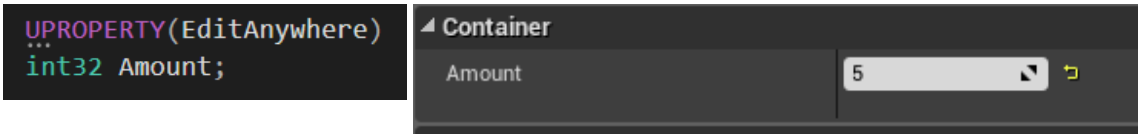
tettu pelimaailmaan. Tämän lisämääritteen avulla muuttuja ei kuitenkaan ole muutettavissa kummassakaan Details-paneelista. Lisämääritettä ei voi käyttää yhdessä Edit-lisämääritteiden kanssa, koska ne kumoaisivat toisensa. [19.]



Kuva 15. VisibleAnywhere-lisämääritteen C++-muuttuja näkyvissä Details-paneelissa.

EditAnywhere

Lisämäärite EditAnywhere asettaa muuttujan näkyväksi ja muutettavaksi sekä Blueprintin Details-paneelissa (kuva 16) että pelimaailman Details-paneelissa sen jälkeen, kun peliolio on asetettu pelimaailmaan. Tätä lisämääritettä onkin hyvä käyttää aloitusarvojen asettamiseen ja testaamiseen ilman, että tarvitsee muokata suoraan C++-koodia. Lisämääritettä voidaan hyödyntää luomaan eri variaatioita luokasta luoduista peliolioista ja pitämään ominaisuuksien teko joustavana. Lisämääritettä ei voi käyttää yhdessä Visible-lisämääritteiden kanssa, koska ne sisältävät toisiaan kumoavia ominaisuuksia. [19.]



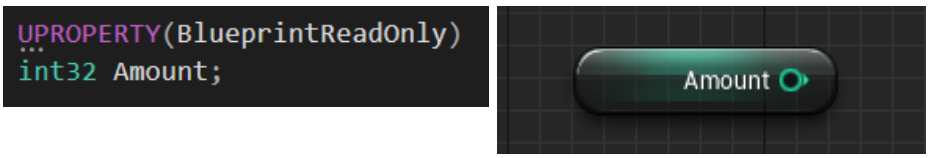
Kuva 16. Details-paneeleissa muutettava C++-muuttuja.

EditDefaultsOnly

Jos muuttuja halutaan muokattavaksi ainoastaan Blueprintin Details-paneeliin, voidaan makrolle antaa lisämäärite EditDefaultsOnly. Tämä lisämäärite toimii samalla tavalla kuin EditAnywhere sillä erolla, että se on muutettavissa ainoastaan Blueprintin Details-paneelissa. Kaikilla Blueprint-luokasta pelimaailmaan luoduissa peliolioissa on Blueprintissä asetetut aloitusarvot. Lisämääritettä ei voi käyttää yhdessä Visible-lisämääritteiden kanssa. [19.]

BlueprintReadOnly

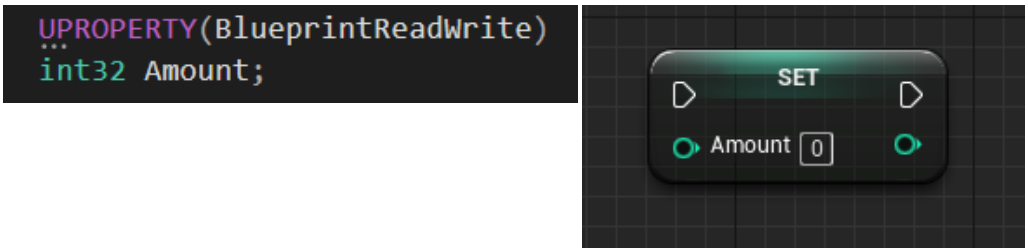
Lisämäärite BlueprintReadOnly luo C++-muuttujasta Blueprintissä käytettävän solmukohdan (kuva 17), jolla C++-muuttujan arvo saadaan Blueprintin Graph-editoriin käytettäväksi. Tällä lisämääritteellä C++-muuttujan arvoa ei voida muuttaa, vaan se voidaan ainoastaan hakea C++-koodista. [19.]



Kuva 17. Blueprintin solmukohta, joka hakee C++-muuttujan.

BlueprintReadWrite

Lisämäärite BlueprintReadWrite luo muuttujasta kaksi solmukohtaa käytettäväksi Blueprintin Graph-editoriin. Toinen on sama BlueprintReadOnly-lisämääritteellä (kuva 17), jolla voi vain hakea muuttujan arvon C++-koodista. Toinen lisämääritteen luoma solmukohta (kuva 18) mahdollistaa C++-muuttujan arvon muuttamisen Blueprintistä. Lisämääritettä ei ole mahdollista käyttää yhdessä BlueprintReadOnly-lisämääritteen kanssa. [19.]



Kuva 18. Suoritettava Blueprintin solmukohta, jonka avulla voidaan muuttaa muuttujan arvoa C++-koodissa.

2.4.4 UFUNCTION()

UFUNCTION()-makron (kuva 19) tarkoitus on mahdollistaa C++-funktioiden kutsun Blueprinteissä funktion omistavasta tai siitä perityistä luokasta. C++-koodista voidaan kutsua funktiota, joka lisämäärittеidensä avulla kutsuu Blueprint-funktiota C++-koodista käsin. UFUNCTION()-makroa käytettäessä tarvitsee omistavan luokan tai struktin olla merkitty lisämääritteellä BlueprintType, jotta siitä voidaan luoda Blueprint-luokka. [3.]

```
UFUNCTION()
void ExampleFunction();
```

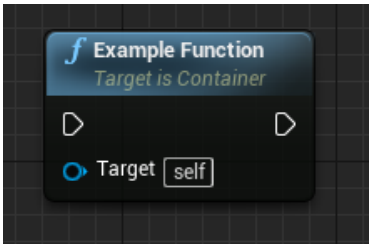
Kuva 19. UFUNCTION()-makro C++-koodissa.

UFUNCTION()-makron lisämääritteet

BlueprintCallable

BlueprintCallable-lisämäärite mahdollistaa C++-funktioiden kutsun Blueprintistä. Määritteellä luodaan suoritettava solmukohta Blueprintiin (kuva 20). [20.]

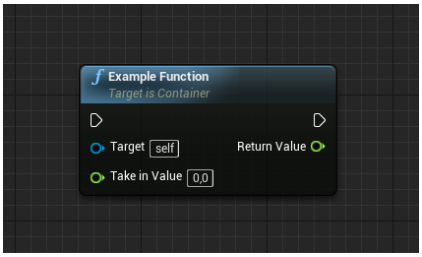
```
UFUNCTION(BlueprintCallable)
void ExampleFunction();
```



Kuva 20. C++-funktion suoritettava solmukohta Blueprintissä.

BlueprintCallable-lisämääritettä käyttäessä on mahdollista luoda funktiolle sisään otettavia parametrejä (kuva 21). Kun Blueprintissä kutsutaan C++-funktiota, ottaa se tarvittavat parametrit sisäänsä. Lisämäärite myös tunnistaa sen, onko funktiolla jokin palautusarvo ja antaa tämän käytettäväksi Blueprintissä. [21.]

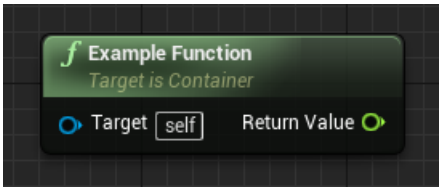
```
UFUNCTION(BlueprintCallable)
float ExampleFunction(float TakeInValue);
```



Kuva 21. BlueprintCallable-lisämääritteellinen funktio sisään otettavalla parametrilla ja palautusarvolla.

Kun funktiossa käytetään avainsanaa *const* (kuva 22), silloin Blueprintin solmukohdassa ei ole suoritusriviä. Const-avainsana takaa, että funktio ei muuta mitään tietoja, vaan funktio hakee vain tiedon C++-koodista. Const-avainsanaa käyttäen on helppo luoda getter-funktioita, jotka vain hakevat tietoja C++-koodista Blueprintiin. Tätä ominaisuutta voidaan käyttää silloin, jos ei haluta noudettavan tiedon olevan julkista tai muutettavissa koodissa. [21.]

```
UFUNCTION(BlueprintCallable)
float ExampleFunction() const;
```

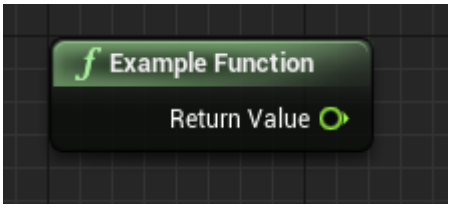


Kuva 22. Const-avainsanan käyttö BlueprintCallable-lisämääritteen kanssa.

BlueprintPure

BlueprintPure-lisämäärite on samantapainen kuin BlueprintCallable const-avainsanan kanssa. Lisämääritettä käytetään vain tietojen hakemiseen C++:sta ilman, että funktion kutsu itsessään vaikuttaa pelin tilaan. Lisämäärite mahdollistaa funktion kutsun C++-koodista ilman solmukohdassa olevaa suoritusriviä (kuva 23). Lisämäärite on erinomainen C++-koodissa suoritettaville matemaattisille toiminnoille. Lisämääritettä käytettäessä funktiolla tulee olla palautusarvo. [21.]

```
UFUNCTION(BlueprintPure)
float ExampleFunction();
```

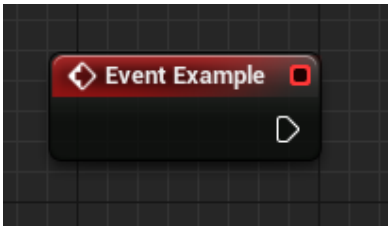


Kuva 23. BlueprintPure-lisämäärite C++-koodissa ja käytettävä solmukohta Blueprintissa.

BlueprintImplementableEvent

BlueprintImplementableEvent-lisämäärite mahdollistaa Blueprintin Eventin kutsun C++-koodista käsin (kuva 24) [20.]. Blueprintin Eventit ovat solmukohtia, joita kutsutaan C++-koodista aloittamaan Blueprinteissä yksittäinen suorituskerta. Suoritus alkaa kutsutun Eventin solmukohdasta. [22.] C++-koodissa Blueprint Event on normaali funktio, jolle on annettu BlueprintImplementableEvent-lisämäärite. [21.]

```
UFUNCTION(BlueprintImplementableEvent)
void Example();
```



Kuva 24. C++-koodista kutsuttava Blueprintin Event.

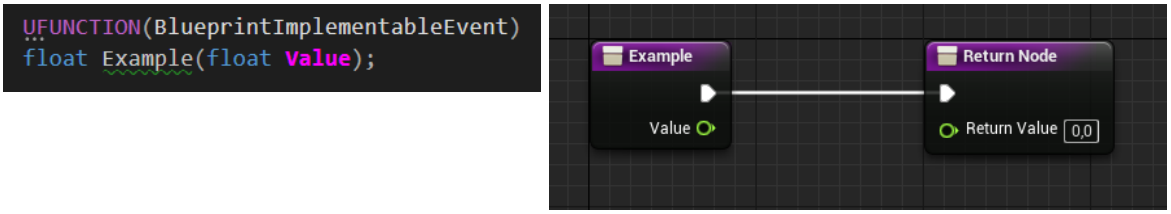
Eventin avulla Blueprinteissä voidaan reagoida pelissä tapahtuviin tapahtumiin. [20.] BlueprintImplementableEvent-lisämääritettä käytetään silloin, kun logiikka on luotu Blueprintiin C++-koodin sijaan. Funktiolla ei saa olla C++-suoritusta, vaan ainoastaan Blueprintissä oleva suoritus. Jos BlueprintImplementableEvent-lisämääritettä käytetään yhdessä BlueprintCallable-lisämääritteen kanssa, voidaan Eventiä kutsua myös suoraan Blueprintistä. Lisämääritteellä luodulle funktiolle voidaan myös antaa parametrejä, jotka tällöin kulkeutuvat C++-koodista Blueprintin suoritettavaan Eventiin (kuva 25). [21.]

```
UFUNCTION(BlueprintImplementableEvent)
void Example(float Value);
```

Kuva 25. C++-koodista kutsuttava Blueprint Event, jolle annetaan kutsuttaessa parametri.

Jos BlueprintImplementableEvent-funktiota kutsuttaessa halutaan funktion palauttavan jotain, täytyy se rakentaa hieman eri tavalla. Yllä oleva esimerkki ei palauta mitään, vaan se vain suorittaa halutun logiikan. Kun C++-koodissa luodun BlueprintImplementableEvent-funktion halutaan palauttavan jotain, täytyy funktio korvata Blueprintissä. Tällöin siitä tehdään C++-funktion kor-

vaava Blueprint-funktio, eikä kutsuttava Event. BlueprintImplementableEvent-funktiota kutsutaan normaalisti C++-koodissa, mutta sen logiikka luodaan Blueprinteihin ja viimeisestä suoritettavasta solmukohdasta funktion suoritus palautuu C++-koodiin (kuva 26). [21.]

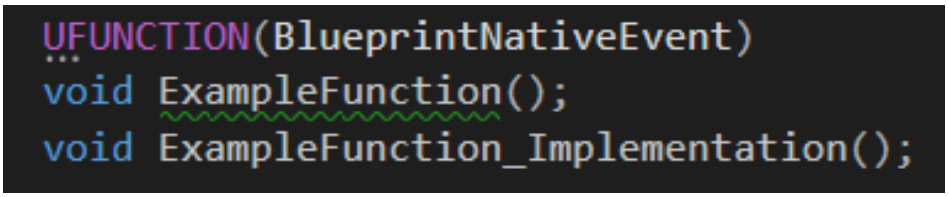


Kuva 26. Blueprintissä korvattu C++-funktio.

BlueprintNativeEvent

BlueprintNativeEvent-lisämäärite mahdollistaa Blueprint-funktion kutsun C++-koodista samalla tavalla kuin BlueprintImplementableEvent. BlueprintNativeEvent etsii käytettävissä olevan C++-funktion ja kutsuu sitä. Jos Blueprintissä ei löydy kutsuttavaa Eventiä tai funktiota, Blueprintistä kutsutaan C++-funktion natiivia versiota.

Funktion implementointi C++-koodissa eroaa hieman muista lisämääritteistä, sillä C++-koodissa täytyy luoda funktiosta kaksi eri versiota, joista toinen on normaali implementaatio ja toinen *_Implementation*-jälkiliitteellä (kuva 27). Perusimplementaatiolla ei ole C++-koodissa suoritettavaa logiikkaa. Jälkiliitteellä oleva versio sisältää C++-koodissa olevan logiikan suorituksen, jos se ei löydä korvattua funktiota Blueprintistä. [20.]



Kuva 27. _Implementation-jälkiliite C++-koodissa

3 C++-koodin ja Blueprintien rinnakkaiskäyttö

Unreal-pelimoottorissa voidaan yhdistää kahta koodaustapaa sujuvasti keskenään. Molempiin tapoihin voidaan luoda logiikkaa ja tallentaa dataa, ja ominaisuudet voidaan implementoida C++-koodiin, Blueprinteihin tai näiden yhdistelmään [23.]. Myös työtapojen hyödyntämisen suhde voi vaihdella.

Jos tavoitteena on saada helppoa ja nopeaa muokattavuutta, voidaan silloin käyttää Blueprintejä (taulukko 1). Jos taas esimerkiksi halutaan maksimoida pelin suorituskyky, käytetään silloin C++-koodia (taulukko 2). Projektin luomistapa pitää aina rakentaa sen tekijöiden taitojen ympärille niin, että projektia on mahdollisimman helppo hallita. Se, millainen versio on parempi, on aina tiimikohtaista. Useat eri tekijät vaikuttavat siihen, kumpaa koodaustapaa käytetään. [24.]

Blueprintin edut

Työnopeus	Blueprintin työstäminen on paljon nopeampaa kuin C++:n.
Iteraation nopeus	On nopeampaa muuttaa Blueprintin logiikkaa sekä dataa ja nähdä muutos suoraan editorissa kuin muuttaa C++-koodia.
Taito	Blueprintit eivät vaadi välttämättä minkäänlaista erityisosaamista.
Data	On helpompaa luoda uusia asioita peliin luomalla C++-luokasta Blueprint ja muokata muuttujat haluttuihin arvoihin kuin muokata C++-koodia.

Taulukko 1. Blueprintin edut. [23.]

C++-koodin edut

Suorituskyky	Huomattavasti nopeampi suorituskyky kuin Blueprintissä.
Suunnittelu	Tekijällä on täysi kontrolli tekemäänsä työhön. Voidaan tarkasti päättää, mitä näytetään ja jaetaan Blueprinteille.
Käyttöoikeus	C++:lla on täysi pääsy kaikkiin muihin pelin järjestelmiin, jos niin halutaan. Pelimoottorin kaikki ominaisuudet ovat myös käytössä.

Matikka	Matemaattisten laskujen teko on selkeämpää ja tehokkaampaa C++:ssa kuin Blueprintissä.
Versionhallinta	Koska koodi on tekstiä, on sitä helpompi hallita ja yhdistellä versionhallintatyökalujen kanssa. Blueprinteissä täytyy yleensä aina ylikirjoittaa vanha versio, ja tämä voi rikkoa ominaisuuksia.

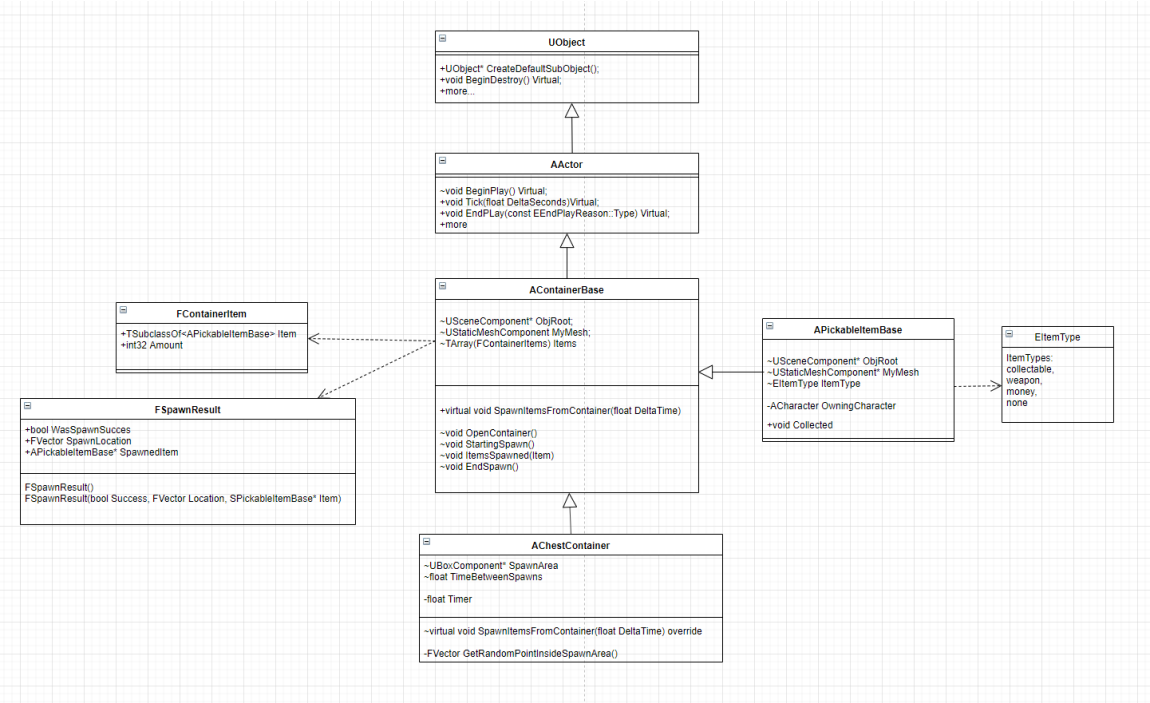
Taulukko 2. C++-koodin edut. [23.]

Tässä opinnäytetyössä tuodaan esille kaksi vaihtoehtoista työtapaa yhdistää C++-koodi ja Blueprintit. Esimerkkityön avulla havainnollistetaan, miten voidaan yhdistää C++-koodin suorituskyky ja Blueprintien muokattavuus. Ensimmäisessä versiossa painotetaan enemmän C++:aa ja toisessa Blueprintejä. Molemmissa esimerkkiversioissa käytetään hyväksi C++:n perinnöllisyyttä. Lopuksi näitä työtapoja pohditaan tehokkuuden, suorituskyvyn ja työryhmän näkökulmasta.

3.1 Esimerkkejä yhteiskäytöstä

Esimerkkinä C++-koodin ja Blueprintien yhteiskäytöstä voidaan käyttää pelimaailmaan luotavaa arkkua, joka avautuessaan luo sen sisällä olevat pelioliot, esimerkiksi kolikot, pelimaailmaan. C++-koodiin luodusta arkkuluokasta on mahdollista luoda Blueprint-luokkia, joiden avulla voidaan luoda arkkuja erilaisilla ominaisuuksilla tai sisällöillä. Jotta Blueprintin ja C++:n yhteiskäytössä voitaisiin luoda omia logiikoita ja muuttujia C++-koodiin niin, että ne olisivat käytössä Blueprintissä, tulisi niitä varten luoda C++-koodista peritty Blueprint-luokka. Blueprint-luokista ominaisuuksien arvoja on yksinkertaisempaa muokata. Jos työssä aiotaan käyttää C++-koodia, Blueprint-luokkia luodessa ne tulisi aina luoda Unrealin pääluokasta peritystä C++-luokasta.

Aluksi luodaan halutunlainen alaluokka arkulle, joka peritään yläluokasta. Yläluokka sisältää ne funktiot ja muuttujat, joita kaikki siitä mahdollisesti perittävät alaluokat tarvitsevat. Samasta yläluokasta voidaan esimerkiksi luoda avattava arkku tai pyörivä tynnyri. Arkku ei kuitenkaan tarvitse pyörimiseen tarvittavaa logiikkaa, mutta molemmat alaluokat tarvitsevat yläluokassa olevia ominaisuuksia. Kuvassa 28 on esitetty arkkuesimerkin koodiarkkitehtuuri.



Kuva 28. UML-suunnitelma arkun koodiarkkitehtuurista.

3.1.1 C++-painotteinen versio

Tässä versiossa kaikki logiikka ja data luodaan C++-koodiin ja Blueprinteihin lisätään ainoastaan visuaalisiin tarkoituksiin tarvittavat ominaisuudet. Logiikka toimii täysin C++-koodissa, jotta peliin saataisi paras suorituskyky. Jos jotain logiikkaa tai dataa halutaan muokata, pitää muutos tehdä silloin C++-koodiin. Tätä varten kuitenkin logiikan säätämiseen tarvittavien muuttujien olisi hyvä olla muutettavissa vähintään Blueprintien Details-paneelissa. Esimerkiksi arkusta luotavien pelioliioiden eli esimerkiksi kolikoiden frekvenssi on yksinkertaisempi säätää Blueprintin Details-paneelissa (kuva 29) kuin C++-koodin kautta.

Kuva 29. Luomisen välinen frekvenssi Blueprintin Details-paneelissa

```
UPROPERTY(EditDefaultsOnly)
...
float TimeBetweenSpawns;
```

Kuva 30. Luomisen välinen frekvenssi

Frekvenssin määrittelevä muuttuja merkitään UPROPERTY()-makrolla ja sille annetaan EditDefaultOnly-lisämäärite (kuva 30). Näin oikean frekvenssivälin löytäminen on helppoa Blueprinteistä käsin muuttujan kuitenkin säilyessä C++-koodissa.

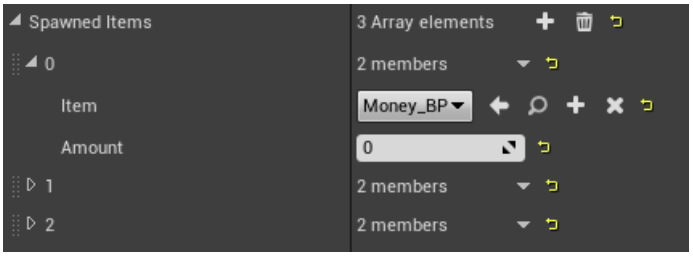
Arkku-esimerkissä arkulle voidaan antaa vaihtoehtoisia sisältöjä kolikoiden lisäksi. Kolikko voisi olla esimerkiksi jalokivi, tai kolikon arvo voisi vaihdella. Nämä vaihtoehdot on hyvä listata taulukkomuotoon TArray-muuttujan avulla, jotta jokaista oliota ei tarvitse luoda erikseen muuttujana arkun luokkaan.

Jotta arkussa olevia peliolioita olisi helppo lisätä ja muokata, pitää luoda taulukko eli TArray-muuttuja. Sitä varten täytyy luoda C++-koodissa Struct-luokka, jota voidaan käyttää muuttujan tapaan listan luomiseen. Luotu Struct-luokka sisältää luotavan peliolion luokan ja niitä luotavan määrän (kuva 31). Struct-luokka tulee sijoittaa yläluokkaan, jotta se olisi kaikille käytössä yläluokasta perittäessä. Taulukon jokainen elementti sisältää Struct-luokasta luodun olion tiedot, joiden pohjalta arkun pelioliot luodaan.

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

public:

    FContainerItem() {}

    UPROPERTY(EditDefaultsOnly)
        TSubclassOf<APickableItemBase> Item;

    UPROPERTY(EditDefaultsOnly)
        int32 Amount;

};
```

Kuva 31. Struct-luokka C++-koodissa.

Jotta taulukko saadaan muokattavaksi Blueprintin Details-paneeliin (kuva 32), luotu muuttuja merkitään UPROPERTY()-makrolla ja sille annetaan EditDefaultsOnly-lisämäärite (kuva 33). Arkun sisältö on nyt helppo määrittää Blueprintin Details-paneelissa, mutta sitä käytetään pelin aikana C++-koodissa. Tämä helpottaa huomattavasti työtä, kun arvojen muuttamiseksi niitä ei tarvitse etsiä koodin seasta.



Kuva 32. Luotu lista struktista Blueprintin Details-paneelissa.

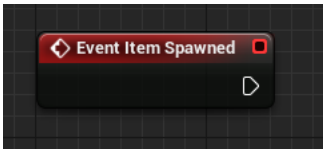
```
UPROPERTY(EditDefaultsOnly)
    TArray<FContainerItem> SpawnedItems;
```

Kuva 33. Struktista luotu TArray-taulukko EditDefaultsOnly-lisämääritteellä.

C++-painotteisessa versiossa luokan logiikka ja data peliolioiden luomiseen pysyy siis C++-koodissa ja Blueprinteihin paljastetaan vain olennaiset osa-alueet. Logiikan ja datan luominen koodiin maksimoi suorituskyvyn ja luettavuuden. Blueprintin Eventien kutsu C++-koodista käsin mahdollistaa sen, että koko tiimi pystyy luomaan oman osa-alueensa logiikoita peliin. Blueprintejä käyttävän ei tarvitse välittää C++-koodin mysteereistä, vaan hän voi käyttää käytössä olevia välineitä Blueprintin puolella. Ohjelmoijat voivat keskittyä suorituskyvyllisen koodin kirjoittamiseen.

C++-koodista kutsutaan Blueprintin Eventejä Blueprinteissä olevan logiikan laukaisemiseksi. Kun peliolio luodaan, voidaan kutsua Blueprint Eventiä. Blueprint Eventiin voi esimerkiksi artisti luoda partikkeliefektin ilmoittamaan uuden peliolion luonnista. C++-koodissa luodaan koodissa kutsuttava Blueprint Event luomalla funktio, jolle annetaan UFUNCTION()-makro ja BlueprintImplementableEvent-lisämäärite (kuva 34). Jotta Blueprintien käyttäjien olisi helpompi löytää C++-koodista tulevat ominaisuudet, kannattaa kaikille makroille antaa Category-lisämäärite, jonka avulla C++-koodista tulevien ominaisuuksien löytäminen on helpompaa. Tässä esimerkissä arkulle on annettu nimi ”Container”.

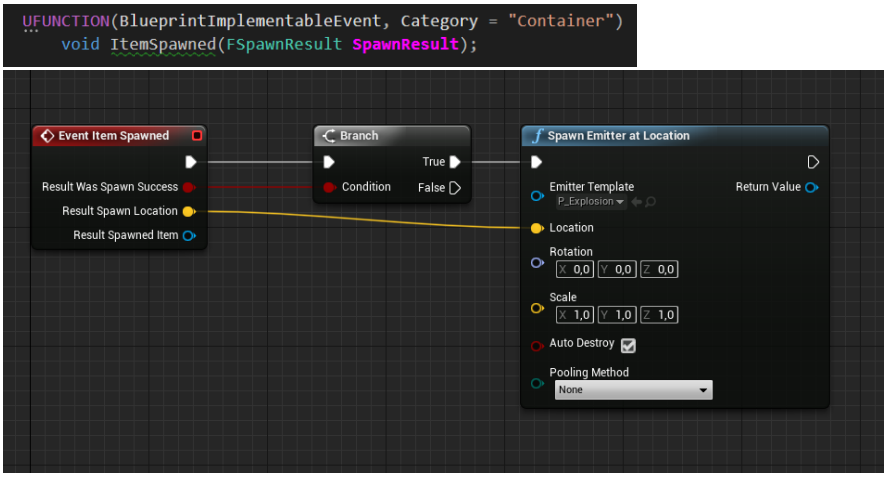
```
UFUNCTION(BlueprintImplementableEvent, Category = "Container")
    void ItemSpawned();
```



Kuva 34. Blueprint Eventin luonti C++-koodissa.

Eventille voidaan antaa myös parametreja. Parametrien avulla voidaan antaa esimerkin partikkeliefektille sen keskipiste pelimaailmassa. Näitä parametrejä voidaan siis käyttää apuna logiikoiden luomiseen Blueprinteissä.

Parametreissä voidaan helposti hyödyntää lisää struktien voimaa ja antaa Eventin kutsulle Struct-olion sisällä olevat muuttujat yhtenä parametrinä. Struct-olio sisältää tarvittavat tiedot peliolion luomisesta. Kun Eventiä kutsutaan C++-koodissa, tarvitsee se parametrit sisäänsä, jotta ne olisivat käytössä Blueprintissä. Kutsun yhteydessä siis laukaistaan Blueprinteissä olevan Eventin logiikka (kuva 35). Blueprintissä oleva logiikka suoritetaan ennen kuin C++-koodin suoritusta jatketaan.



Kuva 35. Blueprint Eventin logiikkaa.

3.1.2 Blueprint-painotteinen versio

Tässä versiossa logiikka ja data luodaan pääsääntöisesti Blueprinteihin, mutta tarvittaessa koodiin lisätään C++-funktioita ja muuttujia, joita kutsutaan Blueprinteistä käsin. Näin peliolion logiikan luominen ja muokkaus pysyy Blueprinteissä ja siten koko tiimin muutettavissa, esimerkiksi sellaisessa tiimissä, jossa ohjelmointitaitoisia on vähemmän. Samalla pidetään pelin suorituskykyä ja luettavuutta yllä luomalla osa logiikasta C++-koodiin ja luodaan niistä Blueprinteissä suoritettavia solmukohtia.

Arkun sisällöstä luodaan edellisen esimerkin tapaan taulukko, joka sisältää struktista luotuja olioita. Tässä versiossa kuitenkin data halutaan pitää mahdollisimman paljon Blueprintien puolella, joten strukti ja siitä luotava taulukko luodaan Blueprinteissä. Tämä kuitenkin tarkoittaa sitä, että C++-koodilla ei ole yksinkertaista pääsyä tähän dataan.

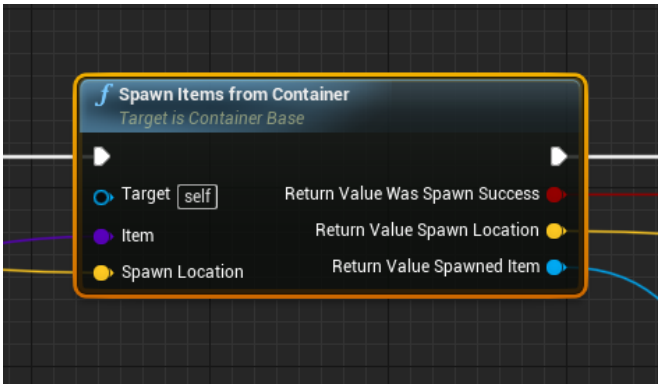
Tässä versiossa peliolioiden luomiseen käytettävä logiikka lisätään C++-koodiin ja sitä kutsutaan Blueprinteista käsin. C++-koodissa käytetään hyväksi perinnöllisyyttä. Yläluokkaan luodaan korvattava funktio (kuva 36), jota kutsutaan Blueprinteistä. C++-koodissa perivä luokka korvaa yläluokan funktion (kuva 37) ja lisää alaluokkaan sopivan toiminnallisuuden. Näin voidaan aina kutsua samaa funktiota, vaikka arkun käyttötarkoitus olisi eri. Blueprintin käyttäjä kutsuu aina samaa funktiota, vaikka Blueprint-luokka olisikin peritty täysin erilaisesta alaluokasta. Yläluokassa oleva funktio merkitään UFUNCTION()-makrolla ja sille annetaan BlueprintCallable-lisämäärite, jotta se olisi kutsuttavissa Blueprintistä käsin.

```
UFUNCTION(BlueprintCallable, Category = "Container")
virtual FSpawnResult SpawnItemsFromContainer(TSubclassOf<APickableItemBase> Item, FVector SpawnLocation);
```

Kuva 36. Blueprintistä kutsuttava C++-funktio yläluokan headerissa.

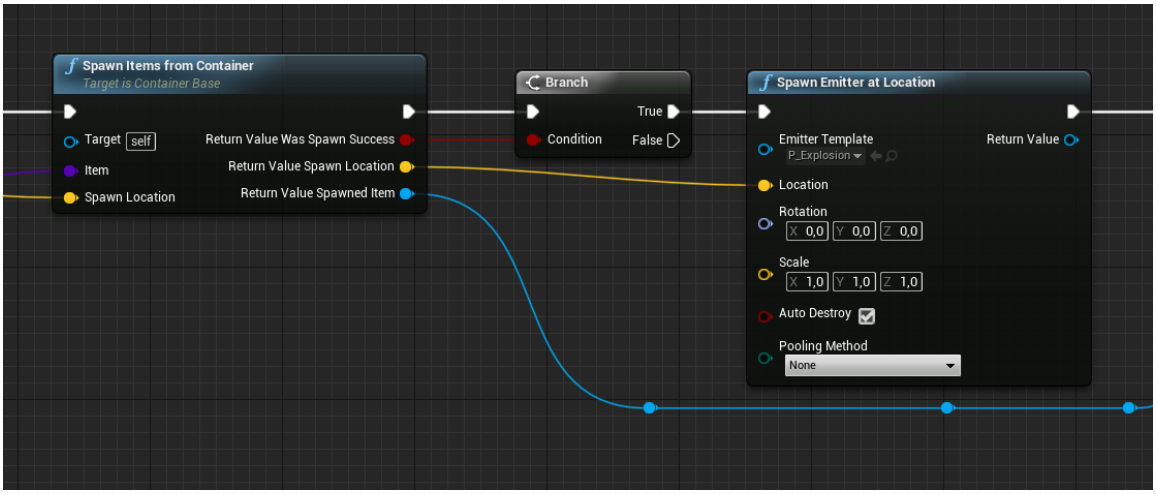
```
FSpawnResult SpawnItemsFromContainer(TSubclassOf<APickableItemBase> Item, FVector SpawnLocation) override;
```

Kuva 37. Yläluokasta korvattu C++-funktio headerissa.



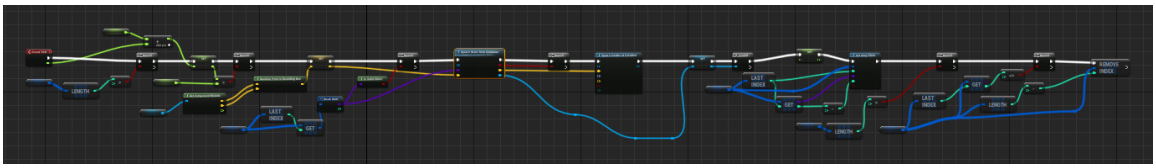
Kuva 38. SpawnItemsFromContainer C++-funktion kutsu Blueprinteissä.

Funktion kutsun yhteydessä palautetaan C++-koodista luomisen tulokset struktin muodossa (kuva 38). Niiden perusteella Blueprinteissä voidaan logiikan suoritusta jatkaa. Esimerkiksi jos luominen oli onnistunut, voidaan pelimaailman haluttuun kohtaan luoda partikkeliefekti osoittamaan peliolion syntyä. Luomisen tulosten palauttaminen on hyvä esimerkki siitä, miksi peliolioiden luominen halutaan tehdä C++-koodin puolelle. Tuloksiin voidaan tarvittaessa lisätä lisää muuttujia, joiden avulla suoritusta voidaan jatkaa tai muuttaa. Blueprinteissä nähdään silloin selkeästi muuttujat ja niiden arvoja on helppo käyttää (kuva 39).



Kuva 39. Logiikan jatkaminen Blueprintissä peliolion luomisen tulosten mukaan.

Blueprint-painotteisessa versiossa toiminnallisuus on rakennettu Blueprinteihin, joten solmukohtia on runsaasti verrattuna C++-painotteiseen versioon (kuva 40). Vaikka Blueprint-painotteisessa versiossa logiikoiden teko yritetäänkin yleisesti pitää Blueprinteissä, olisi sen selkeyden vuoksi hyvä lisätä logiikoita myös C++-koodin puolelle. C++-funktioihin laitettavan logiikan laajuus riippuu täysin tiimistä ja sen kokoonpanosta.



Kuva 40. Esimerkin logiikka luotuna Blueprinttiin.

Blueprintien tärkeiden solmukohtien ympärille muodostuu usein muuttujien muuttamista sisältäviä logiikoita, jotka tekevät Blueprintien lukemisesta monimutkaista ja hidastavat suoritustehoa. Luomiseen tarkoitetun C++-funktion sisällä olevaa logiikkaa olisi myös voitu jättää Blueprinteihin ja kutsua vain Unreal-pelimoottorista valmiiksi löytyvää yksinkertaista Blueprintin suoritettavaa solmukohtaa, joka synnyttää peliolion haluttuun kohtaan maailmassa. Se ei kuitenkaan mahdollista luomisen yhteydessä haluttuja toiminnallisuuksia tai C++-perinnöllisyyden hyötyjä.

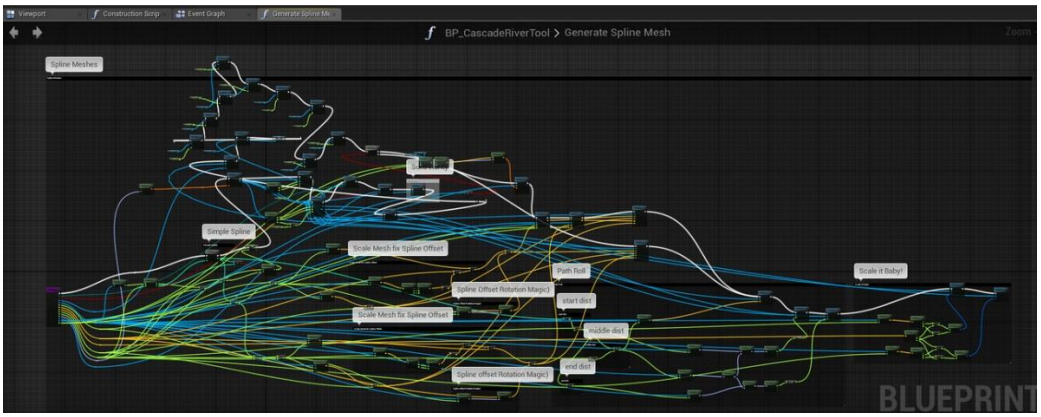
Luomisen yhteydessä olisi myös C++-koodissa esimerkiksi mahdollista alustaa luotu peliolio halutulla tavalla tai käyttää ensimmäisen esimerkin Blueprint Eventtejä hyväksi laukaisemaan yksittäinen suorituskerta. Mitä enemmän toiminnallisuutta lisätään Blueprinteihin, sitä vaikeampi niitä on lukea ja sitä enemmän niiden suoritusteho kärsii.

3.2 Tehokkuus ja suorituskyky

Tehokkuuden näkökulmasta ajateltuna sillä on paljon merkitystä, kumpaan työtapaan valtaosa logiikoista rakennetaan. Suorituskyvyltään C++-koodi on Blueprinttiä huomattavasti tehokkaampi. Tämän takia paljon monimutkaisia tietorakenteita ja silmukoita sisältävien ominaisuuksien tulisi aina olla C++-koodina suorituskyvyn ylläpitämiseksi. Monesti kaikkea ei kuitenkaan kannata siirtää koodin puolelle, vaan ainoastaan raskaimmat laskutoimitukset. Tämä pitää myös Blueprintin muiden ominaisuuksien muutosten teon joustavana.

Monesti Blueprintin suorituskyky ei kuitenkaan ole ongelma käytännössä. Suurin ero ohjelman suorituksessa on, että yksi solmukohta Blueprintissä on hitaampi kuin yksi rivi C++-koodissa. Kun suoritus on solmukohdan sisällä, se on aivan yhtä nopeaa kuin jos sitä olisi kutsuttu C++-koodista. Suorituskyvyn kannalta ei siis ole merkittävää, kuinka raskaita yksittäiset solmukohdat ovat. Jos Blueprint taas sisältää paljon haarautuvia sisäkkäisiä lausekkeita tai muuttujiin tehdään paljon muutoksia, tulisi ne luoda C++-koodin puolelle. [4.]

Pelkällä Blueprintillä saa myös monesti sotkua aikaan. Näissä tilanteissa logiikat olisi jo alun perin pitänyt luoda C++:n puolelle, jotta Blueprintissä oleva osuus säilyisi selkeänä ja helppolukuisena. Silloin puhutaankin ”spagettikoodista” (kuva 41) sen visuaalisen ulkomuodon takia. Mitä enemmän koodissa on solmukohtia, sitä hitaampaa suoritus on. [24.]



Kuva 41. Esimerkki spagetti-Blueprintistä [2.]

Suurin suorituskyvyn ongelma Blueprintin puolella on Tick-funktio, jota kutsutaan joka pelin latauskerralla eli framella. Tick-funktio on Unreal-pelimoottorin sydän, ja se mahdollistaa koodin suorittamisen joka framella ja se suoritetaan monta kertaa sekunnissa. Tick-funktiota kutsutaan ennen kuin peli renderöi tulevan framen. Kaikki asiat, jotka tehdään Tick-funktiossa, tulisi olla

suorituskyvyltään nopeaa, koska sillä on merkitystä koko pelin toimivuuteen. [25.] Tick-funktio on paljon hitaampi suorittaa Blueprintissä kuin natiivissa C++-koodissa, jos se sisältää useita laskutoimituksia. Jokainen solmukohta, joka luodaan Blueprintin Tick-funktioon, huonontaa sen suorituskykyä. Blueprintin Tick-funktiota tulisikin välttää mahdollisuuksien mukaan ja käyttää mieluummin ajastettuja kutsuja tai Unrealin delegaatteja. [23.]

Tehokkuutta voidaan ajatella laskentatehokkuuden lisäksi myös työntekijöiden tehokkuuden näkökulmasta. Muokattavuudeltaan Blueprintit voittavat C++-koodin vaivattomuudessaan. Peliolioiden testaus ja muokkaus on helppoa tehdä suoraan editorissa ja työntekijöillä kuluu tehtäviensä tekemiseen vähemmän aikaa. Myös prototyyppien teko on nopeaa, kun työn jälki on heti näkyvissä ruudulla. [24.]

3.3 Projekti ja työryhmä

Valinta siitä, kumpaa työtapaa käytetään, täytyy aina tehdä projektikohtaisesti ja tiimin taitojen mukaan. Jos tiimissä on enemmän artisteja ja suunnittelijoita kuin ohjelmoijia, valinta kallistuu todennäköisesti Blueprint-painotteiseen työtapaan. Jos taas on enemmän ohjelmoijia kuin muita tekijöitä, valinta kallistuu enemmän C++:n puolelle. [23.] Jo yhden ohjelmointitaitoisen kehittäjän lisääminen tiimiin parantaa projektin suorituskykyä ja luettavuutta, kun logiikoita saadaan enemmän C++-koodin puolelle.

Vaikka tiimi sisältäisikin pelkkiä ohjelmoijia, ei projektia kannata tehdä pelkästään C++-koodissa, vaikka se onkin mahdollista Unreal-pelimoottorissa. Blueprintien oikeanlainen käyttö C++-koodin jatkeena helpottaa kaikkien tiimin jäsenten toimintaa, kun työskentelyssä ei tuhlaannu aikaa monimutkaisten koodirakenteiden ymmärtämiseen. [24.]

4 Pohdinta

Pelin luominen alusta loppuun on pitkä ja monimutkainen prosessi. Projektin suunnittelu tiimille sopivaksi on tärkeää ja siinä Unreal-pelimoottorin Blueprintit auttavat. Näin kaikki projektiin kuuluvat voivat osallistua pelin logiikoiden tekoon ja kaikkien tiimin jäsenten aika voidaan hyödyntää. Kuinka paljon Blueprintejä käytetään, riippuu tiimin rakenteesta.

Jos mukana on aloittelevia ohjelmoijia, voidaan projekti tehdä Blueprinteihin, mutta siirtää osa logiikasta C++-koodiin. Jos taas tiimi koostuu kokeneista kehittäjistä, voidaan projekti tehdä pääsääntöisesti C++-koodiin ja kutsua Blueprint Eventejä, johon voidaan luoda pelilogiikkaa visuaalisiin tarkoituksiin.

C++-painotteinen versio on ihannetilanne Unreal-pelimoottorin käyttämisestä. Ohjelmoijat saavat keskittyä koodiin, kun taas artisti tai suunnittelija voivat keskittyä visuaaliseen tekemiseen. C++-versiossa perinnöllisyyttä on helppo ja selkeä käyttää, jolloin säästetään aikaa ja koodi on luettavampaa. C++-luokan rakenteet ovat helposti muutettavissa ja muokattavissa halutunlaiseksi logiikaksi. C++-painotteinen versio yhdistää erinomaisesti C++:n suorituskyvyn ja Blueprintien muokattavuuden.

Puhuttaessa Blueprintien heikosta suorituksesta täytyy muistaa, että suorituskyky ei ole kaikki kaikessa, jos projektia ei saada valmiiksi osaamattomuuden vuoksi. Jos puolet tiimistä odottaa asian lisäämistä C++-koodiin, silloin työtavan valinta on ollut väärä suhteessa tiimin kokoonpanoon. Voikin olla tiimille hyvä tehdä projekti kokonaan Blueprinteihin ja lisätä suorituskykyä siirtämällä raskaimmat logiikat C++-koodin puolelle.

Projektia ei kuitenkaan kannata myöskään tehdä täysin C++-koodiin, vaikka siihen osaaminen löytyisi. Se hidastaa työskentelyä ja tekee pelin muokattavuudesta hankalaa myös ohjelmoijille. Jokaisessa projektissa kannattaisi siis yhdistää sekä C++ että Blueprintit projektiin soveltuvalla tavalla. Unreal-pelimoottorin tekijöiden tarkoituksena on ollut, että Blueprinteillä saadaan vain lisäarvoa projektille eikä se, että Blueprinteillä luotaisiin kokonaisia pelejä. Tätä ominaisuutta kannattaa siis hyödyntää.

Struktien käyttö on C++-koodissa tärkeää, mutta yhteiskäytössä Blueprintien kanssa struktit saavat uudenlaisen arvon. Strukteja käytettäessä yhdessä Blueprintien kanssa saadaan Blueprintejä hyödynnettyä monipuolisemmin ja lisättyä projektin muokattavuuden tasoa.

Tämän opinnäytetyön tarkoituksena oli selvittää, miten käyttää C++-koodia ja Blueprintejä yhdessä. Selkeää vastausta tähän kysymykseen ei ole, vaan kysymykseen pitää aina vastata tiimin sisällä, sillä jokainen projekti ja tiimi on erilainen. Työn tuloksena voidaan todeta, että projektia ei kannata tehdä pelkästään joko C++-koodilla tai Blueprinteillä. C++:n ja Blueprintien tehokas yhteiskäyttö vaatii laajan tietämyksen Unreal-pelimoottorista ja sen toiminnasta.

Lähteet

(1) Joanna Lee, John P. Doran, Nitish Misra. Unreal Engine: Game Development from A to Z. https://books.google.fi/books?id=bKbWDQAAQBAJ&printsec=frontcover&dq=Unreal+engine&hl=fi&sa=X&ved=0ahUKEwiWurOJ6ObkAhWEwcQBHf7pD9gQ6AEIaDAH#v=onepage&q=Unreal%20engine&f=false.

(2) Tim Sweeney. Unreal Engine is Now Free! Available at: https://www.unrealengine.com/en-US/blog/ue4-is-free. Accessed Nov 1, 2019.

(3) Epic Games. Introduction to Blueprints. Available at: https://docs.unrealengine.com/en-US/Engine/Blueprints/GettingStarted/index.html. Accessed Nov 1, 2019.

(4) Epic Games. Introduction to C++ Programming in UE4. Available at: https://docs.unrealengine.com/en-US/Programming/Introduction/index.html. Accessed Nov 1, 2019.

(5) What is C++ Programming? SoftwareEngineerInsider.com. Available at: https://www.softwareengineerinsider.com/programming-languages/cplusplus.html. Accessed Nov 1, 2019.

(6) Classes. cplusplus.com . Available at: http://www.cplusplus.com/doc/tutorial/classes/. Accessed Nov 1, 2019.

(7) Epic Games. C++ and Blueprints. Available at: https://docs.unrealengine.com/en-US/Gameplay/ClassCreation/CodeAndBlueprints/index.html. Accessed Nov 1, 2019.

(8) Epic Games. Blueprint Overview. Available at: https://docs.unrealengine.com/en-US/Engine/Blueprints/Overview/index.html. Accessed Nov 1, 2019.

(9) Epic Games. Components Window. Available at: https://docs.unrealengine.com/en-US/Engine/Blueprints/UserGuide/Components/index.html. Accessed Nov 1, 2019.

(10) Epic Games. Graph Editor Tab. Available at: https://docs.unrealengine.com/en-US/Engine/Blueprints/Editor/UIComponents/GraphEditor/index.html. Accessed Nov 1, 2019.

(11) Epic Games. Details Panel. Available at: https://docs.unrealengine.com/en-US/Engine/Blueprints/Editor/UIComponents/Details/index.html. Accessed Nov 1, 2019.

(12) Epic Games. Details Panel. Available at: https://docs.unrealengine.com/en-US/Engine/UI/LevelEditor/Details/index.html. Accessed Nov 1, 2019.

(13) Michael Noland. Unreal Property System (Reflection). Available at: https://www.unrealengine.com/en-US/blog/unreal-property-system-reflection. Accessed Nov 1, 2019.

(14) What is a Macro? - Definition from Techopedia. Available at: https://www.techopedia.com/definition/3833/macro. Accessed Nov 1, 2019.

(15) Epic Games. Objects. Available at: https://docs.unrealengine.com/en-US/Programming/UnrealArchitecture/Objects/index.html. Accessed Nov 1, 2019.

(16) Epic Games. Class Specifiers. Available at: https://docs.unrealengine.com/en-US/Programming/UnrealArchitecture/Reference/Classes/Specifiers/index.html. Accessed Nov 1, 2019.

(17) Rama. Structs, USTRUCTS(), They're Awesome. Available at: https://wiki.unrealengine.com/Structs,_USTRUCTS(),_They're_Awesome. Accessed 1.11., 2019.

(18) Epic Games. Struct Specifiers. Available at: https://docs.unrealengine.com/en-US/Programming/UnrealArchitecture/Reference/Structs/Specifiers/index.html. Accessed Nov 1, 2019.

(19) Epic Games. Property Specifiers. Available at: https://docs.unrealengine.com/en-US/Programming/UnrealArchitecture/Reference/Properties/Specifiers/index.html. Accessed Nov 1, 2019.

(20) Epic Games. Function Specifiers. Available at: https://docs.unrealengine.com/en-US/Programming/UnrealArchitecture/Reference/Functions/Specifiers/index.html. Accessed Nov 1, 2019.

(21) Tom Looman. UE4 UFUNCTION Keywords Explained. 2016; Available at: https://www.tomlooman.com/ue4-ufunction-keywords-explained/. Accessed Nov 1, 2019.

(22) Epic Games. Events. Available at: https://docs.unrealengine.com/en-US/Engine/Blueprints/UserGuide/Events/index.html. Accessed Nov 1, 2019.

(23) Epic Games. Balancing Blueprint and C++. Available at: https://docs.unrealengine.com/en-US/Resources/SampleGames/ARPG/BalancingBlueprintAndCPP/index.html. Accessed Nov 1, 2019.

(24) Epic Games. Guidelines for Programming for Blueprints. Available at: https://docs.unrealengine.com/en-US/Engine/Blueprints/TechnicalGuide/Guidelines/index.html. Accessed Nov 1, 2019.

(25) Bob Cober. Unreal - Tick Functions, Delta Time, and the Task Graph. 2018; Available at: http://www.recursiveblueprints.fun/unreal-delta-seconds/. Accessed Nov 1, 2019.

Kuvat

(1) Epic Games. "Spaghetti code". Available at: https://i.ytimg.com/vi/WTRmZ6aFEWM/maxresdefault.jpg. Accessed Nov 1, 2019.

(2) Epic Games. Unreal . Available at: https://blueprintsfromhell.tublr.com/image/180629811281. Accessed Nov 1, 2019.