TAMPEREEN
AMMATTIKORKEAKOULU

TUTUSTUMINEN UNREAL ENGINE 4:ÄÄN

Yksinkertaisen FPS-pelin toteuttaminen

Asko Suvanto

Opinnäytetyö
Toukokuu 2017
Tietojenkäsittely
Pelituotanto

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittely
Pelituotanto

SUVANTO, ASKO:
Tutustuminen Unreal Engine 4:ään
Yksinkertaisen FPS-pelin toteuttaminen

Opinnäytetyö 68 sivua
Toukokuu 2017

Unreal Engine 4 on monipuolinen pelinkehitysalusta ja se on täysin ilmainen. Unreal Engine 4 -pelinkehitysalustalla voi tehdä pelejä, vaikka ei osaisi koodata. Opinnäytetyö esittelee Unreal Engine 4 -pelinkehitysalustan ja sen työkaluja.

Opinnäytetyön tarkoituksena on ohjeistaa, miten tehdään yksinkertainen FPS-peli Unreal Engine 4 -pelinkehitysalustan työkaluja ja C++-ohjelmointia käyttäen. Opinnäytetyö ohjeistaa FPS-pelin kehitystä käytännön esimerkkien kautta. Ohjeistus kattaa FPS-pelin kehityksen projektin luomisesta aina valmiiseen, standalone-versioon asti. Ohjeistuksessa FPS-peliä kehitetään C++-ohjelmistokielen avulla, ja siksi vaatimuksena on C++-ohjelmointikielen osaaminen. Opinnäytetyö tarjoaa kaiken tarvittavan C++-koodin ohjeistuksen mukana.

Opinnäytetyön ohjeistusta seuraamalla lopputulokseksi saadaan täysin toimiva FPS-pelin pohja. Tämä valmis pohja tarjoaa käytännön esimerkkejä siitä, miten FPS-peli voidaan toteuttaa Unreal Engine 4 -pelinkehitysalustaa käyttämällä. Opinnäytetyö on suunnattu henkilöille, jotka ovat kiinnostuneet Unreal Engine 4 -pelinkehitysalustasta ja pelinkehityksestä. Opinnäytetyö toivottavasti innostaa jatkamaan pelinkehitystä ja etsimään lisää tietoa Unreal Engine 4 -pelinkehitysalustan mahdollisuuksista.

Asiasanat: fps-peli, ohje, ue4, unreal engine

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in Business Information Systems
Option of Game Development

SUVANTO ASKO:
Introduction to Unreal Engine 4
Developing a Simple FPS Game

Bachelor's thesis 68 pages
May 2017

Unreal Engine 4 is a powerful game development platform and it is completely free. Developing games can be done on Unreal Engine 4 without even knowing how to code. This thesis serves as an introduction to Unreal Engine 4 game development platform.

This thesis guides the reader through the development of a simple FPS game using Unreal Engine 4 development tools and C++ programming. Previous knowledge of C++ programming is required in this case, even though all the C++ code needed for the game is supplied here. Following the guide in this thesis will create a fully functional base for an FPS game.

This thesis is intended for those who have an interest in Unreal Engine 4 and game development. Hopefully this thesis will spark an interest to continue game development as well as to seek out more information about the capabilities of Unreal Engine 4.

Key words: fps game, tutorial, ue4, unreal engine

SISÄLLYS

1 JOHDANTO ... 8
1.1 Aiheen tausta ... 8
1.2 Aiheen rajaus ... 8
2 UNREAL ENGINE 4 ... 10
2.1 Epic Games Launcher ... 10
2.2 Unreal Editor ... 11
2.3 Unreal Editor ja C++ ... 13
3 PELIN KEHITTÄMINEN ... 14
3.1 Pelialueen luominen ... 14
3.1.1 Pelialueen muokkaaminen ... 14
3.1.2 Objektien lisääminen pelimaailmaan C++-luokista ... 14
3.1.3 C++-luokasta Blueprintiksi ... 17
3.1.4 Health- ja ammuslaatikoiden C++-luokat ... 21
3.1.5 Health- ja ammuslaatikoiden Blueprint:it ... 25
3.2 Pelaajan liikkuminen pelimaailmassa ... 28
3.3 Interaktiivinen pelimaailma ... 34
3.3.1 Objektien luominen dynaamisesti ... 34
3.3.2 Vihollisen C++-luokan lisääminen ... 40
3.3.3 Vihollisen C++-luokasta Blueprintiksi ... 46
3.3.4 Pelaajan hahmon ja vihollisen välinen interaktiivisuus ... 48
3.4 Efektien lisääminen ... 50
3.5 Palaute pelaajalle ... 53
3.5.1 HUD:in lisääminen ... 53
3.5.2 Valikkojen lisääminen ... 58
3.5.3 Voittoehtojen lisääminen ... 60
4 PELIN VIIMEISTELY ... 66
5 POHDINTA ... 67
LÄHTEET ... 68

LYHENTEET JA TERMIT

3D viewport	3D viewport on Unreal Editorin työkalu, joka näyttää pelimaailman kolmiulotteisena.
Blueprint	Blueprint on UE4:n keskeinen elementti, joka voi koostua esim. 3D-mallista ja/tai C++-skriptistä.
Construction Script	Construction Script on Blueprintin konstruktori.
Content Browser	Unreal Editorin työkalu, jolla hallitaan projektin tiedostoja.
Details panel	Details-paneeli on Unreal Editorin työkalu, jolla voi muokata yksittäisen objektin tietoja.
Dynamic Material	Erikoispinnoite, joka voi muuttua reaaliaikaisesti pelissä.
Event Graph	Event Graph on Blueprintin koodialue. Siellä voi lisätä toiminnallisuutta Blueprinttiin käyttämällä valmiita skriptielementtejä.
FPS (First Person Shooter)	First Person Shooter on ensimmäisen persoonan ammuntapeli.

HUD (Heads Up Display)	Graafinen käyttöliittymä. Pelissä se tarkoittaa näyttöä, joka tarjoaa pelaajalle tietoja pelihahmosta ja -ympäristöstä.
Material	Pinnoite 3D-mallille. Voi koostua yhdestä tai useammasta tekstuurista.
Mesh	3D-mallin ulkomuoto.
Modes panel	Modes-paneeli on Unreal Editorin työkalu, jolla muokataan pelialuetta.
Pelimoottori (eng. Game Engine)	Pelimoottori on ohjelmistokehys, johon on jo valmiiksi koodattu esim. grafiikan, äänen tai syötteiden toiminnollisuus.
Template	Mallipohja.
UE4 (Unreal Engine 4)	Unreal Engine 4 on neljännen sukupolven pelimoottori, jonka on kehittänyt Epic Games.
UMG (Unreal Motion Graphics)	Työkalu, jolla Heads Up Display rakennetaan.
Unreal Editor	Pelinkehitysalusta, jolla tehdään pelejä Unreal Engine 4:lle.
Visual Studio	Ohjelmankehitysympäristö, jota käytetään C++-tiedostojen muokkaamiseen.

Widget	Blueprint, jolle voi lisätä interaktiivisia Heads Up Display -elementtejä.
World Outliner	World Outliner on Unreal Editorin työkalu, joka listaa kaikki pelimaailmaan lisätyt objektit.

1 JOHDANTO

1.1 Aiheen tausta

Tämän opinnäytetyön tarkoituksena on tutustuttaa Unreal Engine 4 -pelimoottoriin ja sen pelikehittäjille tarjoamiin mahdollisuuksiin. Opinnäytetyön ohjeet on suunnattu UE4:n versioon 4.13. Opinnäytetyössä ei oteta vastuuta siitä, miten nämä ohjeet toimivat muilla UE4:n versioilla.

Tämän opinnäytetyön tavoitteena on auttaa tekemään yksinkertainen FPS-peli UE4:n avulla tutustuttamalla UE4:ään ja sen käyttöön. Ohjeet kohdistuvat C++-ohjelmointikielen käyttöön UE4-pelimoottorissa ja siksi pelin suunnittelijalta vaaditaan ohjelmointiosaamista C++-kielellä. Tämä opinnäytetyö tutustuttaa UE4:ään ja sen käyttöön, sekä ohjaa yksinkertaisen FPS-pelin toteuttamisen UE4:n avulla. Koska UE4 on englanninkielinen, käytetään ohjeessa samoja englannin kielen sanoja, vaikka niille olisikin suomennos, jotta ohje pysyy mahdollisimman selkeänä.

UE4:n ohella tässä opinnäytetyössä käytetään Microsoft Visual Studio 2015 -ohjelmaa, jolla tehdään C++-skriptit. C++-skriptejä voi tehdä muillakin ohjelmilla, mutta opinnäytetyön ohjeet opastavat vain Visual Studion käyttöön.

1.2 Aiheen rajaus

Tämän opinnäytetyön ohjeet käsittelevät seuraavat asiat:

- UE4-pelimoottoriin tutustuminen
- Pelialueen muokkaaminen ja staattisten objektien lisääminen
- Pelaajan hahmon muokkaaminen
- Dynaamisten objektien luominen
- Efektien lisääminen
- Valikkojen ja HUD:in lisääminen

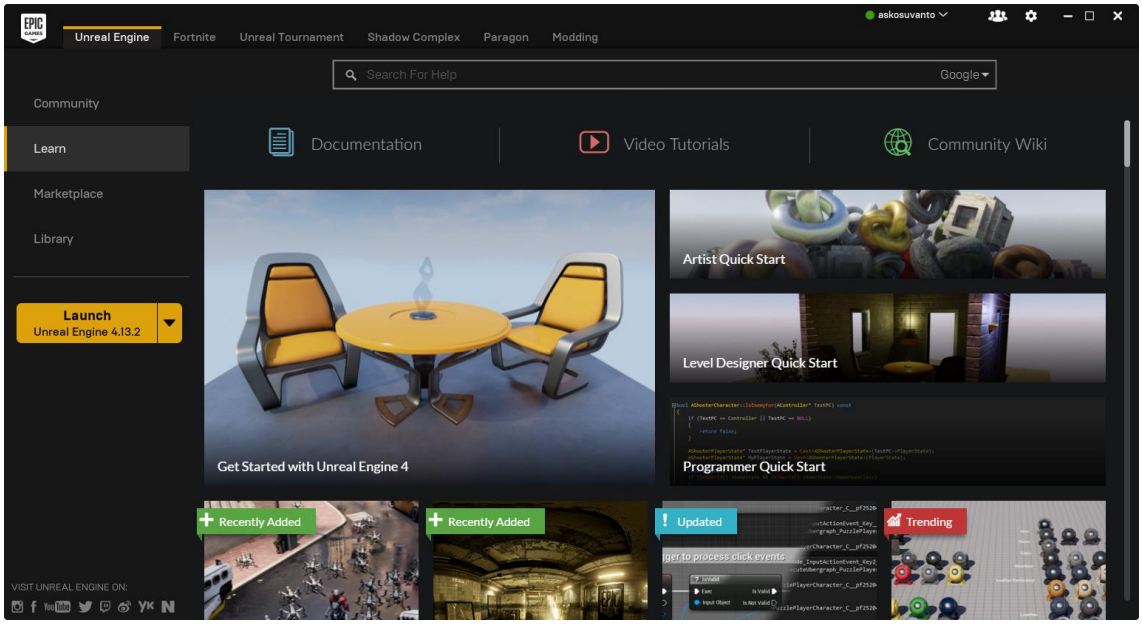
Ohjeita seuraamalla saadaan rakennettua valmis pohja yksinkertaiselle First Person Shooter -pelille, joka toivon mukaan innostaa etsimään lisää tietoa UE4:n mahdollisuuksista ja jatkamaan aloitetun pelin kehitystä.

2 UNREAL ENGINE 4

2.1 Epic Games Launcher

Epic Games Launcher on ohjelma, jolla pelinkehittäjä voi hallita UE4:llä tehtyjä projekteja ja Unreal Engine 4 -pelimoottorin eri versioita. Epic Games Launcher myös tarjoaa Unreal Engine -kehittäjäyhteisön, UE4 -kaupan ja kokoelman neuvoja ja ohjeita pelinkehitykseen.

Pelinkehitykseen tärkeimmät kohdat löytyvät Epic Games Launcherin, Unreal Engine -osiosta (kuva 1). Learn-kohtaan on koottu oppaita ja neuvoja pelinkehitykseen. Nämä oppaat ovat lähes kaikki englanniksi. Marketplace-kohdassa on kauppa, josta voi ostaa valmista sisältöä käytettäväksi omaan pelinkehitykseen. Library-kohdassa ovat saatavilla asennetut Unreal Engine -versiot sekä omat projektit.

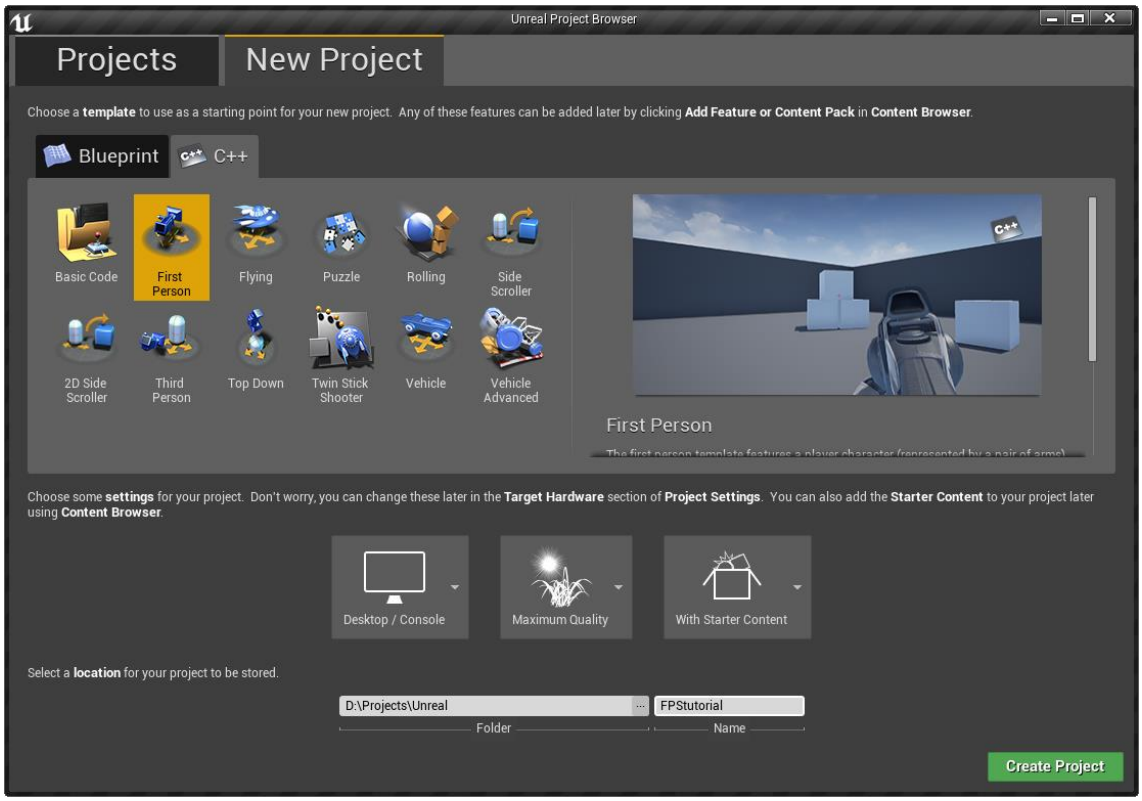


Kuva 1. Epic Games Launcher.

Käynnistä Unreal Editor Launch-nappia painamalla.

2.2 Unreal Editor

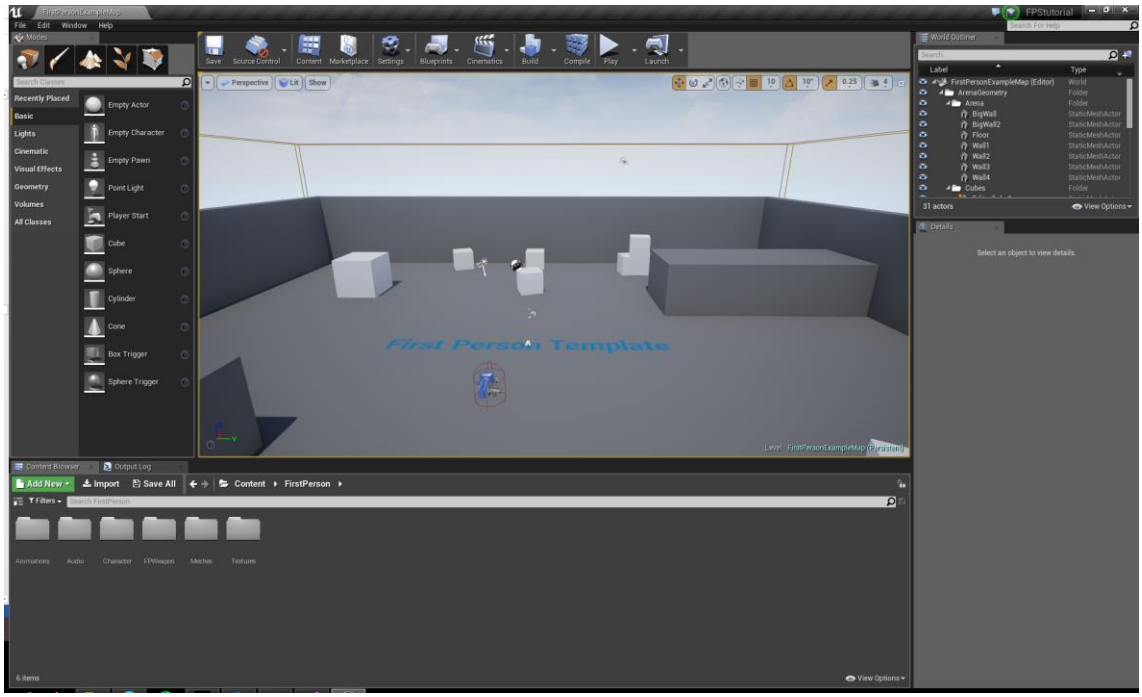
Unreal Editor on ohjelma, jolla varsinainen pelinkehitys tapahtuu. Aluksi on valittava projekti, jota Unreal Editorilla halutaan tehdä, tai valita uuden projektin tekeminen. Valitaan uusi projekti. FPS-pelin kehittämiseen löytyy valmis template, joka helpottaa pelinkehityksen aloittamista. Käytetään tämän pelin kehittämiseen C++:aa, joten valitse C++ First Person Template, kuten kuvassa 2 näkyy. Huomaa, että uuden projektin asetuksien kohdassa lukee: With Starter Content. Tämä lisää projektiin mm. tekstuureja ja efektejä, joita käytetään tässä ohjeessa.



Kuva 2. Valitaan uusi projekti Unreal Editorille.

Enää tarvitsee valita paikka minne projekti luodaan ja mikä sen nimeksi tulee. Huomaa, että projektin nimi voi olla mikä tahansa, mutta sitä käytetään C++-ohjelmoinnissa. Jos nimi on muu kuin FPStutorial, muista muuttaa C++-koodia tarpeen mukaisesti. Sitten vain paina Create Project -nappia. Tämä luo projektin ja lataa Unreal Editorin, sekä käynnistää Visual Studion.

Unreal Editor avautuu Kuvan 3 mukaiseen ikkunaan. Sen voi jakaa viiteen osaan. Vasemmalla on Modes-paneeli, jonka työkaluilla voi muokata ympäristöä, sekä lisätä objekteja pelimaailmaan. Ylhäällä on Toolbar, josta löytyy Unreal Editorin tärkeimmät toiminnot, kuten Save, Compile ja Play. Keskellä on 3D viewport, josta näkee ja voi muokata pelimaailmaa. Alhaalla on Content Browser, jolla voi lisätä ja muokata projektin tiedostoja. Alhaalla on myös Output Log, josta näkee pelissä tulostettavat tekstit, sekä mahdolliset virheilmoitukset. Oikealla on World Outliner, josta näkee kaikki pelimaailmaan lisätyt objektit. Oikealla on myös Details-paneeli, josta voi tarkastella ja muokata valittujen objektien tietoja.



Kuva 3. Unreal Editor.

Tiedostojen hallintaa helpottamaan voi laittaa Sources Paneelin. Se löytyy Content Browserista. Add New -napin alla on Filters-nappi ja sen vasemmalla puolella on Sources Panel -nappi.

Voit myös testata tätä projektia. Ylhäältä Play-nappi käynnistää pelisession. Painamalla 3D viewportissa, saa hiiriohjauksen ja ampumisen. Liikkuminen tapahtuu nuolinäppäimillä tai WASD napeilla. ESC lopettaa pelisession. Shift + F1 vapauttaa hiiren pysäyttämättä pelisessiota.

2.3 Unreal Editor ja C++

Unreal Engine käyttää pelimaailman objektien muokkaamiseen Blueprint-luokkaa. Blueprint on monipuolinen luokka, jonka avulla objekteihin voi lisätä toiminnollisuutta joko C++-skripteillä, tai valmiilla Blueprint elementeillä. Blueprinteihin voi myös lisätä komponentteja, kuten 3D-malleja tai ääniä.

Aikaisemmin valittu C++ template tekee projektin tiedostoihin valmiin Visual Studio projektin. Unreal Editor päivittää Visual Studio projektin, kun C++-tiedostoja lisätään. Visual Studio projektin voi myös päivittää manuaalisesti valitsemalla Unreal Enginestä File -> Refresh Visual Studio Project.

C++-tiedostot löytyvät Content Browser:issa C++ Classes -> FPStutorial kansiosta. Visual Studiossa ne löytyvät Solution Explorerissa Games -> FPStutorial -> Source -> FPStutorial kansiosta.

Tiedostoja voi lisätä ja kääntää Visual Studion puolella, mutta suosittelen, että niin ei tehtäisi, vaan kaikki tiedostojen lisääminen tapahtuisi Content Browserin kautta. Muokatun koodin voi kääntää Unreal Editorin puolella painamalla ylhäällä olevaa Compile-nappia.

Itse C++-luokissa on kolme funktiota, jotka löytyvät lähes jokaisesta peliobjektista. Ne ovat C++-luokan konstruktori, BeginPlay ja Tick. Konstruktori-funktio ajetaan pelin käynnistymisessä. BeginPlay-funktio ajetaan silloin, kun kenttää, jossa objekti on, ladataan. Se myös ajetaan silloin, kun objekti luodaan kentälle. Tick-funktio ajetaan kerran joka frame.

C++-luokan header-tiedostoissa funktiolle ja muuttujille voidaan listätä UFUNCTION- tai UPROPERTY-makro, jolloin nämä funktiot ja muuttujat saadaan esille Unreal Editorin puolella. Käydään näitä makroja tarkemmin läpi sitä mukaan, kun niitä tarvitaan.

3 PELIN KEHITTÄMINEN

3.1 Pelialueen luominen

Edellisessä luvussa tehtiin uusi projekti. Projektissa valittu First Person Template loi valmiin pelialueen. Voit vapaasti muokata pelialueesta sellaisen kuin haluat.

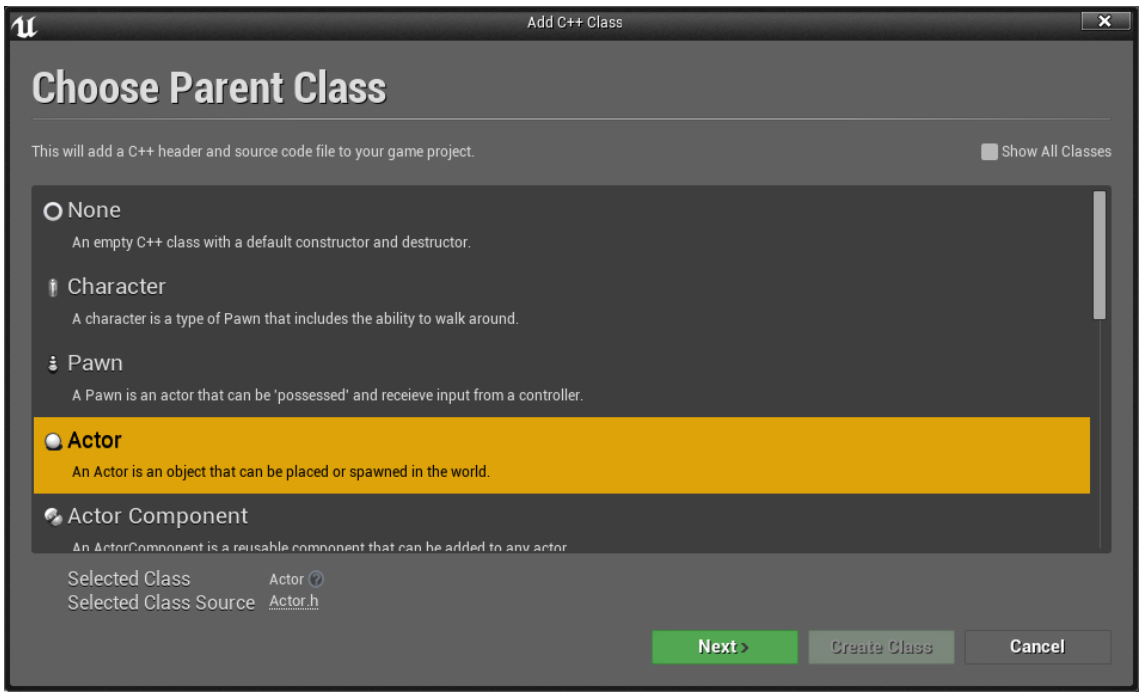
3.1.1 Pelialueen muokkaaminen

Pelialueen muokkaaminen tapahtuu 3D viewportin kautta. Kuvakulmaa voi liikutella WASD- ja nuolinäppäimillä. Hiiren oikea kääntää kuvakulmaa ja vasen valitsee objekteja. Viewportin oikeassa yläkulmassa on työkaluja, joilla objekteja voi liikuttaa, kääntää tai muuttaa kokoa. Työkaluista voi myös valita kuinka suurilla harppauksilla esim. laatikkoa voi liikuttaa.

Käteviä pikanäppäimiä ovat: Copy (ctrl + c), Paste (ctrl + v), Duplicate (ctrl + w), Snap to Floor (end), Delete Object (delete), Focus Selected (f), sekä Save (ctrl + s).

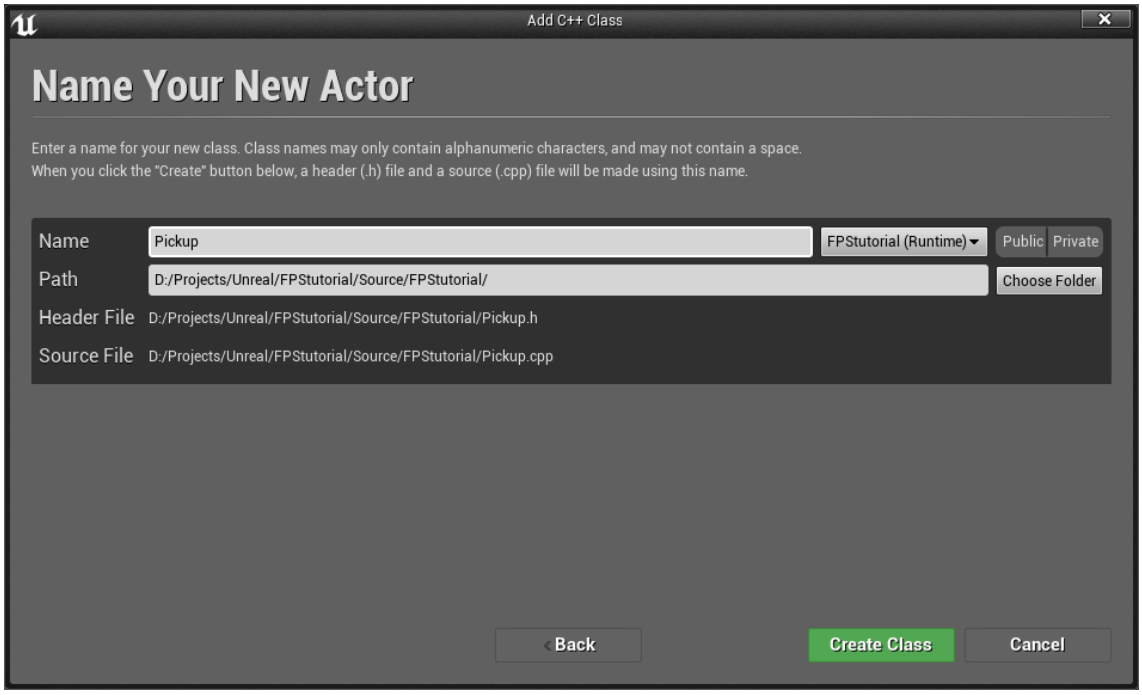
3.1.2 Objektien lisääminen pelimaailmaan C++-luokista

FPS-pelissähän pitää olla health- ja ammuslaatikoita. Sen sijaan, että aloitettaisiin lisäämällä ensin objekti ja liittämällä siihen skripti, aloitetaankin tekemällä skripti. Avaa Content Browserista C++ Classes -> FPStutorial kansio. Sitten Add New napista (tai hiiren oikealla) valitse New C++ Class, jolloin avautuu kuvan 4 mallinen ikkuna. Valitse kantaluokaksi Actor. Actor luokalla ei ole erikoisominaisuuksia ja se on sopiva tähän tarkoitukseen.



Kuva 4. Uuden C++-luokan tekeminen.

Next-nappi avaa kuvan 5 näköisen ikkunan. Uuden C++-luokan nimi ei tarvitse olla Pickup, mutta tätä luokan nimeä käytetään tässä ohjeessa.



Kuva 5. Uuden C++-luokan nimeäminen.

Nyt meillä on uusi luokka. Jotta saataisiin jotakin näkymään pelimaailmassa, niin lisätään siihen StaticMesh-komponentti, eli staattinen 3D-malli. StaticMesh-komponentin lisäämisen Pickup.h:hon onnistuu kirjoittamalla kohtaan:

private:
class UStaticMeshComponent PickupMesh;*

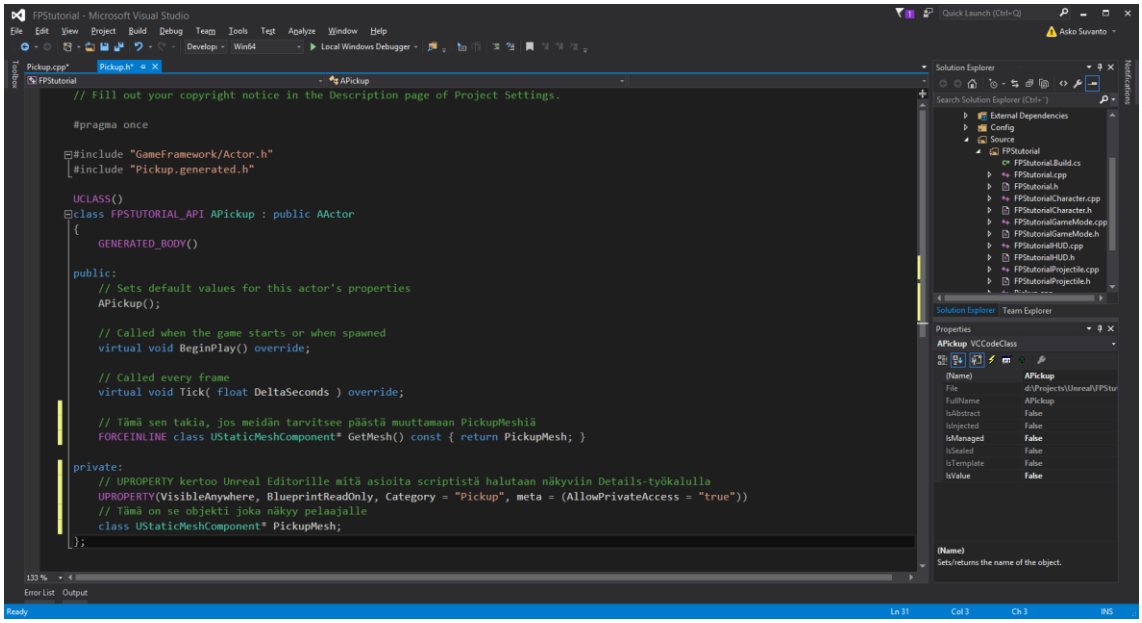
Jotta StaticMesh-komponentin saa näkyviin Unreal Editorissa, tarvitsee ennen yllä olevaa riviä lisätä:

UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Pickup", meta = (AllowPrivateAccess = "true"))

UPROPERTY-makro kertoo Unreal Enginelle, että StaticMesh-komponentti tulee näyttää Blueprintin tiedoissa, mutta sitä ei voi muokata Unreal Editorin puolella. UPROPERTY-makro toimii vain muuttujille. Lisätään vielä funktio, joka palauttaa vain PickupMeshin:

public:
FORCEINLINE class UStaticMeshComponent GetMesh() const { return PickupMesh; }*

Kuvassa 6 näkyy Pickup.h:hon lisätty koodi kokonaisuudessaan.



Kuva 6. Lisätään StaticMesh Pickup.h:hon.

PickupMeshin alustaminen tapahtuu Pickup.cpp:ssä kirjoittamalla:

PickupMesh = CreateDefaultSubobject<UStaticMeshComponent>(TEXT("PickupMesh"));

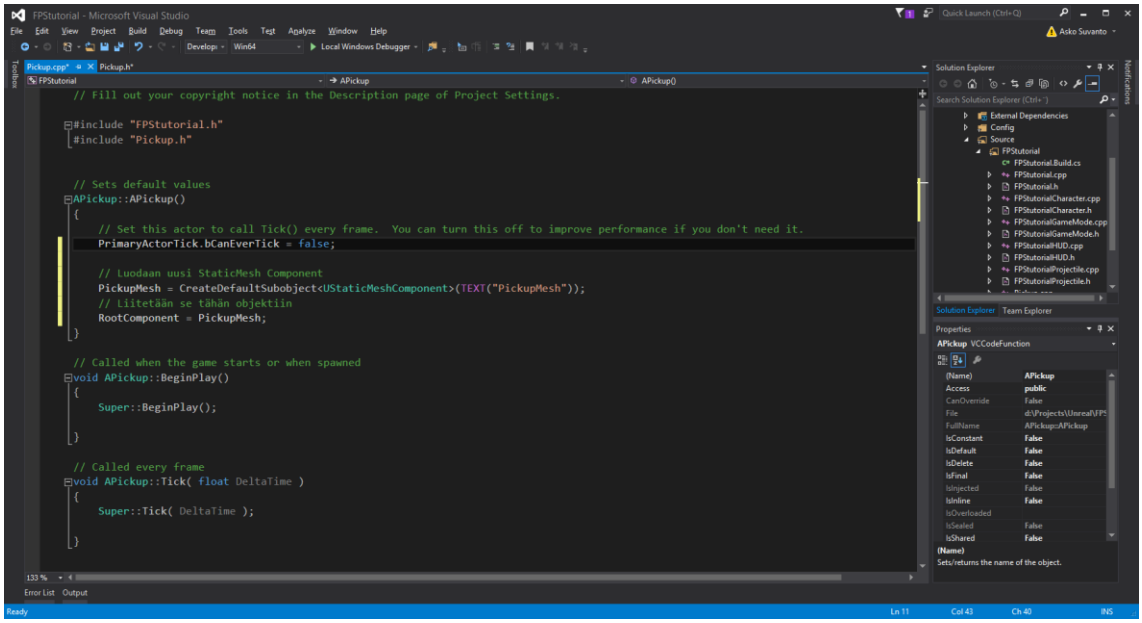
Sitten liitetään PickupMesh Pickupiin rivillä:

RootComponent = PickupMesh;

Lopuksi muutetaan:

PrimaryActorTick.bCanEverTick = false;

Koska tämän objektin ei tarvitse vaikuttaa pelimaailmaan joka frame. Kuvassa 7 näkyy Pickup.cpp:hen lisätty koodi kokonaisuudessaan.

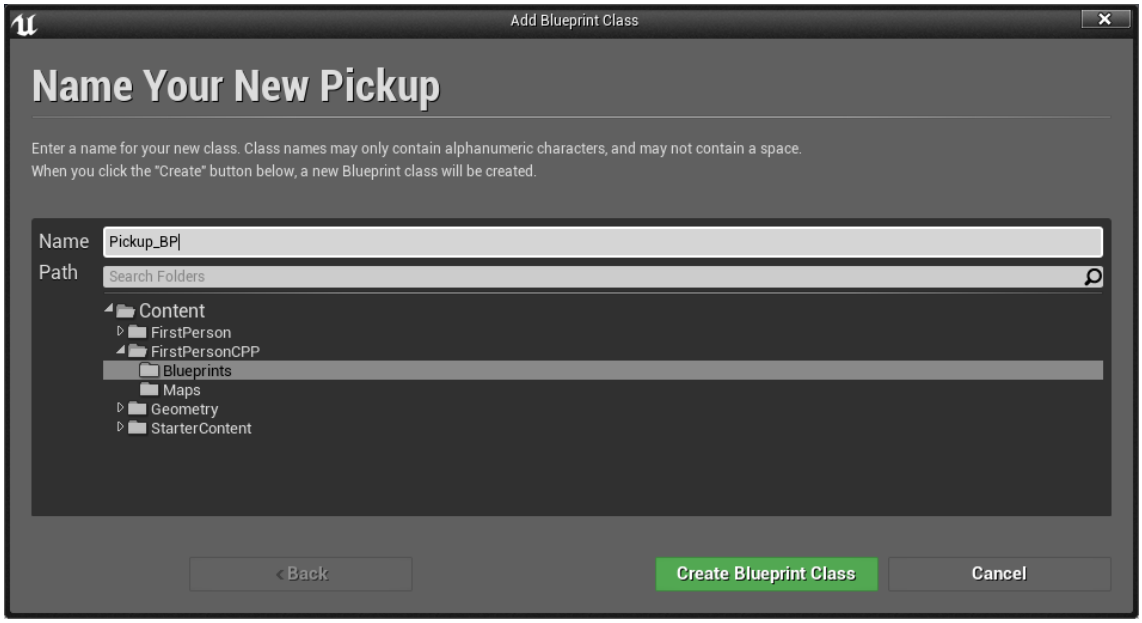


Kuva 7. Lisätään StaticMesh Pickup.cpp:hen.

Tallenna tehdyt muutokset. Unreal Editorin puolella paina Compile-nappia. Nyt meillä on Unreal Enginen mukainen C++-luokka ja seuraavaksi teemme siitä pelimaailmaan lisättävän Blueprintin.

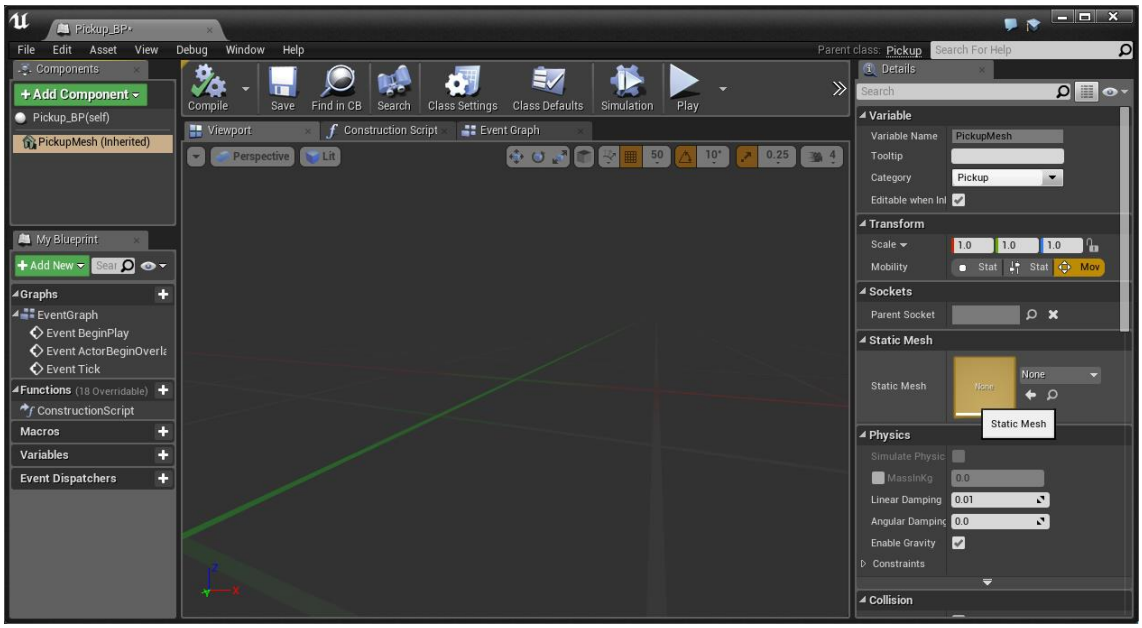
3.1.3 C++-luokasta Blueprintiksi

Pickup on vielä pelkkä C++-luokka, joten nyt siitä on tehtävä Blueprint. Paina hiiren oikealla napilla Pickup-skriptin päällä ja valitse Create Blueprint class based on Pickup. Se avaa kuvan 8 mallisen ikkunan. Laita nimeksi Pickup_BP ja tarkista, että uusi Blueprint tallentuu paikkaan, josta sen löydät helposti.



Kuva 8. Blueprintin teko skriptistä.

Painamalla Create Blueprint Class -nappia avautuu kuvan 9 mukainen ikkuna. Jos ikkuna ei näytä kuvan 9 mukaiselta, pitäisi heti työkalupalkin alla olla linkki: Open Full Blueprint Editor. Nyt vasemmalla näkyy skriptissä tehty PickupMesh-komponentti. Kun se on valittuna, oikealla puolella näkyy Static Mesh -kohdassa None tällä hetkellä. Siihen voit laittaa minkä tahansa listasta haluat, itse valitsen 1M_Cube.



Kuva 9. Pickup-luokasta tehty Blueprint.

Ennen kuin muutokset voi tallentaa, täytyy painaa Compile-nappia (pikanäppäin F7). Tallenna Blueprint ja voit sulkea Pickup_BP ikkunan jos haluat. Pickup Blueprintin lisääminen pelimaailmaan onnistuu raahaamalla Pickup_BP Content Browserista 3D viewporttiin.

Nyt meillä on Pickupiin lisätty 3D-malli, jotta pelaaja näkee sen pelimaailmassa. Se ei kuitenkaan tee vielä mitään. Seuraavaksi lisätään siihen toiminnallisuutta. Ensiksi lisätään yksinkertainen muuttuja, joka kertoo onko tämä Pickup aktiivinen:

```
protected:
bool bIsActive;
```

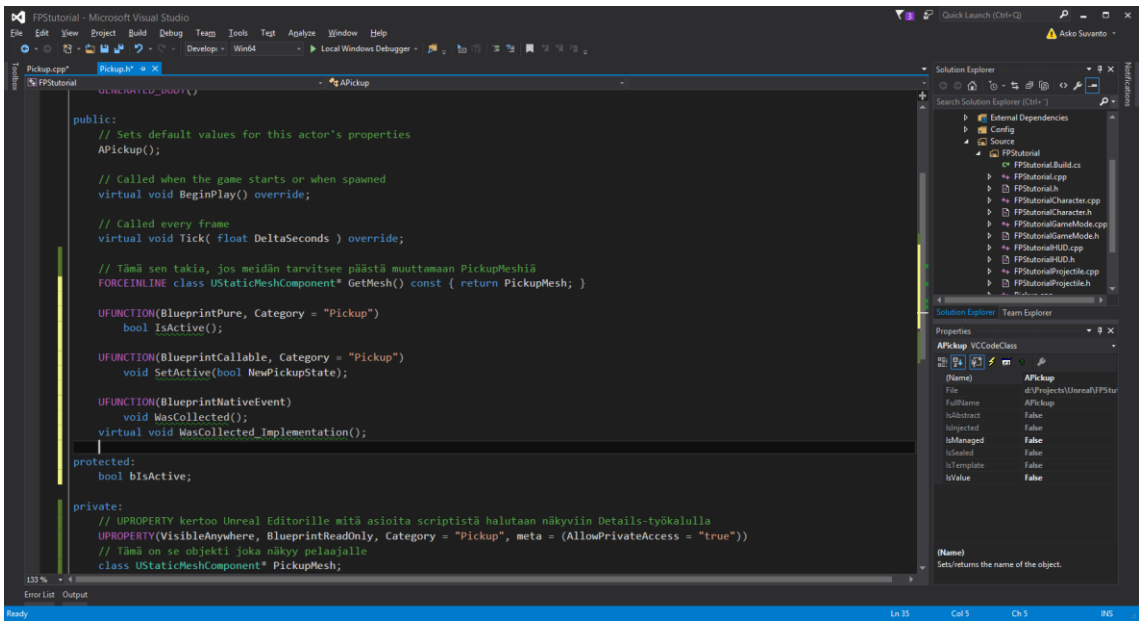
Ja funktiot, jotka palauttavat ja muokkaavat tätä muuttujaa:

```
public:
UFUNCTION(BlueprintPure, Category = "Pickup")
bool IsActive();

UFUNCTION(BlueprintCallable, Category = "Pickup")
void SetActive(bool NewPickupState);
```

UFUNCTION-makro toimii kuten UPROPERTY, mutta vain funktioille. BlueprintPure tarkoittaa, että funktio ei voi muokata tämän Blueprintin tietoja. BlueprintCallable tarkoittaa, että tätä funktiota voi kutsua Blueprintissä valmiiden Blueprint elementtien kautta. Haluamme, että Pickupin voi kerätä. Itse kerääminen tapahtuu FPStutorialCharacter-skriptissä, mutta jotta Pickupille tapahtuisi jotakin keräämisen jälkeen, lisätään funktio:

```
UFUNCTION(BlueprintNativeEvent)
void WasCollected();
virtual void WasCollected_Implementation();
```

BlueprintNativeEvent tarvitaan, kun funktiossa käytetään _Implementation():ia. Kuvassa 10 näkyy Pickup.h:hon lisätty koodi kokonaisuudessaan.



Kuva 10. Pickup.h täydennettynä.

Lisätään Pickup.cpp:n konstruktoriin lähdeasetukset muuttujille bIsActive ja PickupMeshille, sekä liitetään uusi PickupMesh tähän objektiin:

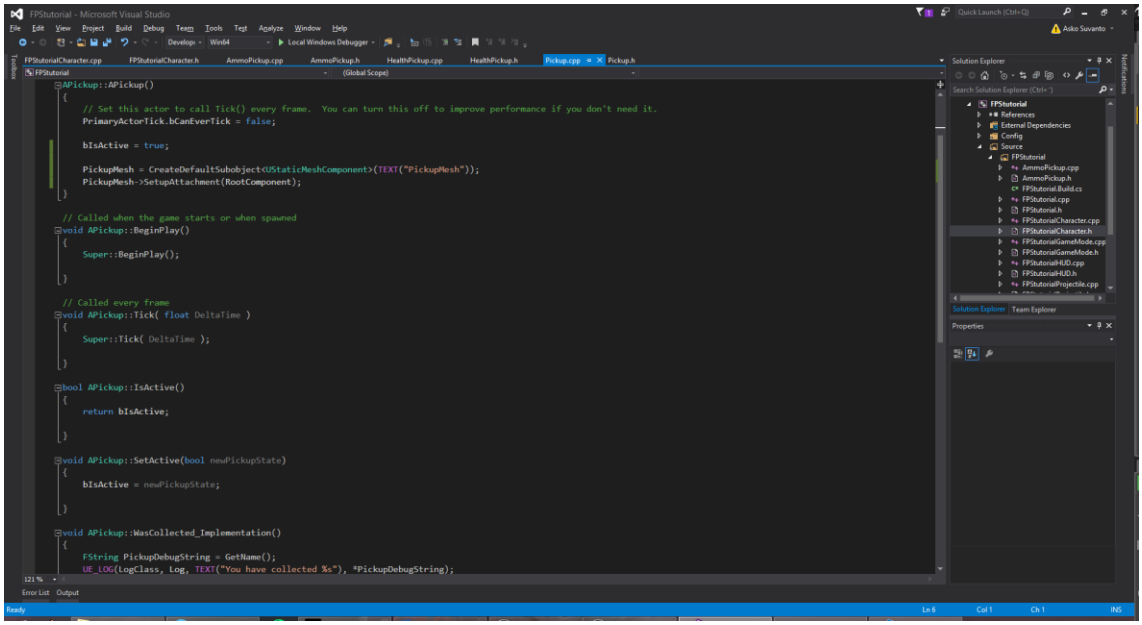
```
// Sets default values
APickup::APickup()
{
    // Set this actor to call Tick() every frame.  You can turn this off to improve
    performance if you don't need it.
    PrimaryActorTick.bCanEverTick = false;

    bIsActive = true;

    PickupMesh = CreateDefaultSubobject<UStaticMeshComponent>(
    TEXT("PickupMesh"));

    PickupMesh->SetupAttachment(RootComponent);
}
```

Seuraavaksi lisätään Pickup.cpp:hen seuraavat funktiot:

```
bool APickup::IsActive()
{
    return bIsActive;
}

void APickup::SetActive(bool newPickupState)
{
    bIsActive = newPickupState;
}

void APickup::WasCollected_Implementation()
{
    FString PickupDebugString = GetName();
    UE_LOG(LogClass, Log, TEXT("You have collected %s"), *PickupDebugString);
```

```
}
```

Huomaa nimiavaruuden erikoisuus ”APickup::”, joka ketoo Unreal Enginelle Pickupin olevan Actor-tyyppinen luokka. UE_LOG on apufunktio, joka tulostaa tekstiä Unreal Editorin Output Logiin. Kuvassa 11 näkyy Pickup.cpp:hen lisätty koodi kokonaisuudessaan.

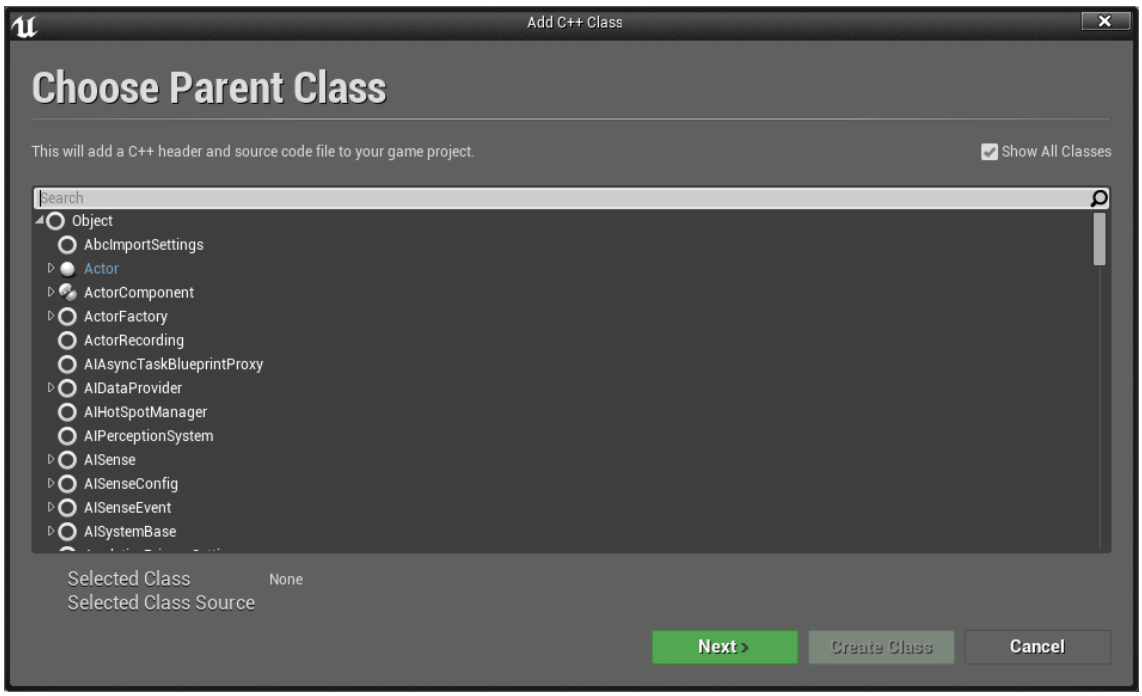


Kuva 11. Pickup.cpp täydennettynä.

Nyt kun meillä on Pickup:in pohjaluokka, meidän tarvitsee seuraavaksi tehdä omat luokat health-laatikoille ja ammuslaatikoille.

3.1.4 Health- ja ammuslaatikoiden C++-luokat

Tehdään health- ja ammuslaatikot periytymään Pickup-luokasta. Tähän on Unreal Editorissa kaksi tapaa. Nopein tapa on Content Browserissa painaa hiiren oikealla napilla Pickup-skriptin päällä ja valita: Create C++ class derived from Pickup. Toinen tapa on painaa Content Browserin Add New -nappia ja valita New C++ Class. Jotta Pickup-luokan saisi valittua, tarvitsee raksittaa ikkunan oikeassa ylänurkassa oleva Show All Classes. Tällöin tulee kuvan 12 mukainen näkymä. Etsintäkenttään kirjoittamalla pickup, löytää Pickup-luokan.

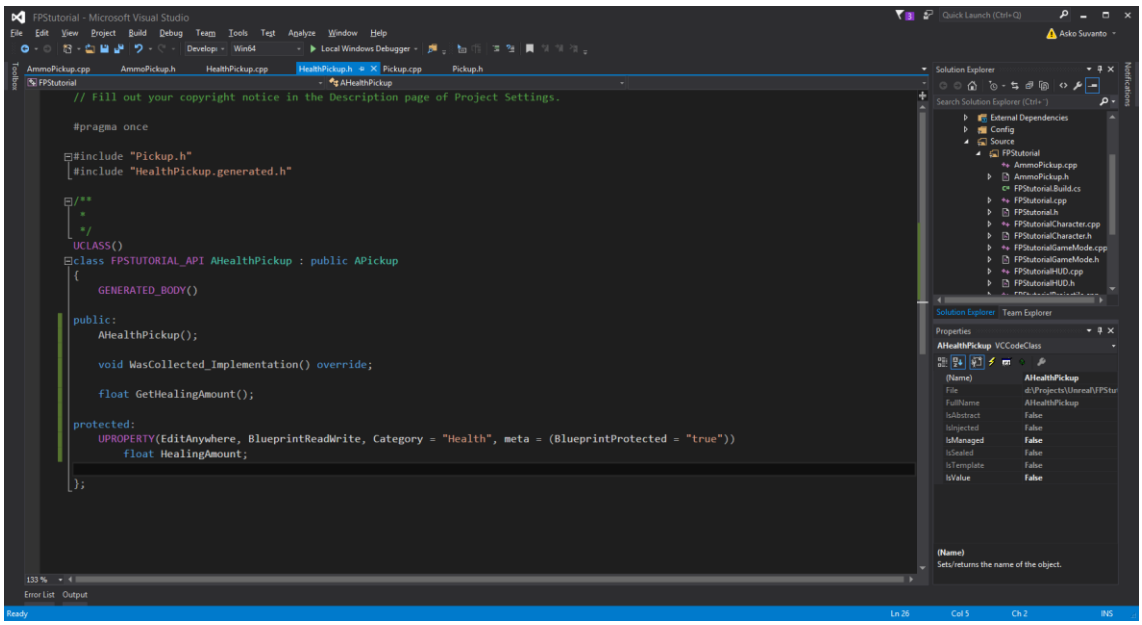


Kuva 12. Kaikki luokat näkyvissä.

Nimetään uudet luokat HealthPickup ja AmmoPickup. Lisätään HealthPickup.h:hon muuttuja, joka kertoo kuinka paljon pelaajan heath paranee yhdestä health-laatikosta:

```
public:
AHealthPickup();

void WasCollected_Implementation() override;

float GetHealingAmount();

protected:
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Health", meta =
(BlueprintProtected = "true"))
float HealingAmount;
```

EditAnywhere ja BlueprintReadWrite antavat luvan muuttaa HealingAmount-muuttujaa Unreal Editorissa. Näin health-laatikon muuttujaa voidaan hienosäätää Unreal Editorissa ilman, että koodia tarvitsee muuttaa. Kuvassa 13 näkyy HealthPickup.h:hon lisätty koodi kokonaisuudessaan.

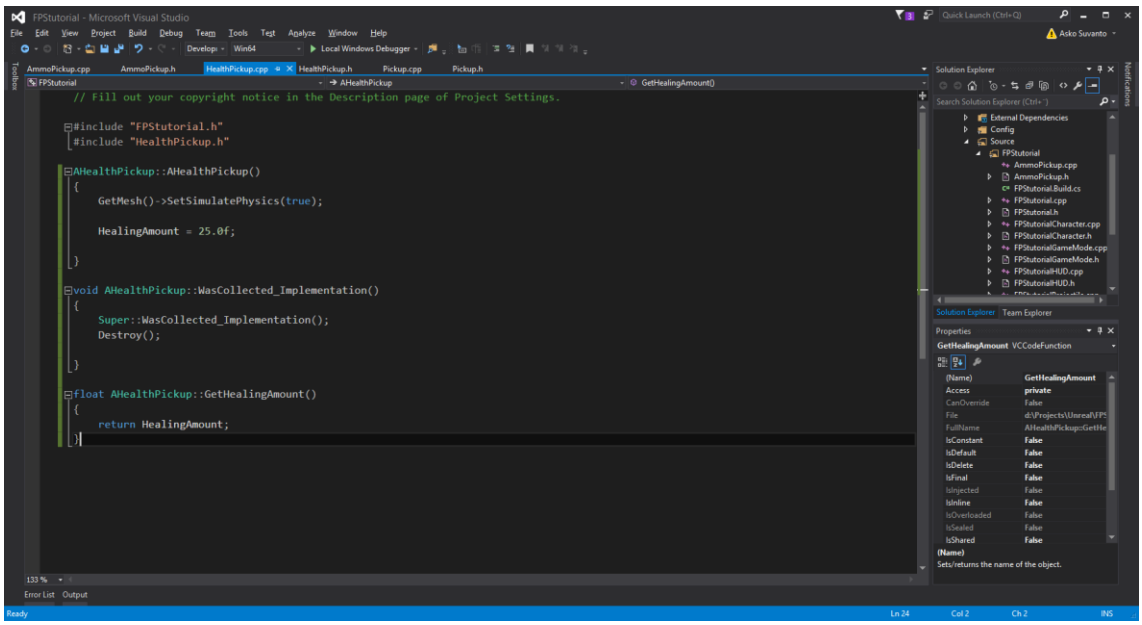


Kuva 13. HealthPickup.h.

HealthPickup.cpp:ssä asetetaan lähdearvo HealingAmount muuttujalle, sekä aktivoidaan fysiikan vaikutus health-laatikkoon. Lisätään funktio, WasCollected, joka tuhoaa health-laatikon, kun se kerätään:

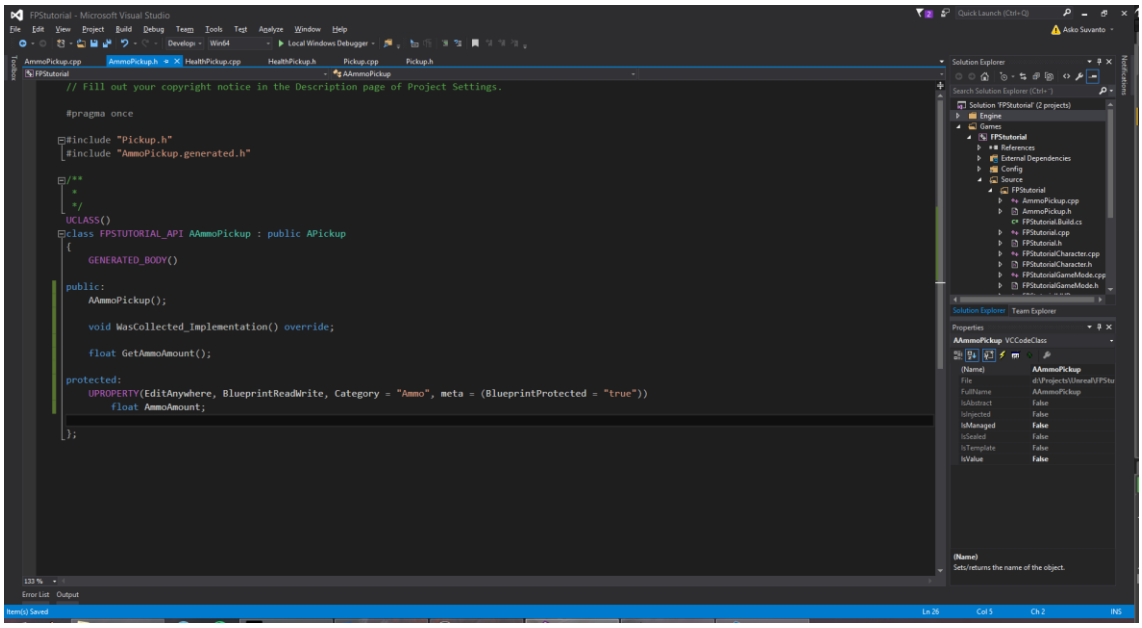
```
AHealthPickup::AHealthPickup()
{
    GetMesh()->SetSimulatePhysics(true);

    HealingAmount = 25.0f;
}

void AHealthPickup::WasCollected_Implementation()
{
    Super::WasCollected_Implementation();
    Destroy();
}

float AHealthPickup::GetHealingAmount()
{
    return HealingAmount;
}
```

Kuvassa 14 näkyy HealthPickup.cpp:hen lisätty koodi kokonaisuudessaan.

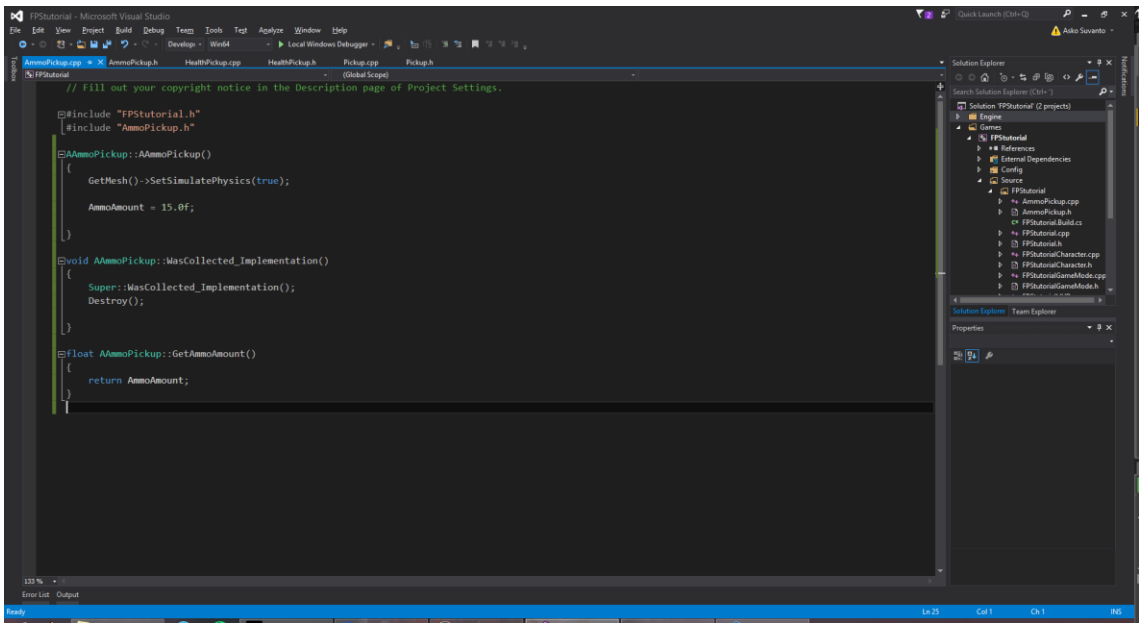


Kuva 14. HealthPickup.cpp.

Nyt kun health-laatikon koodi on valmis, tehdään ammuslaatikon koodi samalla tavalla. Ammuslaatikon koodi on lähestulkoon identtinen health-laatikon koodin kanssa. Muutetaan koodissa muuttuja HealingAmount AmmoAmountiksi, ja sen funktio GetHealingAmount GetAmmoAmountiksi, sekä vaihdetaan muuttujan AmmoAmountin kategoriaksi ”Ammo”. Asetetaan AmmoAmountin lähdearvoksi 15.0f. Kuvissa 15 ja 16 näkyy AmmoPickup.h:hon ja AmmoPickup.cpp:hen lisätyt koodit kokonaisuudessaan.



Kuva 15. AmmoPickup.h:n muutokset.

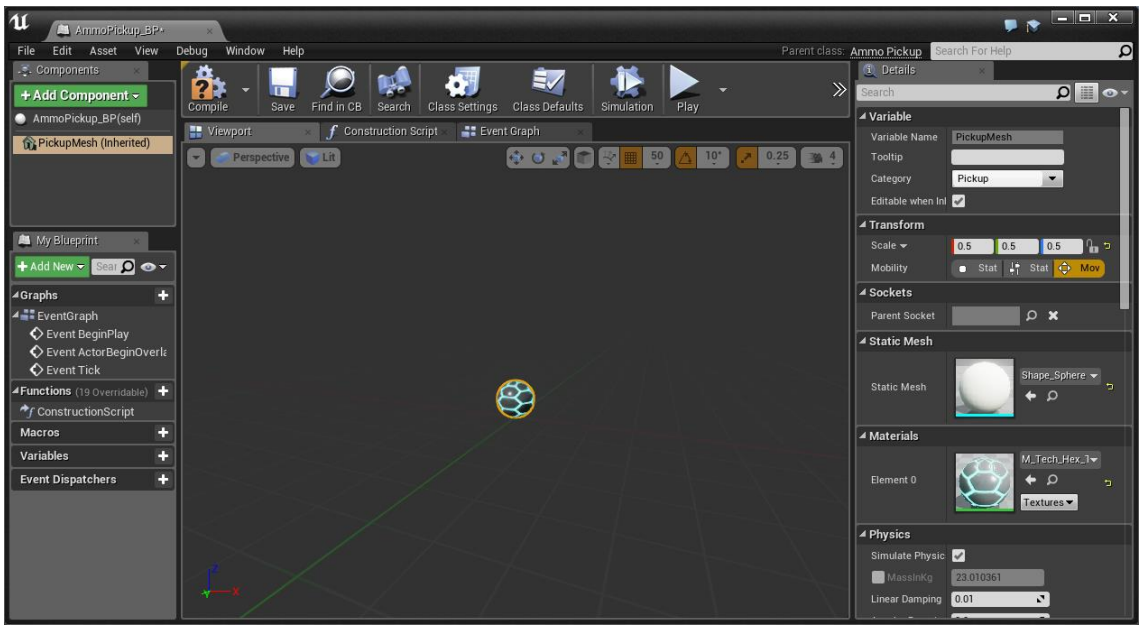


Kuva 16. AmmoPickup.cpp:n muutokset.

Tallenna muutokset koodiin ja Unreal Editorin puolella paina Compile-nappia. Meillä on nyt valmiina health- ja ammuslaatikon koodit, mutta meillä ei ole niille varsinaisia objekteja, jota voisi lisätä pelimaailmaan.

3.1.5 Health- ja ammuslaatikoiden Blueprint:it

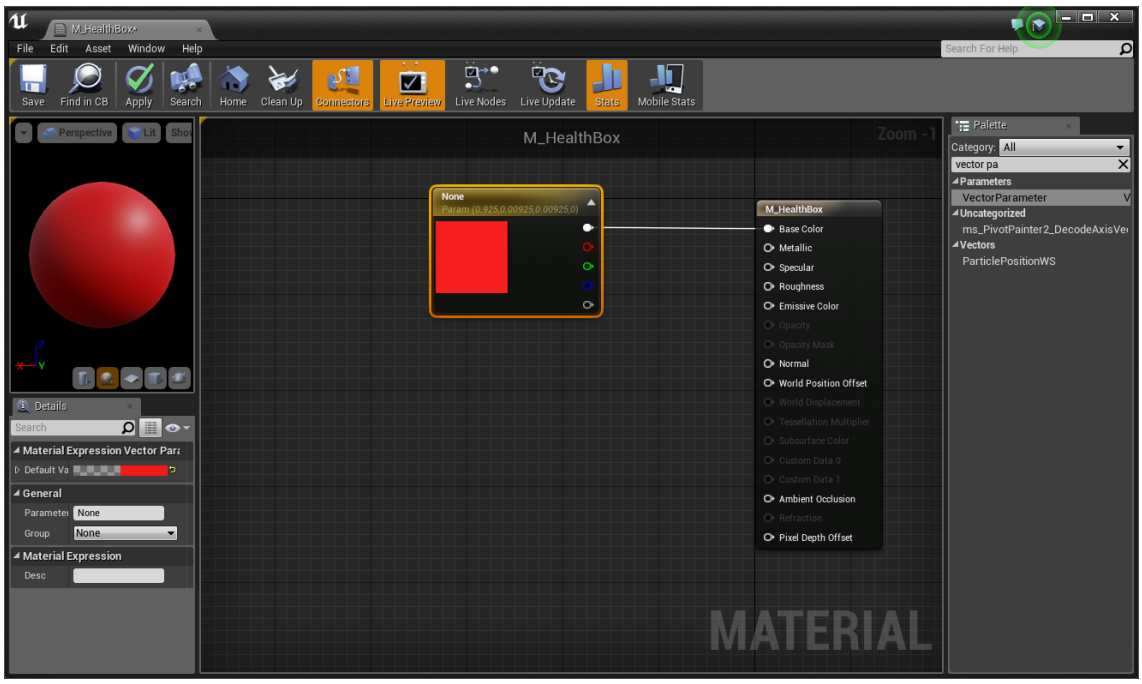
Kuten teimme aikaisemmin Blueprintin Pickup-luokasta, tehdään nyt samalla tavalla uudet Blueprintit HealthPickup- ja AmmoPickup-luokista. Content Browserissa HealthPickup C++-luokan kohdalla painetaan hiiren oikealla napilla, ja valitaan Create Blueprint class based on HealthPickup. Laitetaan nimeksi HealthPickup_BP. Kuten aikaisemminkin, HealthPickup_BP-ikkunassa valitaan PickupMesh-komponentti ja asetetaan Static Meshiksi 1M_Cube. Samoin tehdään AmmoPickupille, paitsi että laitetaankin Static Meshiksi Shape_Sphere. Static Meshin materiaaliksi on automaattisesti valittu Materials-kohdassa valittu M_Basic_Wall. Haluamme kuitenkin, että ammuslaatikko olisi helposti erotettavissa muusta pelimaailmasta, joten vaihdetaan materiaaliksi M_Tech_Hex_Tile_Pulse. Health- ja ammuslaatikot ovat vähän liian suuria, joten puolitetaan niiden koko. Static Mesh -kohdan yläpuolella on Transform-kohta, jolla voidaan skaalata Static Meshiä. Muutetaan kaikki skaalat arvosta 1.0 arvoon 0.5. Valmiin AmmoPickup_BP:n pitäisi näyttää kuvan 17 mukaiselta.



Kuva 17. Valmis AmmoPickup_BP.

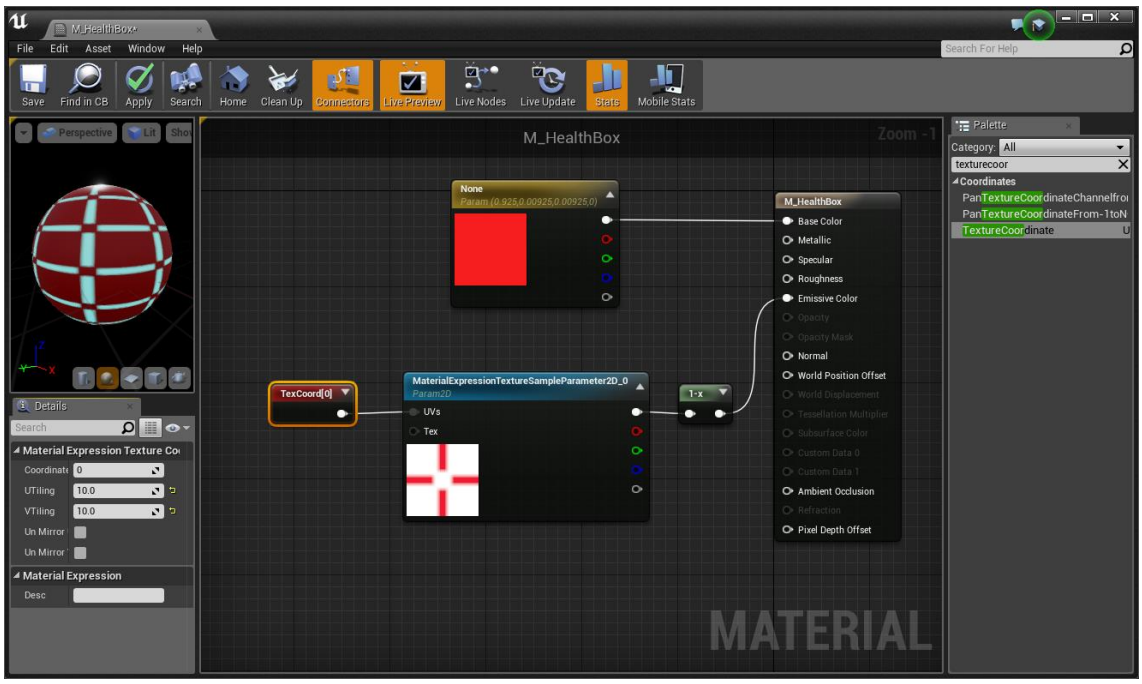
Tallennetaan AmmoPickup_BP:n muutokset painamalla ensin Compile- ja sitten Save-nappia. Health-laatikolle ei ole tarkoituksen mukaista materiaalia valmiina, joten tehdään sellainen. Aloitetaan valitsemalla Content Browserissa FirstPersonCPP-kansio, ja lisäämällä sinne uusi kansio. Laitetaan sille nimeksi Materials. Avaa Materials-kansio ja lisää sinne uusi materiaali painamalla hiiren oikeaa-, tai Content Browserin Add New -nappia, ja valitsemalla sieltä Material. Anna nimeksi M_HealthBox. Avaa uusi materiaali uuteen ikkunaan.

M_HealthBox-ikkunan vasemmassa laidassa on 3D-renderoitu näkymä lopullisesta materiaalista. Keskellä on M_HealthBox-laatikko, joka kuvastaa materiaalin hyväksymiä syötteitä. Oikeassa laidassa on Palette-paneeli. Se on lista Blueprint-elementeistä, joilla voi tuottaa ja muokata syötteitä materiaalille. Lähdetään siitä, että tehdään materiaalista punainen. Etsi Palette-listasta Vector Parameter ja raahaa se keskialueelle. Vector Parameterillä saa kätevästi RGB-värin. Tuplaklikkaa Vector Parameetterin mustaa neliötä ja valitse väriksi sopivanlainen punainen. Sen jälkeen raahaa Vector Parameterin ylin valkoinen pallukka M_HealthBoxin Base Color pallukkaan. Tulokseksi pitäisi saada kuvan 18 mukainen asetelma.



Kuva 18. Punainen väri yhdistettynä materiaaliin.

Punainen väri kävisi jo itselläänkin, mutta lisätään pieni yksityiskohta hyväksikäyttämällä valmista tekstuuria. Etsi Palette-listasta TextureSampleParameter2D ja raahaa se keskialueelle. Nyt vasemmassa laidassa 3D-näkymän alla löytyy Details-paneeli, ja sieltä löytyy Texture-kohta, jossa näkyy valittu tekstuuri. Vaihda siihen tekstuuriksi FirstPersonCrosshair. Kuten aikaisemminkin, lähde raahaamaan TextureSampleParameter2D:n ylintä valkoista pallukkaa, mutta älä yhdistä sitä mihinkään vaan ”pudota” se keskialueen tyhjään kohtaan. Unreal Editor haluaa yhdistää linjan johonkin ja tarjoaa uuden Palette-listan. Kirjoita etsintä kohtaan ”1-”, jolla löydät elementin OneMinus. Tämä kääntää TextureSampleParameter2D:n syötteen päinvastaiseksi. Yhdistä OneMinus M_HealthBox:in kohtaan Emissive Color. Materiaali ei ihan vielä näytä hyvältä. Tekstuuri on liian iso, joten pienennetään sitä käyttämällä TextureCoordinaten tilingia. Etsi Palette-listasta TextureCoordinate ja raahaa se keskialueelle TextureSampleParameter2D:n vasemmalle puolelle. Muuta TextureCoordinaten UTiling ja VTiling kohdat arvosta 1.0 arvoon 10.0. Yhdistä TextureCoordinaten pallukka TextureSampleParameter2D:n UVs-pallukkaan. Lopputuloksena pitäisi olla tarkoituksen mukainen materiaali health-laatikolle ja kuvan 19 mukainen asetelma.

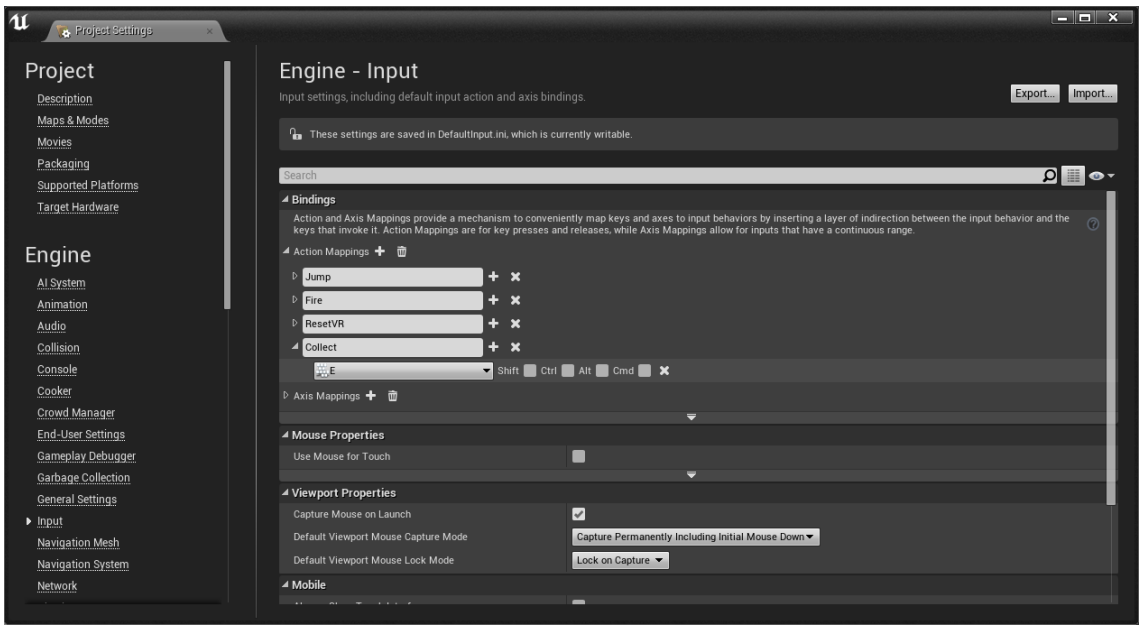


Kuva 19. Valmis materiaali M_HealthBox.

Tallenna materiaali painamalla ensin Apply-nappia M_HealthBox-ikkunan työkalurivillä ja sitten painamalla Save-nappia. Materiaalin yhdistäminen Health-laatikkoon tapahtuu samalla tavalla kuin ammuslaatikon tapauksessakin. HealthPickup_BP Blueprintissä valitaan PickupMesh Component ja vaihdetaan Materials kohtaan M_HealthBox. Nyt meillä on oikean näköiset health- ja ammuslaatikot. Niitä ei voi tosin vielä käyttää. Siihen syvennytään seuraavassa luvussa.

3.2 Pelaajan liikkuminen pelimaailmassa

First Person Templatessa on valmiiksi asetettu näppäimet pelaajan liikkumiselle ja ampumiselle. Haluamme lisätä näppäimen toiminnolle, jolla voimme kerätä health- ja ammuslaatikoita. Näppäimien asetukset löytyvät Unreal Editorin työkalurivillä painamalla Settings-nappia ja valitsemalla Project Settings. Project Settings -ikkunassa Enginen alta löytyy kohta Input. Inputista löytyvät näppäinten asetukset. Axis Mappings -kohdassa on näppäimet pelaajan liikkumiselle ja Action Mappings -kohdassa on näppäimet eri toiminnoille, kuten ampumiselle. Paina Action Mappings -kohdan vieressä olevaa plus nappia. Se lisää uuden toiminnon NewActionMapping_0. Muutetaan sen nimeksi Collect ja asetetaan näppäimeksi E. Kuvassa 20 muutetut näppäinasetukset.



Kuva 20. Näppäinten asetukset.

Pelaajan hahmolle haluamme lisätä healthin, joka vähenee vihollisten osuessa, ja jota saa lisää health-laatikoista. Haluamme lisätä myös ammukset, jotka kuluvat, kun ampuu aseella, ja joita saa lisää ammuslaatikoista. Lisäämme myös toiminnon, jolla health- ja ammuslaatikoita saa kerättyä.

Aloitetaan avaamalla FPStutorialCharacter.h ja lisäämällä tarvittavat funktiot:

```
public:
UFUNCTION(BlueprintPure, Category = "Health")
float GetInitialHealth();

UFUNCTION(BlueprintPure, Category = "Health")
float GetCurrentHealth();

UFUNCTION(BlueprintCallable, Category = "Health")
void UpdateHealth(float HealthChange);

UFUNCTION(BlueprintPure, Category = "Ammo")
float GetInitialAmmoAmount();

UFUNCTION(BlueprintPure, Category = "Ammo")
float GetCurrentAmmoAmount();

UFUNCTION(BlueprintCallable, Category = "Ammo")
void ChangeAmmoAmount(float AmmoAmountChange);

UFUNCTION(BlueprintCallable, Category = "Ammo")
void SetFiringAbility(bool bAllowFiring);
```

SetFiringAbility-funktiolla voidaan estää pelaajan ampumistoiminto. Tätä funktiota tarvitaan myöhemmässä vaiheessa. Seuraavaksi lisätään muuttujat:

```
protected:
UFUNCTION(BlueprintCallable, Category = "Pickups")
void CollectPickups();

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Health", meta =
(BlueprintProtected = "true"))
float InitialHealth;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Ammo", meta =
(BlueprintProtected = "true"))
float InitialAmmoAmount;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Ammo", meta =
(BlueprintProtected = "true"))
float WeaponAmmoCost;

private:
UPROPERTY(VisibleAnywhere, Category = "Health")
float CurrentHealth;

UPROPERTY(VisibleAnywhere, Category = "Ammo")
float CurrentAmmoAmount;

bool bCanFire;

UPROPERTY(VisibleAnywhere, BlueprintReadOnly, meta = (AllowPrivateAccess =
"true"))
class USphereComponent* CollectionSphere;
```

Meidän tarvitsee tietää mitä laatikoita ollaan keräämässä, kun pelaaja painaa keräämisnappia. Tässä auttaa Unreal Enginen toiminnot, jotka laskevat sisäkkäin olevia objekteja. Lisäämme pelaajan hahmolle CollectionSphere Componentin, joka määrittää alueen, jossa tarkastelemme sisäkkäisiä objekteja.

Seuraavaksi avaa FPStutorialCharacter.cpp ja lisää konstruktoriin lähtöasetukset CollectionSpherelle, healthille ja ammuksille seuraavanlaisesti:

```
CollectionSphere =
CreateDefaultSubobject<USphereComponent>(TEXT("CollectionSphere"));
CollectionSphere->SetupAttachment(RootComponent);
CollectionSphere->SetSphereRadius(200.0f);

InitialHealth = 100.0f;
CurrentHealth = InitialHealth;

InitialAmmoAmount = 100.0f;
CurrentAmmoAmount = InitialAmmoAmount;

WeaponAmmoCost = -10.0f;
bCanFire = true;
```

Sitten tehdään toteutukset aikaisemmin lisätyille funktioille:

```
float AFPStutorialCharacter::GetInitialHealth()
{
	return InitialHealth;
}

float AFPStutorialCharacter::GetCurrentHealth()
{
	return CurrentHealth;
}

void AFPStutorialCharacter::UpdateHealth(float HealthChange)
{
	float NewHealth = CurrentHealth + HealthChange;

	if (NewHealth > 100.0f)
	{
		CurrentHealth = 100.0f;
	}
	else if (NewHealth < 0.0f)
	{
		CurrentHealth = 0.0f;
		UE_LOG(LogClass, Log, TEXT("Player Died!!"));
	}
	else
	{
		CurrentHealth = NewHealth;
	}
}

float AFPStutorialCharacter::GetInitialAmmoAmount()
{
	return InitialAmmoAmount;
}

float AFPStutorialCharacter::GetCurrentAmmoAmount()
{
	return CurrentAmmoAmount;
}

void AFPStutorialCharacter::ChangeAmmoAmount(float AmmoAmountChange)
{
	float NewAmmoAmount = CurrentAmmoAmount + AmmoAmountChange;

	if (NewAmmoAmount > 100.0f)
	{
		CurrentAmmoAmount = 100.0f;
	}
	else if (NewAmmoAmount < 0.0f)
	{
		CurrentAmmoAmount = 0.0f;
	}
	else
	{
		CurrentAmmoAmount = NewAmmoAmount;
	}
}

void AFPStutorialCharacter::SetFiringAbility(bool bAllowFiring)
{
	bCanFire = bAllowFiring;
}
```

Seuraavaksi meidän tarvitsee muokata ampumisfunktiota, jotta pelaaja ei pysty enää ampumaan, kun ammukset ovat loppu. Etsi funktio OnFire() ja muokkaa sitä näin:

```
void AFPStutorialCharacter::OnFire()
{
	if (CurrentAmmoAmount >= FMath::Abs(WeaponAmmoCost) && bCanFire)
	{
		...
		//Tähän funktion muut toiminnot
		...

		ChangeAmmoAmount(WeaponAmmoCost);
	}
	else
	{
		UE_LOG(LogClass, Log, TEXT("Out of Ammo"));
	}
}
```

Näin tarkistetaan, onko pelaajalla tarpeeksi ammuksia, ja jos ei ole, niin kerrotaan siitä. Seuraavaksi lisätään funktio, jolla saadaan kerättyä health- ja ammuslaatikoita. Sitä varten, meidän tarvitsee lisätä Pickup-luokat. Eli FPStutorialCharacter.cpp include-osioon kirjoita:

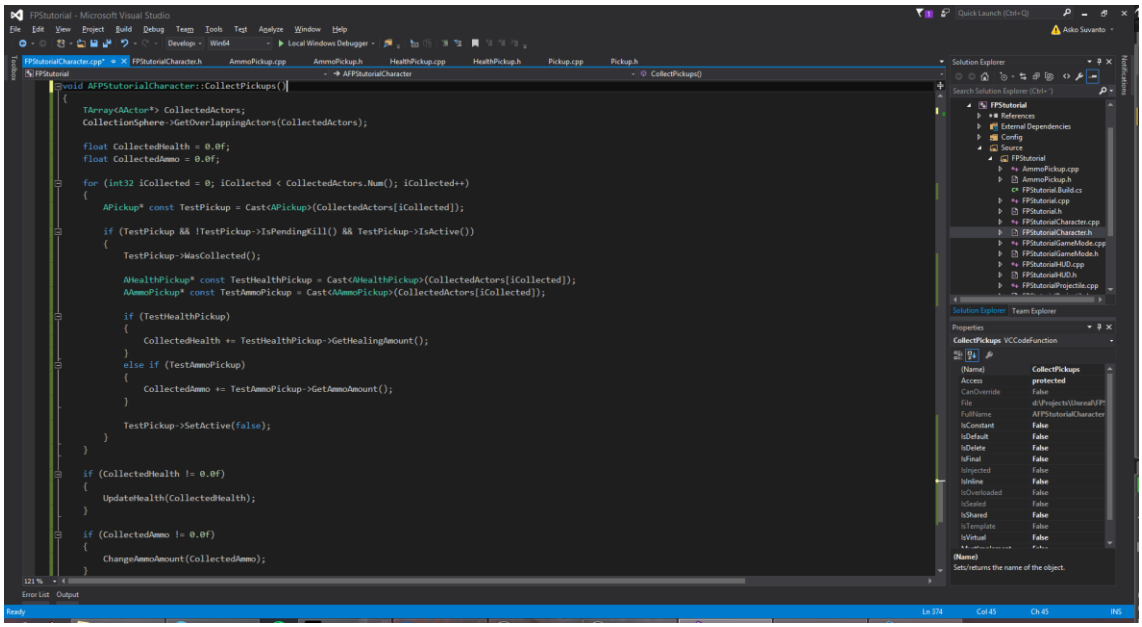
```
#include "Pickup.h"
#include "HealthPickup.h"
#include "AmmoPickup.h"
```

Sitten lisää CollectPickups-funktio seuraavanlaisesti:

```
void AFPStutorialCharacter::CollectPickups()
{
	TArray<AActor*> CollectedActors;
	CollectionSphere->GetOverlappingActors(CollectedActors);

	float CollectedHealth = 0.0f;
	float CollectedAmmo = 0.0f;

	for (int32 iCollected = 0; iCollected < CollectedActors.Num(); iCollected++)
	{
		APickup* const TestPickup = Cast<APickup>(CollectedActors[iCollected]);

		if (TestPickup && !TestPickup->IsPendingKill() &&
		TestPickup->IsActive())
		{
			TestPickup->WasCollected();

			AHealthPickup* const TestHealthPickup = Cast<AHealthPickup>(
			CollectedActors[iCollected]);
```

```
                AAmmoPickup* const TestAmmoPickup = Cast<AAmmoPickup>(
                CollectedActors[iCollected]);

                if (TestHealthPickup)
                {
                        CollectedHealth += TestHealthPickup->GetHealingAmount();
                }
                else if (TestAmmoPickup)
                {
                        CollectedAmmo += TestAmmoPickup->GetAmmoAmount();
                }

                TestPickup->SetActive(false);
        }
}

if (CollectedHealth != 0.0f)
{
        UpdateHealth(CollectedHealth);
}

if (CollectedAmmo != 0.0f)
{
        ChangeAmmoAmount(CollectedAmmo);
}
```

CollectionSpheren GetOverlappingActors-funktio etsii kaikki Actor-tyyppiset objectit, jotka ovat sen sisällä ja lisää ne taulukkoon. Sen jälkeen käydään läpi kaikki taulukon objektit ja, jos se on health- tai ammuslaatikko, niin tallennetaan sen arvo ja deaktivoidaan se. Kuvassa 21 näkyy CollectPickups-funktion toteutus.



Kuva 21. CollectPickup-funktio.

Viimeiseksi lisätään Collect-näppäimelle toiminto. Pelaajan hahmon syötteet löytyvät FPStutorialCharacter.cpp:stä funktiosta SetupPlayerInputComponent. Sinne lisää rivi:

```
PlayerInputComponent->BindAction("Collect", IE_Pressed, this,
&AFPStutorialCharacter::CollectPickups);
```

Tämä asettaa Collect-näppäimen kutsumaan CollectPickups-funktiota. Tallenna muutokset ja Unreal Editorissa paina Compile-nappia. Nyt meillä on pelaajan hahmolla tietty määrä healthia ja ammuksia. Ammukset kuluvat ampumalla ja niitä saa lisää keräämällä ammuslaatikoita. Seuraavassa luvussa lisätään pelimaailmaan vihollisia, sekä satunnaisesti ilmestyviä health- ja ammuslaatikoita.

3.3 Interaktiivinen pelimaailma

Pelimaailmaan saadaan eloa vihollisilla, jotka yrittävät tappaa pelaajan hahmon, sekä health- ja ammuslaatikoilla, jotka satunnaisesti ilmestyvät eripuolille pelialuetta.

3.3.1 Objektien luominen dynaamisesti

Aloitetaan tekemällä SpawnVolume. SpawnVolume on objekti, joka luo uusia objekteja satunnaisesti satunnaisiin paikkoihin sisällään. Tee uusi C++-luokka, joka periytyy Actor-luokasta, ja laita sen nimeksi SpawnVolume. Lisää SpawnVolume.h:hon seuraavat funktiot ja muuttujat:

```
FORCEINLINE UBoxComponent* GetWhereToSpawn() const { return WhereToSpawn;
}

UFUNCTION(BlueprintPure, Category = "Spawning")
FVector GetRandomPointInVolume();

UFUNCTION(BlueprintCallable, Category = "Spawning")
void SetSpawningActive(bool bShouldSpawn);

protected:
//UPROPERTY(EditAnywhere, Category = "Spawning")
TSubclassOf<class APickup> WhatToSpawn;

UPROPERTY(EditAnywhere, Category = "Spawning")
TArray<TSubclassOf<class APickup>> WhatToSpawnArray;

FTimerHandle SpawnTimer;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Spawning")
float SpawnDelayRangeLow;
```

```
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Spawning")
float SpawnDelayRangeHigh;

private:
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Spawning", meta =
(AllowPrivateAccess = "true"))
class UBoxComponent* WhereToSpawn;

void SpawnPickup();

float SpawnDelay;
```

WhereToSpawn on alue, jonka sisällä luodaan uusia objekteja. Alueen koko ja paikka asetetaan Unreal Editorissa. WhatToSpawnArrayhin listataan objektit, joita halutaan luoda. Nämä asetetaan Unreal Editorissa. SpawnTimer hallitsee ajoitusta objektien luomisessa. SpawnVolume.cpp:n konstruktoriin lisätään seuraavat lähdearvot:

```
ASpawnVolume::ASpawnVolume()
{
    PrimaryActorTick.bCanEverTick = false;

    WhereToSpawn =
    CreateDefaultSubobject<UBoxComponent>(TEXT("WhereToSpawn"));
    WhereToSpawn->SetupAttachment(RootComponent);

    SpawnDelayRangeLow = 1.0f;
    SpawnDelayRangeHigh = 3.5f;

}
```

Seuraavaksi lisätään funktioiden toteutukset:

```
void ASpawnVolume::SetSpawningActive(bool bShouldSpawn)
{
    if (bShouldSpawn)
    {
        SpawnDelay = FMath::FRandRange(SpawnDelayRangeHigh,
        SpawnDelayRangeHigh);

        GetWorldTimerManager().SetTimer(SpawnTimer, this,
        &ASpawnVolume::SpawnPickup, SpawnDelay, false);
    }
    else
    {
        GetWorldTimerManager().ClearTimer(SpawnTimer);
    }
}

FVector ASpawnVolume::GetRandomPointInVolume()
{
    FVector SpawnOrigin = WhereToSpawn->Bounds.Origin;
    FVector SpawnExtend = WhereToSpawn->Bounds.BoxExtent;

    Return UKismetMathLibrary::RandomPointInBoundingBox(SpawnOrigin,
    SpawnExtend);
```

```
}

void ASpawnVolume::SpawnPickup()
{
	if (WhatToSpawnArray.Num() > 0)
	{
		int32 RandomIndex = FMath::FRandRange(0, WhatToSpawnArray.Num());

		WhatToSpawn = WhatToSpawnArray[RandomIndex];
	}

	if (WhatToSpawn != NULL)
	{
		UWorld* const World = GetWorld();

		if (World)
		{
			FActorSpawnParameters SpawnParams;

			SpawnParams.Owner = this;
			SpawnParams.Instigator = Instigator;

			FVector SpawnLocation = GetRandomPointInVolume();

			FRotator SpawnRotarion;

			SpawnRotarion.Yaw = FMath::FRand() * 360.0f;
			SpawnRotarion.Pitch = FMath::FRand() * 360.0f;

			SpawnRotarion.Roll = FMath::FRand() * 360.0f;

			APickup* SpawnedPickup = World->SpawnActor<APickup>(
			WhatToSpawn, SpawnLocation, SpawnRotarion, SpawnParams);

			SpawnDelay = FMath::FRandRange(SpawnDelayRangeHigh,
			SpawnDelayRangeHigh);

			GetWorldTimerManager().SetTimer(SpawnTimer, this,
			&ASpawnVolume::SpawnPickup, SpawnDelay, false);
		}
	}

}
```

SpawnVolume.cpp:n alkuun tarvitsee vielä lisätä matematiikkakirjasto ja Pickup:

```
#include "Kismet/KismetMathLibrary.h"
#include "Pickup.h"
```

KismetMathLibrarystä löytyy valmis apufunktio, joka laskee satunnaisen pisteen tiedetyn alueen sisällä. Sillä saadaan laskettua kohta, johon uusi laatikko luodaan. Tallenna muutokset ja Unreal Editorissa paina Compile-nappia.

Toisin kuten olemme tehneet aikaisemmin, emme tee SpawnVolumesta Blueprintiä, vaan raahaa SpawnVolume C++-luokka suoraan pelialueeseen. SpawnVolume näkyy neliönä,

joka kuvastaa aluetta, jonka sisällä luodaan uusia objekteja. Liikuta laatikko pelialueen yläpuolelle ja muotoile siitä sen kokoinen alue, kuin minne haluat uusien health- ja ammuslaatikoiden ilmestyvän. Oikealla laidalla Details-paneelista löytyy Spawning-kohta ja siellä tekemämme WhatToSpawnArray. Lisää siihen kaksi elementtiä ja valitse listasta AmmoPickup_BP ja HealthPickup_BP.

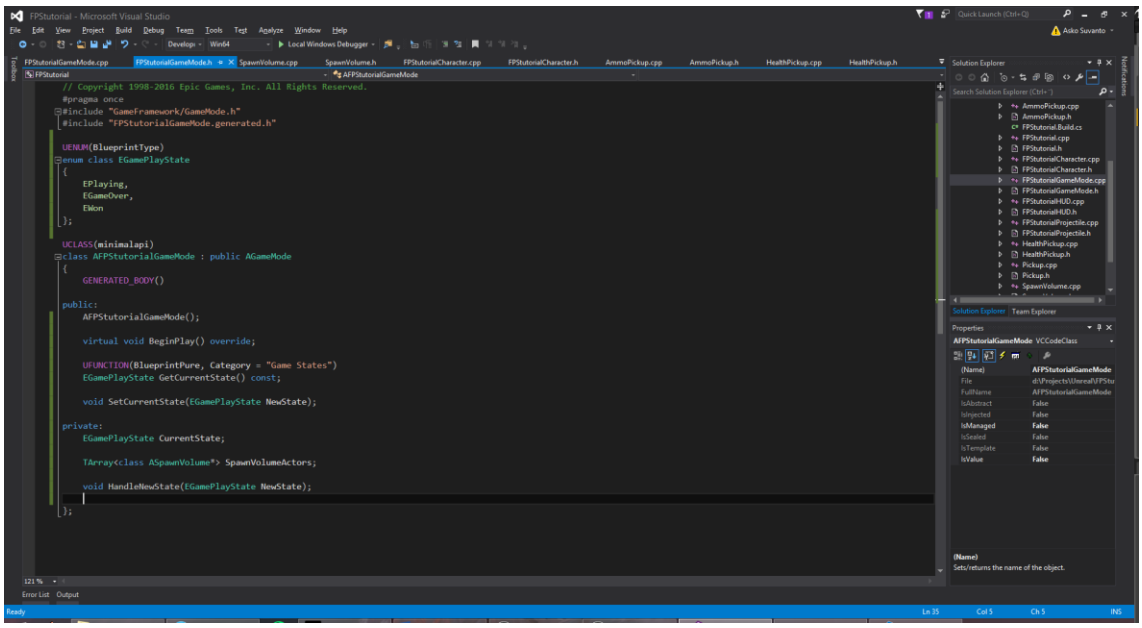
SpawnVolume ei kuitenkaan tee pelissä vielä mitään. Sitä varten meidän tarvitsee asettaa oma GameMode. GameMode on pelin taustalla toimiva Blueprint, joka pitää huolta esim. GameStatesta ja HUD:ista. Avaa FPStutorialGameMode.h. Lisätään peliin GameStatet. Ensiksi tehdään enum-luokka tarvitsemistamme GameStateista:

```
 UENUM(BlueprintType)
enum class EGamePlayState
{
	EPlaying,
	EGameOver,
	EWon
};
```

Lisätään tarvittavat muuttujat ja funktiot:

```
public:
virtual void BeginPlay() override;

UFUNCTION(BlueprintPure, Category = "Game States")
EGamePlayState GetCurrentState() const;

void SetCurrentState(EGamePlayState NewState);

private:
EGamePlayState CurrentState;

TArray<class ASpawnVolume*> SpawnVolumeActors;

void HandleNewState(EGamePlayState NewState);
```

Funktio BeginPlay ajetaan aina, kun objekti luodaan pelimaailmaan. SpawnVolumeActors-taulukko pitää kirjaa pelimaailmassa olevista SpawnVolumeista. Tällä hetkellä pelissä on vain yksi SpawnVolume, mutta tämän arrayn avulla niitä voi lisätä pelimaailmaan ilman, että tarvitsee muokata koodia. HandleNewState-funktiossa tehdään kaikki tarvittavat toimenpiteet, kun GameState muuttuu. Kuvassa 22 näkyy FPStutorialGameMode.h:hon tehdyt muutokset kokonaisuudessaan.



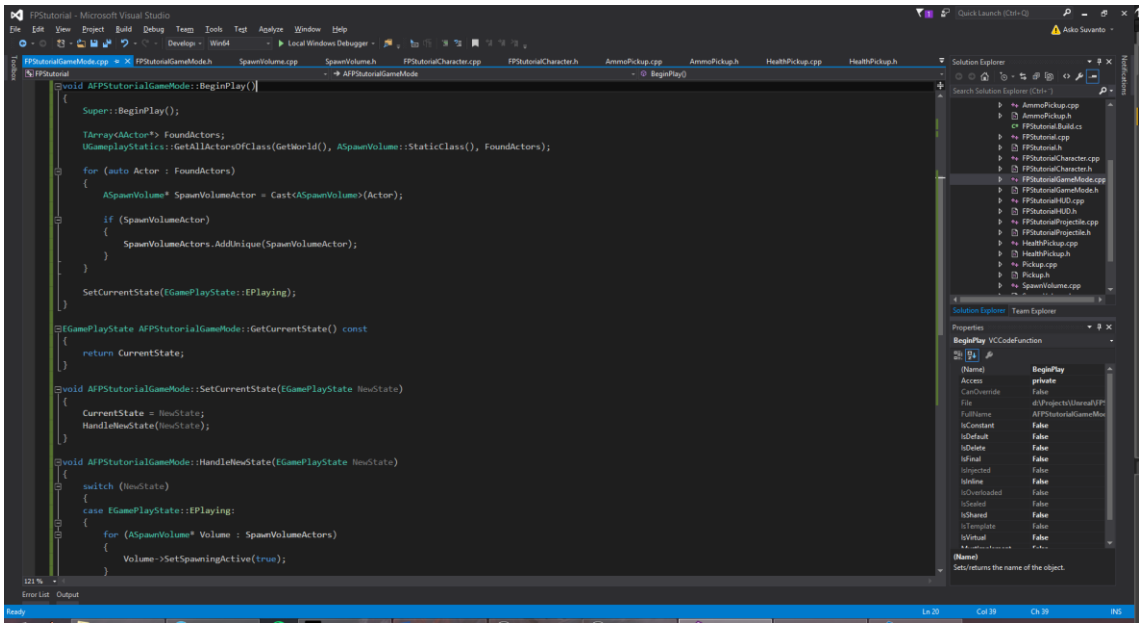
Kuva 22. FPStutorialGameMode.h

Lisätään FPStutorialGameMode.cpp:hen funktioiden toiteutukset seruraavalla tavalla:

```
void AFPStutorialGameMode::BeginPlay()
{
	Super::BeginPlay();

	TArray<AActor*> FoundActors;

	UGameplayStatics::GetAllActorsOfClass(GetWorld(),
	ASpawnVolume::StaticClass(), FoundActors);

	for (auto Actor : FoundActors)
	{
		ASpawnVolume* SpawnVolumeActor = Cast<ASpawnVolume>(Actor);

		if (SpawnVolumeActor)
		{
			SpawnVolumeActors.AddUnique(SpawnVolumeActor);
		}
	}

	SetCurrentState(EGamePlayState::EPlaying);
}

EGamePlayState AFPStutorialGameMode::GetCurrentState() const
{
	return CurrentState;
}

void AFPStutorialGameMode::SetCurrentState(EGamePlayState NewState)
{
	CurrentState = NewState;
	HandleNewState(NewState);
}

void AFPStutorialGameMode::HandleNewState(EGamePlayState NewState)
{
```

```
        switch (NewState)
        {
        case EGamePlayState::EPlaying:
        {
                for (ASpawnVolume* Volume : SpawnVolumeActors)
                {
                        Volume->SetSpawningActive(true);
                }
        }
        break;

        case EGamePlayState::EWon:
        {
                //Tähän palataan myöhemmässä vaiheessa
        }
        break;

        case EGamePlayState::EGameOver:
        {
                //Tähän palataan myöhemmässä vaiheessa
        }
        break;

        }
}
```

Funktiossa BeginPlay, eli kun peli aloitetaan, listataan kaikki pelimaailmassa olevat SpawnVolumet, sekä asetetaan uudeksi GameStateksi EPlaying. GameState EPlaying aktivoi kaikki listatut SpawnVolumet. Kuvassa 23 näkyy FPStutorialGameMode.cpp:hen tehtyjä muutoksia.



Kuva 23. FPStutorialGameMode.cpp.

Tallenna muutokset ja Unreal Editor:issa paina Compile-nappia. Nyt jos haluat testata peliä, niin pitäisi health- ja ammuslaatikoiden alkaa satelemaan pelissä.

3.3.2 Vihollisen C++-luokan lisääminen

Nyt kun pelaajan hahmolla on toimiva health, on aika tehdä vihollinen, joka kuluttaa sitä. Aloitetaan tekemällä uusi C++-luokka. Aikaisemmasta poiketen, vihollisen C++-luokka periytyykin Pawn-luokasta. Anna uuden C++-luokan nimeksi EnemyUnit. Avaa EnemyUnit.h. Haluamme tehdä viholliselle yksinkertaisen tekoälyn. Vihollisen täytyy kulkea määrättyä reittiä pitkin, kunnes se huomaa pelaajan hahmon olevan tarpeeksi lähellä, ja alkaa ampumaan sitä. Lisää EnemyUnit.h:hon seuraavat Statet:

```
UENUM(BlueprintType)
enum class EEnemyUnitState : uint8
{
	SE_AIOff UMETA(DisplayName = "AI Off"),
	SE_Patrolling UMETA(DisplayName = "Patrolling"),
	SE_Hunting UMETA(DisplayName = "Hunting")
};
```

UMETA-makro antaa uudelleen nimetä enum-muuttujat, jotka näkyvät Unreal Editorissa. Lisätään tarvittavat muuttujat ja funktiot seuraavasti:

```
public:
UPROPERTY(EditAnywhere, Category = "AI")
TArray<AActor*> WaypointList;

UPROPERTY(EditAnywhere, Category = "AI")
float MoveSpeed;

UFUNCTION(BlueprintPure, Category = "AI")
float GetEnemyHealth();

UFUNCTION(BlueprintPure, Category = "AI")
float GetEnemyInitialHealth();

UFUNCTION(BlueprintCallable, Category = "AI")
void UpdateHealth(float HealthChange);

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "AI")
EEnemyUnitState EnemyState;

UPROPERTY(EditDefaultsOnly, Category = "AI")
TSubclassOf<class AFPStutorialProjectile> ProjectileClass;

UPROPERTY(EditAnywhere, Category = "AI")
FVector FiringOffset;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "AI")
USceneComponent* EnemyWeapon;

UFUNCTION(BlueprintCallable, Category = "AI")
```

```
void SetAIActive(bool bShouldAct);

/** Sound to play each time we fire */
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "AI")
class USoundBase* FireSound;

FORCEINLINE class USphereComponent* GetSphereComponent() const { return
DetectionSphere; }

UFUNCTION(BlueprintImplementableEvent, Category = "Effects")
void DeathEffect();

protected:
UFUNCTION(BlueprintImplementableEvent, Category = "Effects")
void HealthChangeEffect();

private:
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, meta = (AllowPrivateAccess =
"true"))
class USphereComponent* DetectionSphere;

UPROPERTY(VisibleAnywhere, Category = "AI", meta = (BlueprintProtected = "true"))
bool bHasTarget;

UPROPERTY(VisibleAnywhere, Category = "AI", meta = (BlueprintProtected = "true"))
float EnemyHealth;

void Patrol(float DeltaTime);
void Hunt(float DeltaTime);
void FindTarget();
void Attack();

float InitialEnemyHealth;
float AttackTimer;
ACharacter* TargetCharacter;
int32 WaypointIndex;
```

Uutena makrona on BlueprintImplementableEvent, joka tarkoittaa sitä, että funktion toteutus tehdäänkin Blueprintissä, eikä koodissa. EditDefaultsOnly mahdollistaa muuttujan muokkaamisen vain Unreal Editorin Blueprint-ikkunassa. EnemyWeapon on pelkkä Blueprint-komponentti. Käytämme tätä referenssipisteenä kohtaan, josta vihollinen ampuu. Palaamme HealthChangeEffect- ja DeathEffect-funktioiden toteutukseen seuraavassa luvussa. Lisää konstruktori- ja BeginPlay-funktioon nämä:

```
AEnemyUnit::AEnemyUnit()
{
    // Set this pawn to call Tick() every frame.  You can turn this off to improve
    performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

    DetectionSphere =
    CreateDefaultSubobject<USphereComponent>(TEXT("DetectionSphere"));

    DetectionSphere->SetupAttachment(RootComponent);
    DetectionSphere->SetSphereRadius(1000.0f);
```

```
	MoveSpeed = 10.0f;
	InitialEnemyHealth = 100.0f;
	EnemyHealth = InitialEnemyHealth;
	EnemyState = EEnemyUnitState::SE_AIOff;

	AttackTimer = 0.0f;
	FiringOffset = FVector(0.0f, 0.0f, -50.0f);
}

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

	WaypointIndex = 0;
	bHasTarget = false;
}
```

Tick-funktio on funktio, joka ajetaan jokainen frame. Lisätään sinne siis vihollisen Statet:

```
void AEnemyUnit::Tick( float DeltaTime )
{
	Super::Tick( DeltaTime );

	switch (EnemyState)
	{
	case EEnemyUnitState::SE_AIOff:
	{
		// Do nothing
	}
	break;

	case EEnemyUnitState::SE_Patrolling:
	{
		Patrol(DeltaTime);
		FindTarget();
	}
	break;

	case EEnemyUnitState::SE_Hunting:
	{
		Hunt(DeltaTime);
	}
	break;
	}
}
```

Lisätään vielä puuttuvat funktiot:

```
void AEnemyUnit::SetAIActive(bool bShouldAct)
{
	if (bShouldAct)
	{
		EnemyState = EEnemyUnitState::SE_Patrolling;
	}
	else
	{
		EnemyState = EEnemyUnitState::SE_AIOff;
```

```
	}
}

float AEnemyUnit::GetEnemyHealth()
{
	return EnemyHealth;
}

float AEnemyUnit::GetEnemyInitialHealth()
{
	return InitialEnemyHealth;
}

void AEnemyUnit::UpdateHealth(float HealthChange)
{
	float NewHealth = EnemyHealth + HealthChange;

	if (NewHealth > InitialEnemyHealth)
	{
		EnemyHealth = InitialEnemyHealth;
	}
	else if (NewHealth <= 0)
	{
		EnemyHealth = 0;

		DeathEffect();
		Destroy();
	}
	else
	{
		EnemyHealth = NewHealth;
	}

	HealthChangeEffect();
}

void AEnemyUnit::Patrol(float DeltaTime)
{
	if (WaypointList.Num() > 0)
	{
		AActor* TestPoint = Cast<AActor>(WaypointList[WaypointIndex]);

		if (TestPoint != NULL)
		{
			FVector Direction = GetActorLocation() -
			TestPoint->GetActorLocation();

			FVector CurrentLocation = GetActorLocation();
			FVector TargetLocation = TestPoint->GetActorLocation();

			if (Direction.Size() < 2.0f)
			{
				WaypointIndex++;

				if (WaypointIndex >= WaypointList.Num())
				{
					WaypointIndex = 0;
				}
			}
			else
			{
```

```
                        FVector NewLocation = FMath::VInterpConstantTo(
                        CurrentLocation, TargetLocation, DeltaTime, MoveSpeed *
                        10.0f);

                        SetActorLocation(NewLocation);

                        Direction.Set(Direction.X, Direction.Y, 0.0f);

                        FRotator TargetRotation = FRotationMatrix::MakeFromX(
                        Direction).Rotator();

                        FRotator NewRotation = FMath::Lerp(GetActorRotation(),
                        TargetRotation, DeltaTime);

                        SetActorRotation(NewRotation);
                    }
                }
            }
        }

        void AEnemyUnit::Hunt(float DeltaTime)
        {
            FVector Direction = GetActorLocation() - Cast<AFPStutorialCharacter>(
            TargetCharacter)->GetActorLocation();

            Direction.Set(Direction.X, Direction.Y, 0.0f);

            FRotator TargetRotation = FRotationMatrix::MakeFromX(Direction).Rotator();

            FRotator NewRotation = FMath::Lerp(GetActorRotation(), TargetRotation,
            DeltaTime * 2.0f);

            SetActorRotation(NewRotation);

            FVector CurrentDirection = NewRotation.Vector();
            FVector TargetDirection = TargetRotation.Vector();

            float AngleToTarget = FMath::RadiansToDegrees(
            FMath::Acos(FVector::DotProduct(CurrentDirection, TargetDirection)));

            if (AngleToTarget < 8.0f)
            {
                if (AttackTimer <= 0)
                {
                    Attack();
                    AttackTimer = 1.5f;
                }
            }

            if (AttackTimer > 0)
            {
                AttackTimer -= DeltaTime;
            }

            if (GetDistanceTo(TargetCharacter) > 1500.0f)
            {
                bHasTarget = false;
                EnemyState = EEnemyUnitState::SE_Patrolling;
            }
        }

        void AEnemyUnit::FindTarget()
        {
```

```
        TArray<AActor*> CollectedActors;
        DetectionSphere->GetOverlappingActors(CollectedActors);

        int32 FoundCharacters = 0;

        for (int32 iCollected = 0; iCollected < CollectedActors.Num(); iCollected++)
        {
                AFPStutorialCharacter* const TestActor = Cast<AFPStutorialCharacter>(
                CollectedActors[iCollected]);

                if (TestActor && !TestActor->IsPendingKill())
                {
                        FoundCharacters++;

                        if (!bHasTarget)
                        {
                                TargetCharacter = TestActor;
                                bHasTarget = true;
                                EnemyState = EEnemyUnitState::SE_Hunting;
                        }
                }
        }

        if (FoundCharacters == 0)
        {
                bHasTarget = false;
                EnemyState = EEnemyUnitState::SE_Patrolling;
        }
}

void AEnemyUnit::Attack()
{
        if (ProjectileClass != NULL && EnemyWeapon != nullptr)
        {
                UWorld* const World = GetWorld();
                if (World != NULL)
                {
                        const FRotator SpawnRotation =
                        EnemyWeapon->GetComponentRotation();

                        const FVector SpawnLocation =
                        EnemyWeapon->GetComponentLocation() + FiringOffset;

                        AFPStutorialProjectile* NewProjectile =
                        World->SpawnActor<AFPStutorialProjectile>(ProjectileClass,
                        SpawnLocation, SpawnRotation);

                        NewProjectile->Tags.Add("EnemyProjectile");

                        // try and play the sound if specified
                        if (FireSound != NULL)
                        {
                                UGameplayStatics::PlaySoundAtLocation(this, FireSound,
                                GetActorLocation());
                        }
                }
        }
}
```

SetAIActive-funktiota tulemme käyttämään myöhemmässä vaiheessa, kun lisäämme voitolle ja epäonnistumiselle GameStatet. Patrol-funktio liikuttaa vihollista asetettua reittiä

pitkin. Reitti tehdään Unreal Editorissa hieman myöhemmin. Hunt-funktio pysäyttää vihollisen, ja kääntää sen kohti pelaajan hahmoa. FindTarget-funktio etsii pelaajan hahmoa DetectionSpheren sisältä. Attack-funktio ampuu pelaajaa kohti. Huomaa, että kun ammus on luotu pelialueelle, sille asetetaan Tag. Tällä saadaan varmistettua, että viholliset eivät voi vahingoittaa toisiaan, tai että pelaaja ei voi vahingoittaa itseään. Lisätään EnemyUnit.cpp:n alkuun vielä nämä:

```
#include "FPStutorialCharacter.h"
#include "FPStutorialProjectile.h"
#include "FPStutorialGameMode.h"
```

Nyt kun meillä on vihollisen koodi valmiina, mennään Unreal Editorin puolelle tekemään viimeistelyt.

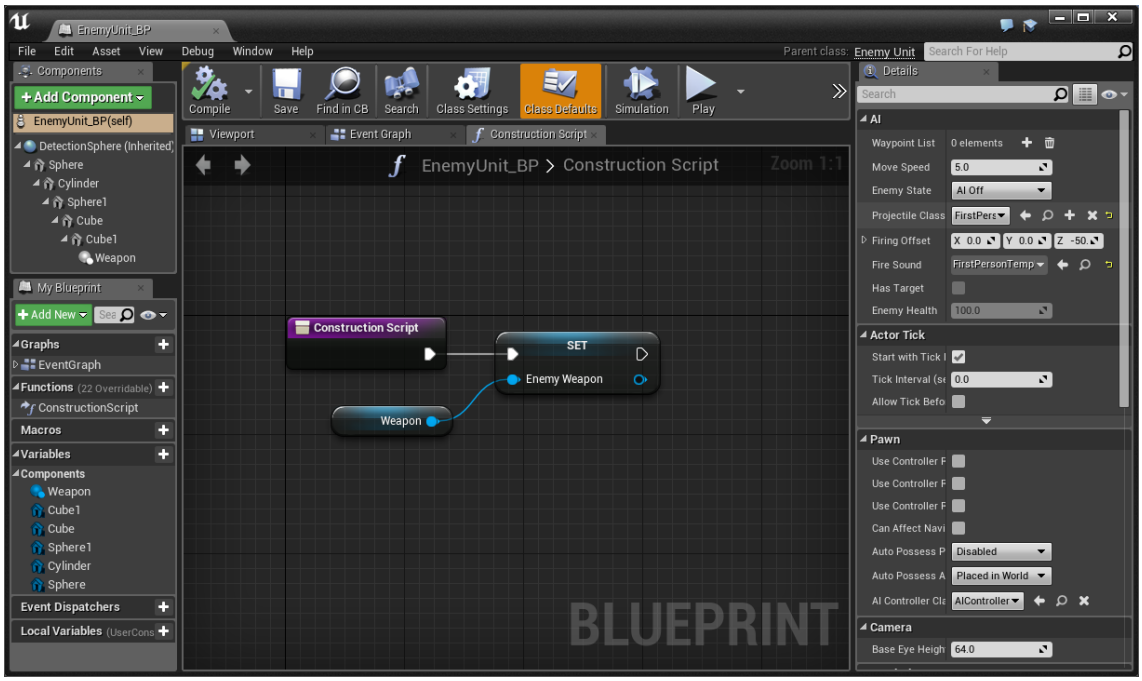
3.3.3 Vihollisen C++-luokasta Blueprintiksi

Tehdään EnemyUnitista Blueprint ja nimetään se EnemyUnit_BP. Avaa EnemyUnit_BP Full Blueprint Editor -moodissa. Ikkunan keskikohdalta löytyy Viewport-välilehti, jossa vihollisen 3D-malli näkyy. Tällä hetkellä vihollisella ei ole 3D-mallia, vain DetectionSphere näkyy punaisina viivoina. Vasemman laidan Add Component -napista voit lisätä viholliselle 3D-malleja. Käytä yksinkertaisia muotoja, kuten neliö ja pallo, ja tee vihollisesta sen näköinen kuin itse haluat. Huomaa, että eteenpäin suunta on -X. Raahaa EnemyUnit_BP pelialueelle ja valitse Details-paneelista EnemyStateksi Patrolling, niin näet miten vihollinen kääntyy, kun pelaajan hahmo tulee sen lähelle.

Kun olet saanut vihollisen näyttämään sellaiselta kuin haluat, lisää Add Component -napista vielä Scene-komponentti ja anna sen nimeksi WeaponRef. Tämä WeaponRef on apupiste, josta vihollinen ampuu, joten liikuta se sopivaan kohtaan. Nyt meidän tarvitsee kertoa EnemyUnitille, että WeaponRef on se piste, josta haluamme vihollisen ampuvan. Koodissa teimme sille muuttujan EnemyWeapon, eli meidän tarvitsee yhdistää WeaponRef EnemyWeaponiin. Se tapahtuu Blueprintin Construction Script -välilehdellä. Keskialueella ei näy muuta, kuin Construction Script -laatikko. Lähde vetämään sen laatikon Exec-pallukasta ja pudota se tyhjään kohtaan. Unreal Editor kysyy muuttujaa, jota

haluamme muokata, joten etsi listasta SetEnemyWeapon. Oikeassa reunassa näkyy Blueprintissä tehdyt muuttujat, nappaa tehty WeaponRef ja raahaa se keskialueelle. Raahaa WeaponRefin sininen pallukka EnemyWeaponin pallukkaan.

Nyt meillä on ampumiskohta, mutta vielä puuttuu itse ammus. Valitse vasemman puolen Component-paneelista EnemyUnit_BP(self). Oikean puolen Details-paneelista löytyy Projectile Class -muuttuja, johon tarvitsee valita FirstPersonProjectile. Voit myös lisätä ampumisäänen viholliselle valitsemalla jonkin äänen Fire Sound -muuttujan äänilistasta. Kuvassa 24 näkyy EnemyUnit_BP:n Construction Scriptin asettelu.



Kuva 24. EnemyUnit_BP:n Construction Script.

Tarkista vielä pelissä, että vihollinen ampuu oikeaan suuntaan ja käännä WeaponRefiä tarvittaessa. Tallenna muutokset. Vihollinen tällä hetkellä pysyy vain paikoillaan, eikä liiku, niin kuin haluaisimme. Teimme EnemyUnitin koodiin vihollisen liikkumaan ennalta määrättyä reittiä pitkin. Tämän reitin teemme reittipisteitä käyttämällä. Tähän tarkoitukseen on erinomainen objekti nimeltään TargetPoint. Se on käytännössä pelkkä apupiste, jolla ei ole mitään pelissä näkyvää 3D-mallia. Etsi Unreal Editorin vasemman puolen Modes-paneelista TargetPoint ja raahaa se pelialueelle. Asettele näistä sopivanlainen reitti viholliselle. Huomaa, että vihollinen kiertää reittiä ympäri, siis viimeisestä pisteestä ensimmäiseen. Kun se on tehty, valitse pelialueelta vihollinen, jonka haluat kulkevan tätä reitti pitkin. Unreal Editorin oikean puolen Details-paneelista löytyy koodaamamme AI-kohta ja sieltä WaypointList. Lisää siihen niin monta elementtiä, kuin tarvitset ja lisää

TargetPointit elementteihin. Huomaa, että vihollinen kulkee reittiä siinä järjestykessä, missä ne on laitettu WaypointListiin.

3.3.4 Pelaajan hahmon ja vihollisen välinen interaktiivisuus

Meillä on nyt vihollisia, jotka liikkuvat, ja ampuvat pelaajaan hahmoa, kun se tulee tarpeeksi lähelle. Viholliset eivät voi kuitenkaan vielä vahingoittaa pelaajaa, eikä pelaaja voi vahingoittaa vihollisia. Tarvitsemme siis jonkin asian, joka kertoo pelaajan hahmolle ja viholliselle, että siihen on osuttu. Käytetään tähän FPStutorialProjectilea. FPStutorialProjectile laskee osumia jo valmiiksi, joten sitä ei tarvitse muokata paljoa. FPStutorial.cpp:n OnHit-funktiossa käsitellään tapahtumat, kun ammus törmää johonkin. Lisätään sinne toiminnot, mitä tehdään, kun ammus törmää pelaajaan hahmoon tai viholliseen:

```
void AFPStutorialProjectile::OnHit(UPrimitiveComponent* HitComp, AActor*
OtherActor, UPrimitiveComponent* OtherComp, FVector NormalImpulse, const
FHitResult& Hit)
{
	// Only add impulse and destroy projectile if we hit a physics
	if ((OtherActor != NULL) && (OtherActor != this) && (OtherComp != NULL))
	{
		if (OtherActor->GetName().Contains("Enemy"))
		{
			AEnemyUnit* TestEnemy = Cast<AEnemyUnit>(OtherActor);

			if (Tags.Contains("PlayerProjectile"))
			{
				TestEnemy->UpdateHealth(-10.0f);
			}

			DeathEffect();
			Destroy();
		}
		else if (OtherActor->GetName().Contains("FirstPersonCharacter"))
		{
			AFPStutorialCharacter* TestCharacter =
			Cast<AFPStutorialCharacter>(OtherActor);

			if (Tags.Contains("EnemyProjectile"))
			{
				TestCharacter->UpdateHealth(-15.0f);
			}

			DeathEffect();
			Destroy();
		}
		else if (OtherComp->IsSimulatingPhysics())
		{
			OtherComp->AddImpulseAtLocation(GetVelocity() * 100.0f,
			GetActorLocation());
```

```
                DeathEffect();
                Destroy();
            }
        }
    }
```

FPStutorialProjectile.cpp:n alkuun tarvitsee vielä lisätä:

```
#include "EnemyUnit.h"
#include "FPStutorialCharacter.h"
```

Teimme aikaisemmin Tagin vihollisen ammuksille, nyt meidän tarvitsee tehdä sama pelaajan hahmon ammuksille. Avaa FPStutorialCharacter.cpp ja muuta rivi OnFire-funktiossa:

```
World->SpawnActor<AFPStutorialProjectile>(ProjectileClass, SpawnLocation,
SpawnRotation);
```

Näyttämään tältä:

```
AFPStutorialProjectile* NewProjectile =
World->SpawnActor<AFPStutorialProjectile>(ProjectileClass, SpawnLocation,
SpawnRotation);

NewProjectile->Tags.Add("PlayerProjectile");
```

Lisätään FPStutorialGameModeen vielä koodi, joka aktivoi kaikki pelialueella olevat viholliset pelin alkaessa. Koodi on hyvin samanlainen, kuin mitä teimme SpawnVolumelle. Avaa FPStutorialGameMode.h ja lisää rivi:

```
private:
TArray<class AEnemyUnit*> EnemyUnitActors;
```

Ja FPStutorialGameMode.cpp:n BeginPlay-funktioon:

```
UGameplayStatics::GetAllActorsOfClass(GetWorld(), AEnemyUnit::StaticClass(),
FoundActors);

for (auto Actor : FoundActors)
{
    AEnemyUnit* EnemyActor = Cast<AEnemyUnit>(Actor);

    if (EnemyActor)
    {
        EnemyUnitActors.AddUnique(EnemyActor);
    }
}
```

Sekä HandleNewState-funktioon:

```
case EGamePlayState::EPlaying:
{
	for (ASpawnVolume* Volume : SpawnVolumeActors)
	{
		Volume->SetSpawningActive(true);
	}

	for (AEnemyUnit* Enemy : EnemyUnitActors)
	{
		Enemy->SetAIActive(true);
	}
}
break;
```

Lisää vielä FPStutorialGameMode.cpp:n alkuun:

```
#include "EnemyUnit.h"
```

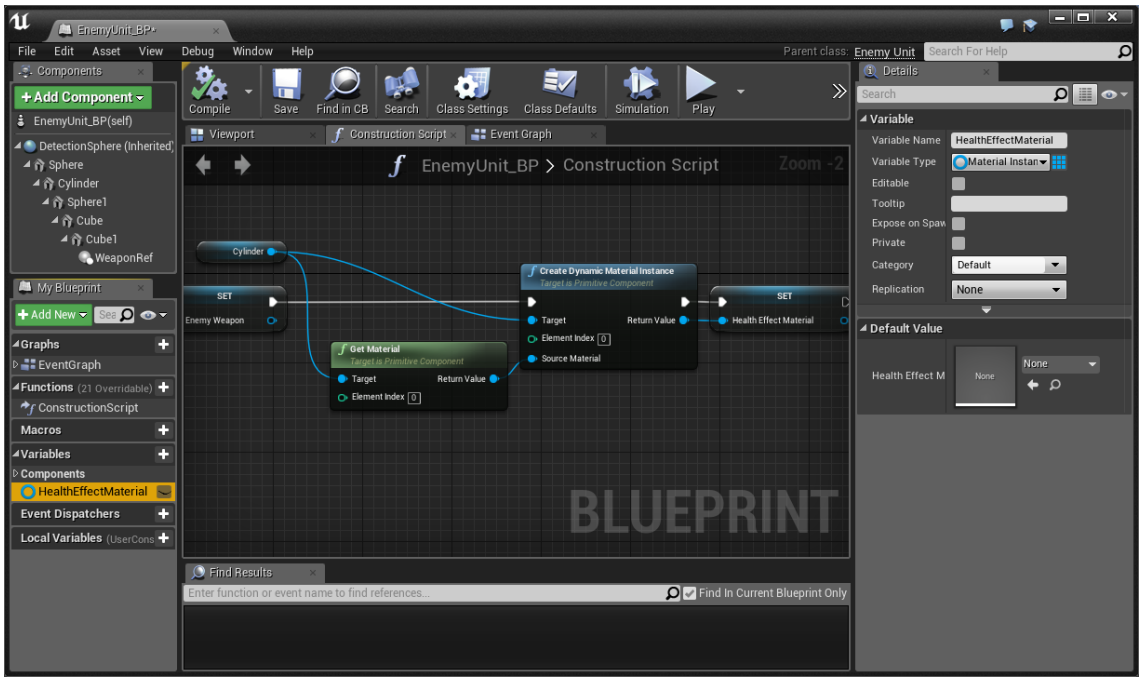
Tallenna muutokset ja Unreal Editorissa paina Compile-nappia. Nyt niin pelaajan hahmo, kuin vihollinenkin menettää healthia, kun siihen osuu. Voit kokeilla, että kun ammut vihollista tarpeeksi kauan, se katoaa pelialueelta, siis kuolee. Seuraavassa luvussa lisätään peliin efektejä, joiden avulla pelaaja huomaa muuttavansa pelimaailmaa.

3.4 Efektien lisääminen

Meillä on nyt pelissä vihollisia, jotka ampuvat, ja joita voi tuhota. Haluamme nähdä visuaalisesti, että olemme osuneet viholliseen, ja että se on menettänyt healthia. Toteutamme tämä lisäämällä viholliselle dynaamisen materiaalin. Dynaamisessa materiaalissa voi, esimerkiksi muuttaa sen väriä muuttujalla. Tässä tapauksessa muuttuja on vihollisen health määrä.

Avaa EnemyUnit_BP:n Construction Script -välilehti. Raahaa keskialueelle 3D-komponentti, jonka haluat ilmoittavan vihollisen kunnosta. Itse valitsen Cylinderin. Raahaa Cylinderin pallukka tyhjään kohtaan ja etsi listasta CreateDynamicMaterialInstance. DynamicMaterialInstance tarvitsee meshin lisäksi SourceMaterialin. Raahaa Cylinderin pallukkaa uudelleen tyhjälle alueelle ja etsi GetMaterial. Raahaa GetMaterialin ReturnValue

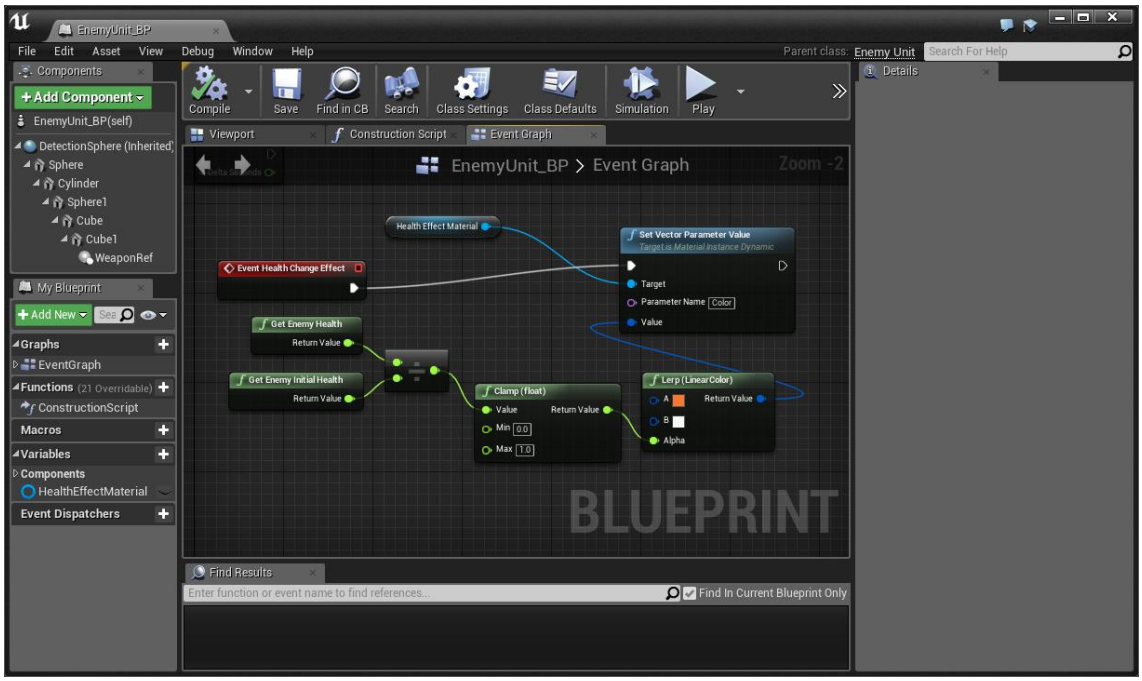
pallukka SourceMaterialin pallukkaan. Lisäksi tarvitsemme uuden dynaamisen materiaalin. Ikkunan vasemman puolen Variables-kohtaan lisää uusi DynamicMaterialInstance painamalla plus-nappia. Anna muuttujan nimeksi HealthEffectMaterial ja raahaa se keskialueen tyhjään kohtaan ja valitse Set. Nyt raahaa CreateDynamicMaterialInstancen ReturnValue HealthEffectMaterialiin ja raahaa EnemyWeaponin Exec-pallukka DynamicMaterialInstancen pallukkaan ja siitä eteenpäin HealthEffectMaterialiin. Tuloksen pitäisi näyttää kuvan 25 mukaiselta.



Kuva 25. EnemyUnit_BP:n Construction Script täydennettynä.

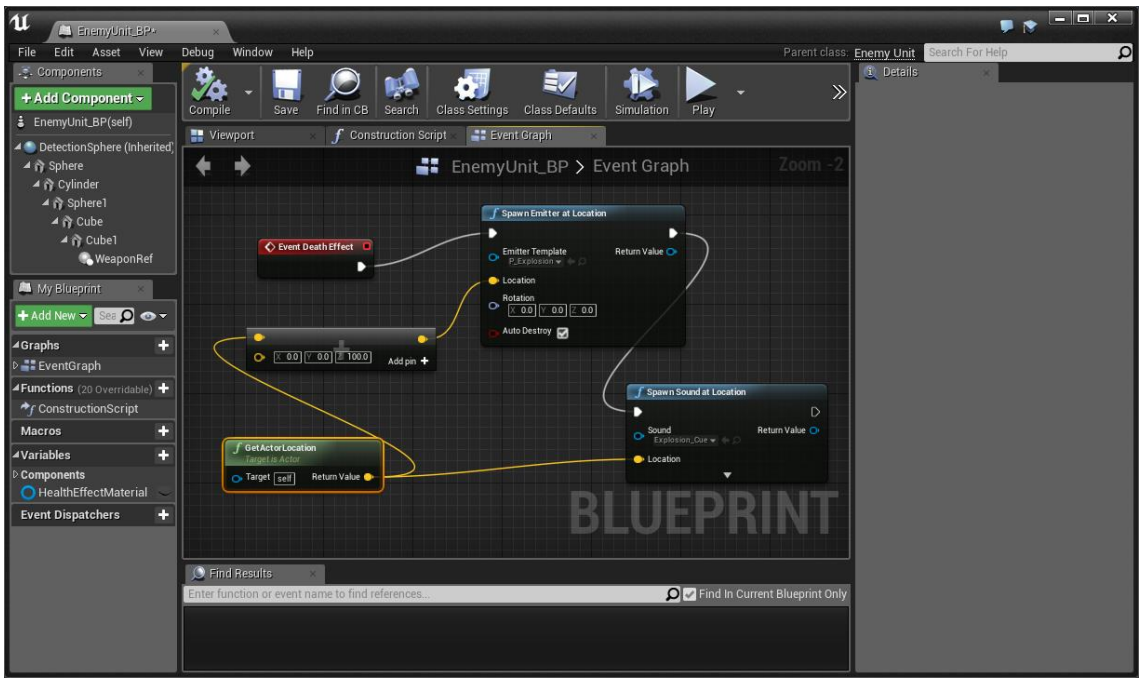
Nyt meillä on dynaaminen materiaali, mutta vielä tarvitsee tehdä toiminto mitä sille tapahtuu, kun pelaaja osuu viholliseen. Se tehdään EnemyUnit_BP:n Event Graph -välilehdellä. Raahaa HealthEffectMaterial-muuttuja keskialueelle ja valitse Get. Raahaa HealthEffectMaterialin pallukka tyhjään kohtaan ja etsi listasta SetVectorParameterValue. Muuta ParameterName-kohtaan Color. Tämä siis viittaa Cylinderissä olevan materiaalin parametrilaatikkoon, jonka nimi on Color. Minun Cylinder käytää BasicShapeMaterialia, jossa löytyy parametri nimeltä Color. Vielä tarvitaan laskutoimitus, joka laskee, kuinka paljon vihollisella on healthia verrattuna lähtöarvoon. Paina hiiren oikealla napilla tyhjässä kohdassa ja etsi GetEnemyHeath- ja GetEnemyInitialHealth-funktiot. Raahaa GetEnemyHealth pallukka tyhjään kohtaan, kirjoita div ja valitse float/float. Raahaa GetEnemyInitialHealth float/float:in alempaan pallukkaan. Raahaa float/floatin ReturnValue ja etsi Clamp(float). Raahaa Clampin ReturnValue ja etsi Lerp(LinearColor). Laita tähän haluamasi värit. B-kohdan väri on vihollisen hyvä kunto ja A-kohta on vihollisen huono

kunto. Sitten Raahaa Lerpin ReturnValue SetVectorParameterValuen Value-pallukkaan. Viimeisenä lisää ikkunan vasemmalta puolelta Graphs-kohdasta Event Health Change Effect ja raahaa sen pallukka SetVectorParameterValueeseen. Lopputuloksen pitäisi näyttää kuvan 26 mukaiselta.



Kuva 26. EnemyUnit_BP:n Event Graph.

Compile, Save ja testaa miltä vihollinen nyt näyttää, kun sitä vahingoittaa. Eikö olisi hienompaa jos vihollinen räjähtäisi kuollessaan? Meillä on jo valmiina räjähdysefekti, joten meidän vaan tarvitsee käyttää sitä vihollisen kuoleman hetkellä. Tätä varten olemme jo koodanneet vain Blueprintissä toteutettavan DeathEffect-funktion. Avaa EnemyUnit_BP:n Event Graph takaisin. Paina hiiren oikealla napilla tyhjässä kohdassa ja etsi listasta EventDeathEffect. Raahaa sen pallukka tyhjään kohtaan ja etsi listasta SpawnEmitterAtLocation. Raahaa lähtevä valkoinen pallukka tyhjään kohtaan ja etsi listasta SpawnSoundAtLocation. Paina hiiren oikealla tyhjässä kohdassa ja etsi GetActorLocation. Raahaa sen ReturnValue-pallukka suoraan SpawnSoundAtLocationin Location-pallukkaan. Raahaa GetActorLocationin ReturnValue uudelleen tyhjään kohtaan, kirjoita add vector ja valitse vector+vector. Laita vector+vectorin Z-kohtaan sopiva määrä. Tämä nostaa räjähdyksen paikkaa ylemmäs. Valitse SpawnEmitteriin ja SpawnSoundiin sopivat efektit. Itse valitsin räjähdykseksi P_Explosion:in ja ääneksi Explosion_Cue. Lopputuloksen pitäisi näyttää kuvan 27 mukaiselta.



Kuva 27. EnemyUnit_BP:n efekti.

Compile, Save ja testaa miltä näyttää, kun vihollinen tuhoutuu. Seuraavassa luvussa lisätään peliin palautteita pelaajalle, kuten näytetään kuinka paljon pelaajalla on healthia ja ammuksia.

3.5 Palaute pelaajalle

Lisätään peliin pelaajalle näkyviin health- ja ammuspalkit, joista pelaaja näkee sen hetkisen health- ja ammusmäärän. Lisätään peliin myös alkuvalikko, sekä voitto- ja epäonnistumisehdot. Aloitetaan lisäämällä HUD.

3.5.1 HUD:in lisääminen

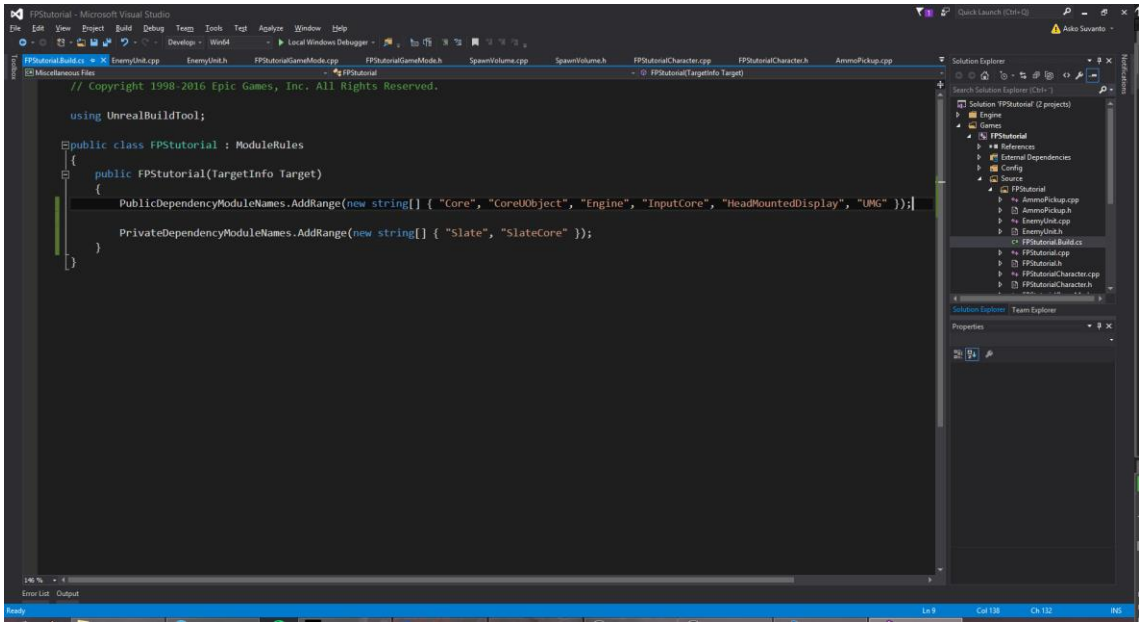
Unreal Engine käyttää UMG:tä, jonka avulla peliin voi lisätä Widget:ejä, jotka voivat koostua esimerkiksi napeista ja palkeista. UMG pitää lisätä projektiin muokkaamalla projektin build-tiedostoa. Build-tiedosto löytyy samasta source-kansiosta, jossa kaikki tekemämme C++-luokkien tiedostot sijaitsevat. Minun tapauksessani build-tiedosto löytyy paikasta:

FPStutorial\Source\FPStutorial\FPStutorial.Build.cs

Tämän tiedoston näkyy myös Visual Studion Solution Explorerissa. Avaa FPStutorial.Build.cs ja muokkaa sitä seuraavasti:

```
using UnrealBuildTool;

public class FPStutorial : ModuleRules
{
    public FPStutorial(TargetInfo Target)
    {
        PublicDependencyModuleNames.AddRange(new string[] { "Core",
        "CoreUObject", "Engine", "InputCore", "HeadMountedDisplay", "UMG" });

        PrivateDependencyModuleNames.AddRange(new string[] { "Slate",
        "SlateCore" });
    }
}
```

Kuvassa 28 näkyy FPStutorial.Build.cs:n koodi kokonaisuudessaan.



Kuva 28. FPStutorial.Build.cs.

Tarvitsemme UMG:hen vielä Unreal Enginen täyden kirjaston, joten avaa FPStutorial.h ja muuta rivi:

```
#include "EngineMinimal.h"
```

näyttämään tältä:

```
#include "Engine.h"
```

Tämä antaa käyttöömme täyden Unreal Engine -kirjaston, mutta se voi hidastuttaa Compilea. Seuraavaksi avaa FPStutorialGameMode.h ja lisää sinne uusi Widget:

```
protected:
UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category = "Health", meta =
(BlueprintProtected = "true"))
TSubclassOf<class UUserWidget> HUDWidgetClass;

UPROPERTY()
class UUserWidget* CurrentWidget;
```

Lisätään myös Widgetin tarvitsemat funktiot ja muuttujat:

```
public:
UFUNCTION(BlueprintPure, Category = "Health")
float GetPlayerMaxHealth() const;

UFUNCTION(BlueprintPure, Category = "Ammo")
float GetPlayerMaxAmmo() const;

protected:
UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category = "Health", meta =
(BlueprintProtected = "true"))
float PlayerMaxHealth;

UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category = "Ammo", meta =
(BlueprintProtected = "true"))
float PlayerMaxAmmo;
```

Lisää FPStutorialGameMode.cpp:hen lähdearvot muuttujille ja totetutukset funktiolle. Lisätään sinne myös uusi Widget ja asetetaan se pelinäkymään. Lisää FPStutorialGameMode.cpp:n konstruktoriin seuraavasti:

```
PlayerMaxHealth = 100.0f;
PlayerMaxAmmo = 100.0f;
```

Sitten lisää BeginPlay-funktioon uusi Widget:

```
if (HUDWidgetClass != nullptr)
    {
        CurrentWidget = CreateWidget<UUserWidget>(GetWorld(),
        HUDWidgetClass);

        if (CurrentWidget != nullptr)
        {
            CurrentWidget->AddToViewport();
        }
    }
```

Lisää myös funktiot:

```
float AFPStutorialGameMode::GetPlayerMaxHealth() const
{
	return PlayerMaxHealth;
}

float AFPStutorialGameMode::GetPlayerMaxAmmo() const
{
	return PlayerMaxAmmo;
}
```

Uusi Widget tarvitsee vielä kirjaston:

```
#include "Blueprint/UserWidget.h"
```

Itse Widgetin tekeminen tapahtuu Unreal Editorin puolella. Lisää Content Browserissa uusi User Interface -> Widget Blueprint. Anna sen nimeksi PlayerHUD. Avaa PlayerHUD. Huomaa ikkunan työkalurivillä Designer - Graph. Designerillä tehdään pelaajalle näkyvät elementit ja Graphissa tehdään niiden toiminnat. Etsi vasemman puolen Palette-paneelista Progress Bar ja raahaa se sopivaan kohtaan keskialueen neliön sisälle. Uusi palkki on hieman pieni, joten suurennetaan sitä sopivan kokoiseksi. Huomaa oikean puolen Details-paneelissa palkin tiedot näkyvät numeerisina. Anna palkin nimeksi HealthBar. Voit muuttaa palkin väriä Details-paneelin Fill Color -kohdassa. Lisää sitten toinen samaan tyyliin ja anna sen nimeksi AmmoBar.

Kun sinulla on palkit haluamallasi tavalla, valitse HealthBar ja etsi Details-paneelista Progress -> Precent. Numeerisen arvon vieressä on Bind-nappi. Paina sitä ja valitse Create Binding. Ikkuna avaa Graphin, jossa voimme tehdä toiminnon HealthBarille. Tähän tarkoitukseen tarvitsemme tietää pelaajan nykyisen healthin ja sen maksimiarvon. Paina hiiren oikealla napilla tyhjässä kohdassa ja etsi GetPlayerCharacter. Raahaa sen pallukka tyhjään kohtaan ja etsi CastToFistPersonCharacter. Paina hiiren oikealla napilla sen kohdalla ja valitse listasta Convert to pure cast. Tämä poistaa Exec-pallukat, koska niitä ei tässä tapauksessa tarvita. Raahaa AsFirstPersonCharacter-pallukka ja etsi GetCurrentHealth. Raahaa sen ReturnValue ja etsi Division (float/float). Raahaa sen ReturnValue ReturnNoden ReturnValueeseen. Sen jälkeen lisää GetGameMode. Raahaa sen ReturnValue ja etsi CastToFPStutorialGameMode. Muuta se pure castiksi. Raahaa

AsFPStutorialGameMode ja etsi GetPlayerMaxHealth. Raahaa vielä sen ReturnValue Divisioniin. Kuvassa 29 näkyy HealthBarin toiminnot. Nyt HealthBar muuttuu sen mukaan, kuinka paljon pelaajan hahmolla on healthia jäljellä.

Kuva 29. HealthBarin toiminnot.

Tehdään AmmoBarille samat toiminnot, paitsi että GetCurrentHealth muutetaan GetCurrentAmmoAmountiksi. Samoin GetPlayerMaxHealth muutetaan GetPlayerMaxAmmoksi.

Nyt meillä on toimiva HUD. Jotta sen saa näkymään pelissä, meidän tarvitsee ensin tehdä FPStutorialGameModesta Blueprint. Se onnistuu samalla tavalla, kuten aikaisemminkin. Anna sille nimeksi GameMode_BP. Avaa GameMode_BP jos se ei ole vielä avattuna. Details-paneelissa on koodaamamme Health-kohta, jossa on PlayerMaxHealth. Sen alla on HUDWidgetClass johon lisätään tekemämme PlayerHUD. Meidän pitää vielä kertoa Unreal Editor:ille, että haluamme käyttää uutta GameMode_BP:tä. Valitse Unreal Editor:in työkaluriviltä Blueprints ja valitse sieltä GameMode -> Select GameMode Class -> GameMode_BP. Nyt pelissä näkyvät health- ja ammuspalkit.

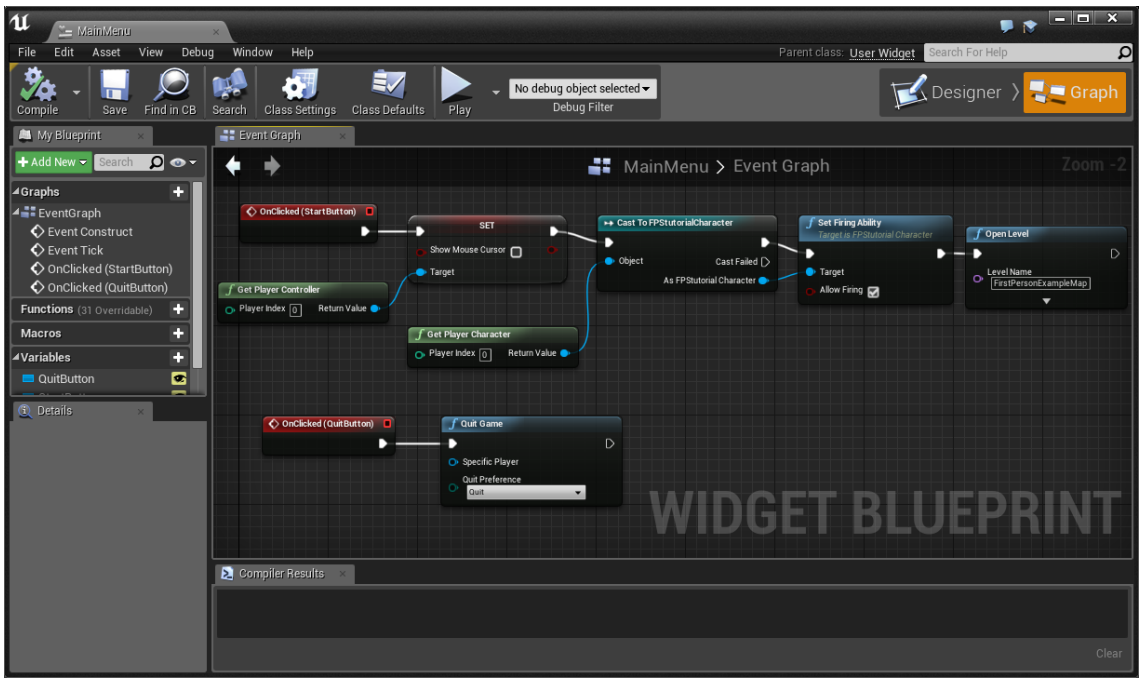
3.5.2 Valikkojen lisääminen

Tehdään pelille alkuvalikko. Varmista, että olet tallentanut kaikki Blueprintit ja pelikentän. Mene Content Browserissa Maps-kansioon ja lisää sinne uusi Level. Anna sen nimeksi Menu ja avaa se. Menu on ihan tyhjä kenttä, jonka voimme jättää sellaiseksi. Lisätään siihen vain HUD samaan tyyliin kuin aikaisemminkin.

Eli tee uusi User Interface -> Widget Blueprint. Anna sille nimeksi MainMenu. Avaa MainMenu ja lisää kaksi Buttonia, sekä niille kummallekin Text-elementti. Anna ensimmäisen Buttonin nimeksi StartButton ja toisen Buttonin nimeksi QuitButton. Säädä Textien Details-paneelissa Content- ja Appearance-kohdat sopiviksi. Lisää myös alkuteksti, kuten pelin nimi tai main menu. StartButtonilla on Details-paneelissa Events-kohta. Haluamme, että nappi tekee jotakin, kun sitä painetaan. Lisää siis OnClicked-toiminto.

Aloitetaan lisäämällä GetPlayerController. Raahaa ReturnValueta ja etsi SetShowMouseCursor. Varmista, että ShowMouseCursorin kohta on rastittamatta, koska haluamme piilottaa hiiren kursorin. Seuraavaksi lisää GetPlayerCharater. Raahaa ReturnValue ja etsi CastToFPStutorialCharater. Raahaa AsFPStutorialCharacter ja etsi SetFiringAbility. Varmista, että AllowFiring on rastitettu, koska haluamme pelaajan pystyvän ampumaan päästyään pelin varsinaiseen kenttään. Lisää viimeisenä OpenLevel ja kirjoita LevelName-kohtaan FirstPersonExampleMap. Raahaa OnClicked(StartButton) pallukka SetShowMouseCursorin pallukkaan. Ja siitä eteenpäin CastToFPStutorialCharacteriin. Ja sitten SetFiringAbilityn, ja lopulta OpenLevelin pallukkaan.

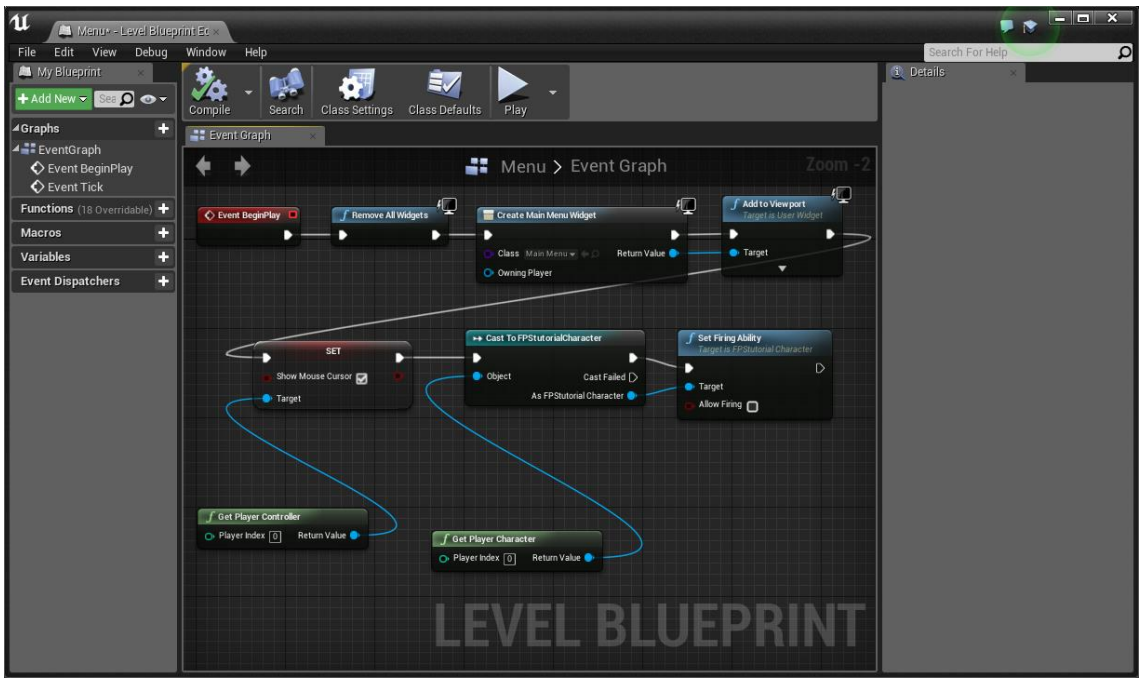
Mene Designerin puolelle ja lisää QuitButtonille OnClicked-toiminto. Graphissa lisää QuitGame ja varmista, että QuitPreference on Quit. Raahaa OnClicked(QuitButton) pallukka QuitGamen pallukkaan. Lopputuloksen pitäisi näyttää kuvan 30 mukaiselta.



Kuva 30. MainMenun nappien toiminnot.

MainMenu Widget pitää vielä lisätä Menu-kenttään samaan tyyliin, kuten aikaisemmin teimme FPStutorialGameModeen. Tällä kertaa käytämmekin siihen Level Blueprintia. Level Blueprint on kyseisen kentän globaali Blueprint. Sitä päästään muokkaamaan valitsemalla Unreal Editorin työkaluriviltä Blueprints, ja sieltä Open Level Blueprint.

Lähdetään liikkeelle EventBeginPlaystä. Raahaa sen pallukkaa ja lisää ensin RemoveAllWidgets. Tämä jälkeen voimme laitta uuden Widgetin lisäämällä CreateWidget. Aseta sen Class-kohtaan tekemämme MainMenu. Liitetään tehty Widgetti peliin raahamalla CreateWidgetin ReturnValue ja lisäämällä AddToViewport. Seuraavaksi haluamme hiiren kursorin näkyviin valikossa, ja että pelaajan hahmo ei voi tällä hetkellä ampua. Tämä menee samalla tavalla, kuin mitä laitoimme MainMenu Widgetin napeille. Lopputuloksen pitäisi näyttää kuvan 31 mukaiselta.



Kuva 31. Menu-kentän Level Blueprint.

Tallenna muutokset ja testaa peliä. Start-napin pitäisi ladata kenttä ja aloittaa peli. Seuraavaksi lisäämme toiminnot, mitä tapahtuu, kun pelaaja on saanut tuhottua tarpeeksi vihollisia, tai kun pelaaja kuolee.

3.5.3 Voittoehtojen lisääminen

Tarvitsemme peliin vielä voittoehdon, eli mitä tapahtuu, kun pelaaja on saanut tuhottua tarpeeksi vihollisia. Tarvitsemme myös ehdon, että mitä tapahtuu, kun pelaaja kuolee. Meillä on jo osittain koodia tehty FPStutorialGameModeen, joten aloitetaan sieltä.

Avaa FPStutorialGameMode.h ja lisää sinne seuraavat muuttujat ja funktio:

```
public:
virtual void Tick(float DeltaSeconds) override;

UFUNCTION(BlueprintPure, Category = "Game States")
float GetCurrentScore() const;

UFUNCTION(BlueprintCallable, Category = "Game States")
void UpdateScore(float ScoreChange);

protected:
UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category = "Score", meta =
(BlueprintProtected = "true"))
float PlayerScoreGoal;
```

```
private:
float PlayerScore;
bool bReturningToMenu;
float LoadingDelay;
```

Tick-funktio pitää kirjaa, kuinka paljon healthia pelaajan hahmolla on jäljellä. Se myös laskee onko pelaaja saanut tarpeeksi pisteitä tuhoamalla vihollisia. Lisää FPStutorialGameMode.cpp:n konstruktoriin nämä:

```
PlayerScoreGoal = 50.0f;
LoadingDelay = 5.0f;
bReturningToMenu = false;
```

LoadingDelay antaa pelaajalle hetken huomata, että hän on voittanut tai hävinnyt pelin, ennen kuin alkuvalikko ladataan uudelleen. Lisää BeginPlay-funktioon rivi:

```
PlayerScore = 0;
```

Lisää Tick-funktio seuraavanlaisesti:

```
void AFPStutorialGameMode::Tick(float DeltaTime)
{
	Super::Tick(DeltaTime);

	if (!bReturningToMenu)
	{
		AFPStutorialCharacter* PlayerCharacter = Cast<AFPStutorialCharacter>(
		UGameplayStatics::GetPlayerPawn(this, 0));

		if (EnemyUnitActors.Num() <= 0)
		{
			SetCurrentState(EGamePlayState::EWon);
			bReturningToMenu = true;
		}
		else if (PlayerCharacter->GetCurrentHealth() == 0.0f)
		{
			SetCurrentState(EGamePlayState::EGameOver);
			bReturningToMenu = true;
		}
	}
	else
	{
		if (LoadingDelay <= 0)
		{
			UGameplayStatics::OpenLevel(this, "Menu");
		}
		else
		{
			LoadingDelay -= DeltaTime;
		}
	}
}
```

Seuraavaksi mennään muokkaamaan HandleNewState-funktiota, jossa mellä onkin jo valmiina paikka koodillemme. Muuta voittokohtaa näyttämään tältä:

```
case EGamePlayState::EWon:
{
	for (ASpawnVolume* Volume : SpawnVolumeActors)
	{
		Volume->SetSpawningActive(false);
	}

	for (AEnemyUnit* Enemy : EnemyUnitActors)
	{
		Enemy->SetAIActive(false);
	}
}
break;
```

Pelaaja kun on voittanut, voidaan laatikoiden luominen ja jäljellä olevien vihollisten liikkuminen pysäyttää. Muuta sitten epäonnistumiskohtaa näin:

```
case EGamePlayState::EGameOver:
{
	APlayerController* PlayerController =	UGameplayStatics::GetPlayerController(
	this, 0);

	if (PlayerController)
	{
		PlayerController->SetCinematicMode(true, false, false, true, true);
	}

	ACharacter* PlayerCharacter = UGameplayStatics::GetPlayerCharacter(this, 0);

	if (PlayerCharacter)
	{
		PlayerCharacter->GetMovementComponent()->SetJumpAllowed(false);

		AFPStutorialCharacter* Player = Cast<AFPStutorialCharacter>(
		PlayerCharacter);

		Player->SetFiringAbility(false);
	}

	for (ASpawnVolume* Volume : SpawnVolumeActors)
	{
		Volume->SetSpawningActive(false);
	}

	for (AEnemyUnit* Enemy : EnemyUnitActors)
	{
		Enemy->SetAIActive(false);
	}
}
break;
```

Epäonnistumiskohta eroaa voitosta siinä, että epäonnistuminen tarkoittaa pelaajan hahmon kuolemaa. Kuollut hahmo ei voi liikkua, taikka ampua. SetCinematicMode-funktio muuttaa pelaajan kotrolleja niin, että pelaaja ei voi liikuttaa hahmoaan, taikka ampua. Lisätään vielä toeutukset GetCurrentScore- ja UpdateScore-funktioille:

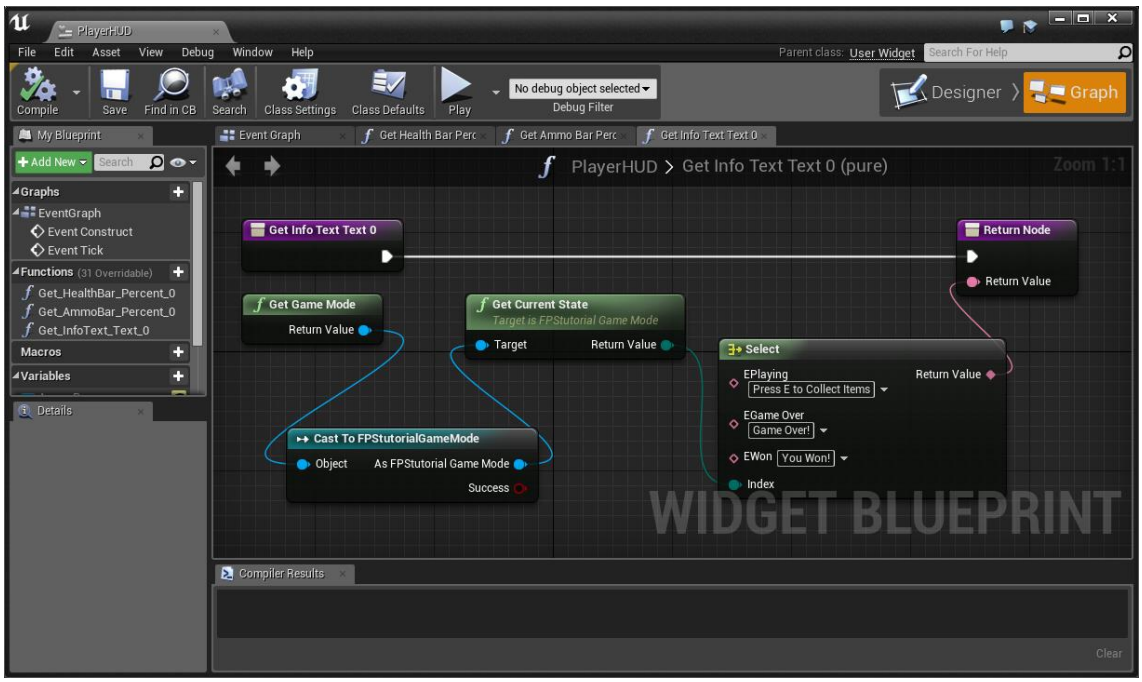
```
float AFPStutorialGameMode::GetCurrentScore() const
{
	return PlayerScore;
}

void AFPStutorialGameMode::UpdateScore(float ScoreChange)
{
	PlayerScore += ScoreChange;
}
```

Seuraavaksi avaa EnemyUnit.cpp ja mene muuttamaan UpdateHealth-funktiota seuraavasti:

```
else if (NewHealth <= 0)
{
	EnemyHealth = 0;

	AFPStutorialGameMode* GameMode = Cast<AFPStutorialGameMode>(
	UGameplayStatics::GetGameMode(this));

	GameMode->UpdateScore(50.0f);

	DeathEffect();
	Destroy();
}
```

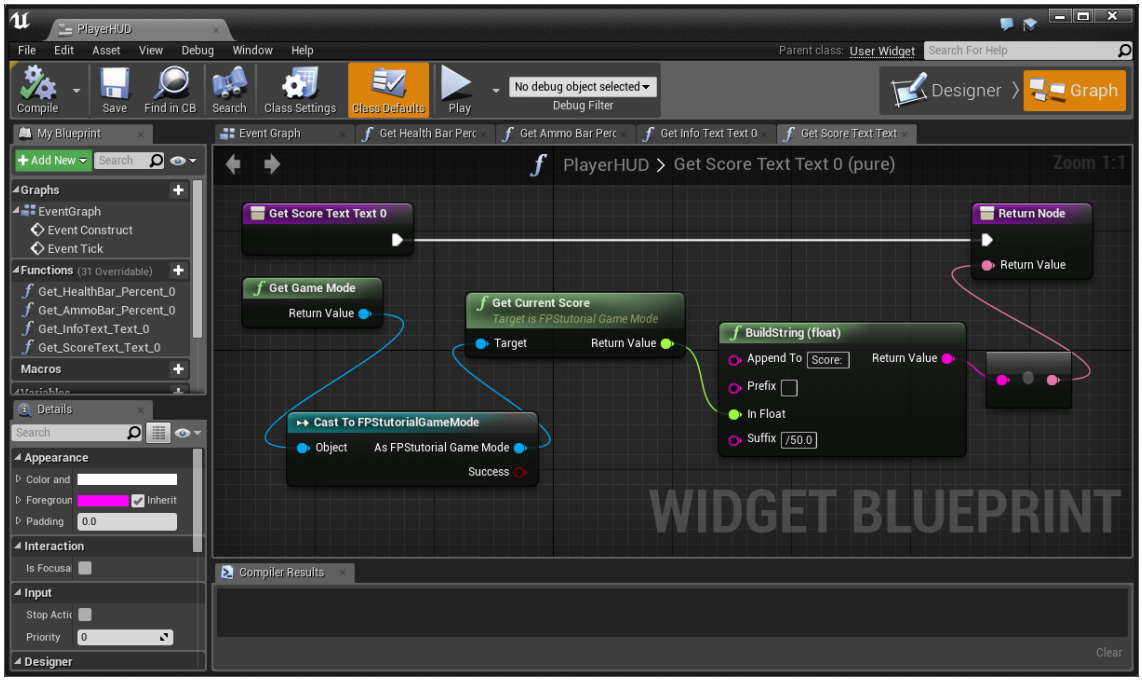
Nyt meillä on toimivat voitto- ja epäonnistumisehdot. Pelaaja ei kuitenkaan tiedä paljonko pisteitä on kerätty. Lisätään PlayerHUD:iin pisteet ja vähän infoa pelaamisesta.

Avaa PlayerHUD ja lisää kaksi Textiä. Anna niille nimet InfoText ja ScoreText. Valitse InfoText ja etsi Details-paneelista Context -> Text. Huomaa ruksata yllä oleva Size to Content. Paina Text-kohdan Bind-nappia ja valitse Create Binding. Lisää ensimmäisenä GetGameMode. Raahaa sen pallukkaa ja lisää CastToFPStutorialGameMode. Tee siitä pure cast. Raahaa AsFPStutorialGameMode-pallukkaa ja lisää GetCurrentState. Raahaa sen ReturnValue-pallukkaa ja lisää Select-elementti. Raahaa sen ReturnValue ReturnNoden ReturnValueeseen. Select valitsee siihen kirjoitetun tekstin kyseisen GameStaten aikana. Lopputuloksen pitäisi olla kuvan 32 mukainen.



Kuva 32. InfoTextin toiminnat.

Tee seuraavaksi ScoreText:ille Binding samalla tavalla. Ensin GetGameMode, sitten CastToFPStutorialGameMode ja viimeiseksi GetCurrentScore. Raahaa GetCurrentScoren ReturnValue-pallukkaa ja lisää BuildString(float). Siihen voi lisätä haluamasi teksin PlayerScoren lisäksi. Sen jälkeen raahaa sen ReturnValue-pallukka ReturnNoden ReturnValue-pallukkaan. Unreal Editor tekee automaattisesti tarvittavan muunnoksen. Lopputuloksen pitäisi olla kuvan 33 mukainen.



Kuva 33. ScoreTextin toiminnat.

Nyt olemme saaneet tehtyä alun toimivalle FPS-pelille. Seuraavassa luvussa katsotaan mitä pelille voi tästä eteenpäin tehdä ja mitä vielä tarvitaan, että peliä voi pelata ilman Unreal Editoria.

4 PELIN VIIMEISTELY

Nyt meillä on toimiva peli, josta löytyy mm. alkuvalikko, HUD, pisteenlasku ja yksinkertaisella tekoälyllä varustettu vihollinen. Seuraavaksi tehdään pelistä versio, jota voi pelata ilman Unreal Editoria.

Unreal Editor:issa on valmiina työkalu, jolla pelistä saa helposti tehtyä Standalonen. Sitä ennen meidän tarvitsee muokata projektin asetuksia hieman. Valitse Unreal Editorin työkaluriviltä Settings ja sieltä Project Settings. Project Settings ikkunassa löytyy Projectin alta Maps & Modes. Sieltä pääsemme valitsemaan minkä kentän peli lataa, kun se käynnistyy. Valitse Game Default Map -kohtaan Menu-kenttä. Default GameMode-kohtaan on jo valmiiksi valittu tekemämme GameMode_BP. Muita asetuksia meidän ei tarvitse muuttaa. Seuraavaksi valitse Unreal Editor -ikkunan File-valikosta Package Project ja sieltä alusta, jolla haluat peliä pelattava. Omassa tapauksessani valitsen Windows ja sieltä Windows (64bit). Unreal Editori kysyy minne haluat tallentaa pelin. Huomaa, että Unreal Editor pakkaa peliin kaiken sisällön, mitä projektissa on. Olemme käyttäneet tässä projektissa StarterContentia, jossa on aika paljon sisältöä. Pelin tekeminen Standaloneksi voi siis viedä runsaasti aikaa ja vaatia runsaasti tilaa.

Lopputulos on toimiva FPS-peli. Tai ainakin hyvä pohja, jolle on hyvä rakentaa peliä.

5 POHDINTA

Opinnäytetyö käsittelee Unreal Engine 4 -pelinkehitysalustaa pinnallisesti. Opinnäytetyössä keskitytään FPS-tyylisen pelin kehittämiseen ja sen tarjoamiin ongelmiin, sekä C++-ohjelmoinnin käyttöön Unreal Editor:in kanssa. Unreal Engine 4 -pelinkehitysalusta tarjoaa työkaluja monen muunlaisien pelien kehittämiseen, joita opinnäytetyössä ei käytetty.

Opinnäytetyön ohjetta seuraamalla aikaansaatu peli on hyvin pelkistetty ja sisällötön. Opinnäytetyö yrittää olla rajaamatta lukijan omaa luovuutta ja on siksi epätarkka ohjaamaan lukijaa luovuutta vaatimissa sisältöä lisäävissä kohdissa.

Toivottavasti opinnäytetyö lisäsi kiinnostusta Unreal Engine 4 -pelinkehitysalustaan ja pelinkehitykseen ylipäänsä. Lisää tietoa Unreal Engine 4:jästä löytää Unreal Engine:n internet sivustolta. Lisää ohjeita löytyy Unreal Engine 4 Documentaation internetsivulta. Ohjeita löytyy myös Epic Games Launcher:ista Learn-kohdasta. Ohjeita ja apua voi aina kysyä UE4 AnswerHub:in internet sivustolta.

LÄHTEET

Unreal Engine 4 Documentation. Luettu 25.1.2016.
https://docs.unrealengine.com/latest/INT/index.html

MSDN C++ Language Reference. Luettu 25.1.2016.
https://msdn.microsoft.com/en-us/library/3bstk3k5.aspx

Unreal Engine API Reference. Luettu 26.1.2016.
https://docs.unrealengine.com/latest/INT/API/index.html

UE4 AnswerHub. Luettu 26.1.2016.
https://answers.unrealengine.com/

First Person Shooter C++ Tutorial. Luettu 22.2.2016.
https://wiki.unrealengine.com/First_Person_Shooter_C%2B%2B_Tutorial

Introduction to C++ Programming in UE4. Luettu 22.2.2016.
https://docs.unrealengine.com/latest/INT/Programming/Introduction/index.html

C++ 3rd Person Battery Collector Power Up Game Tutorial. Luettu 22.2.2016.
https://docs.unrealengine.com/latest/INT/Videos/PLZlv_N0_O1gYup-gvJtMs-gJqnEB_dGiM4/mSRov77hNR4/index.html