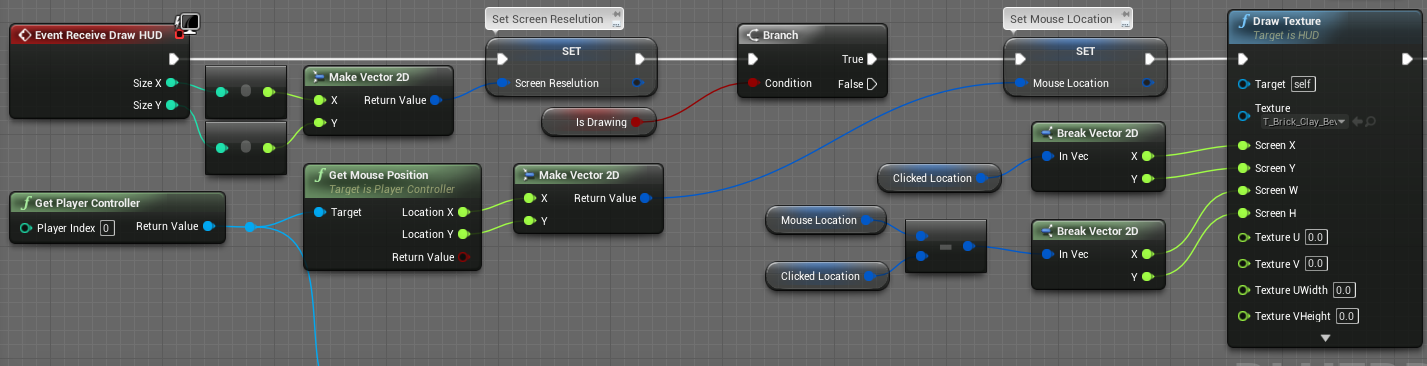
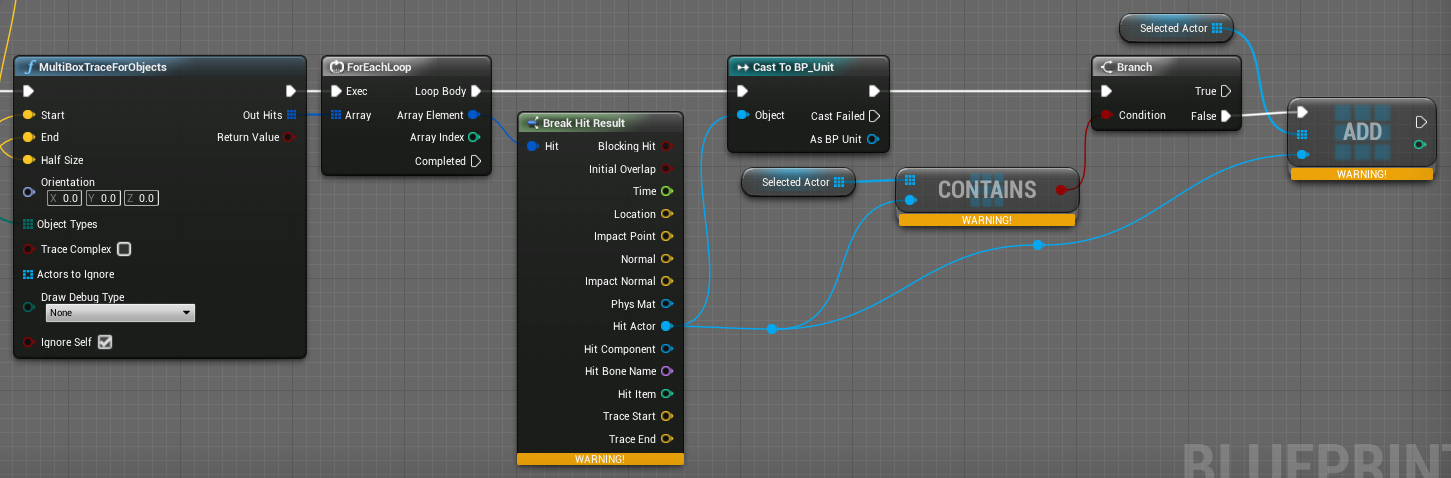
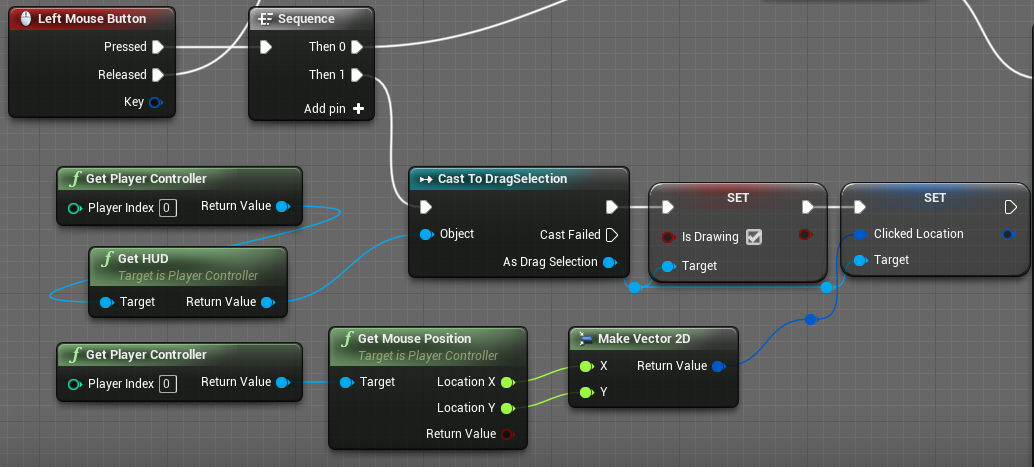
# Spil Ide

Vi vil under dette forløb forsøge, at lavet en multiplayer real time strategi. Målet er at 2 spiller skal være i stand til forbinde sig til et spil og kæmpe mod hinanden. Hver spiller skal være i stand til, at bygge en eller flere bygninger samt to eller flere enheder. For at en spiller skal kunne vinde over den anden så skal man ødelægge modstanderens central bygning som gives med i starten af spillet til begge spiller. Vi vil også forsøge, at lave passive enheder hvis opgave vil være at samle ressourcer og bygge bygninger, dog vil denne del være lav prioriter. Der er umiddelbart ingen planer om, at lave tech træ som vil gør det muligt at opgrader eller låse op for nye enheder.

# Multi og Single-enheds markering

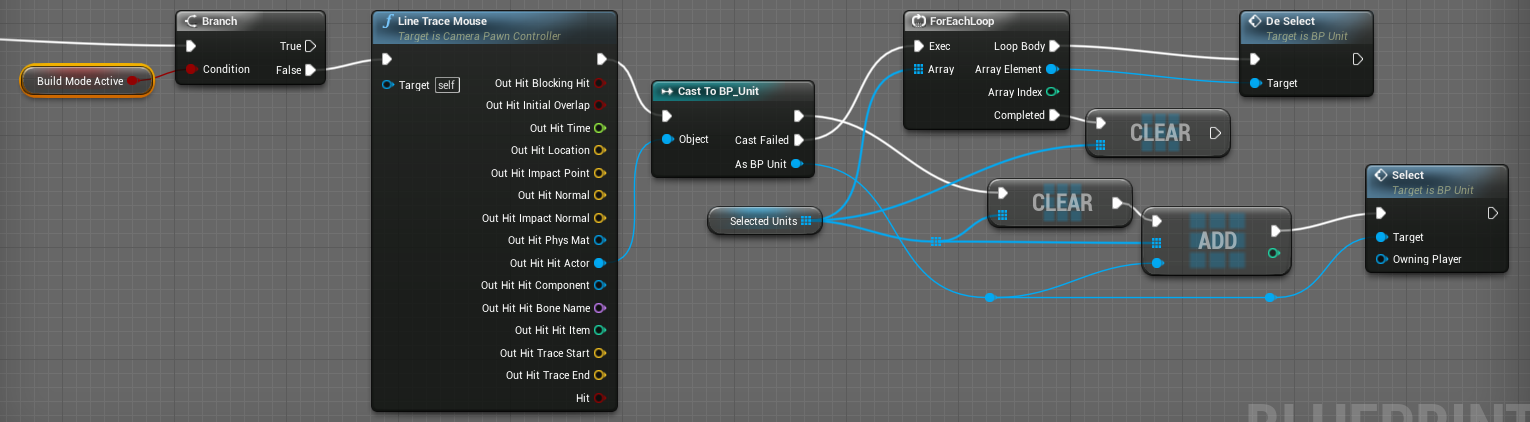
Dette er første del af, at kunne markere flere enheder. For at spilleren skal være i stand til at se hvilke enheder der bliver markeret, tegnes en boks på skærmen. Det første der gøres er, at kalde et nyt event som tillader os at tegne på skærmen, dette event har to integers, X og Y, som er spil vinduets størrelse. Dette konventer man om til en *Vector2D* som sørger for, at der tegnes inden for spil vindue, herefter bruges en *branch* som kører et tjek på variablen *IsDrawing*, på den måde kan man sørge for, at der kun tegnes imens at musen bliver holdt nede. Hvis der tegnes skal der bruges en lokation for hvor man startede med at tegne. Dette laves også om til en *Vector2D*. Vi skal endvidere også bruge den nuværende lokation for, at opdatere den tegnet firkant. *Screen X* og *Y* på metoden *Draw Texture* er der hvor teksturen, som skal lave boksen, starter, og det er her vi skal bruge vores *Clicked location*. *Screen H* er højden og *W* er bredden på teksturen, og disse udregnes ved at trække punktet hvor der klikkes med musen, *Clicked location*, fra musens nuværende position, *Mouse location*.

Når man så har tegnet bokse skal de enheder som er inde for boksen grænser identificeres. Dette gør vi ved at bruge metoden *MultiBoxTraceForObjects*. Denne metode finder alle objekter inden for dens grænser, og via den tilføjer man dem til et array, og hvis kun noget specifikt ønskes at markers kaster man det til den ønsket type.

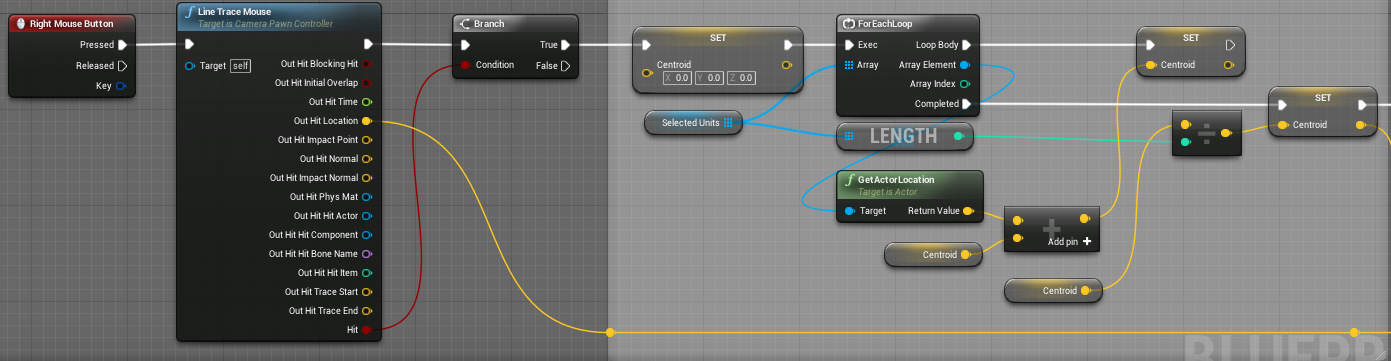
Når man så har de forrige metoder klar, kan man begynde at binde det til de ønskede input fra spilleren. Når der her trykkes på venstre museknap vil den kalde den nødvendige metode, heri sætter vi også de nødvendig variabler som *IsDrawing* og *Clicked location*.

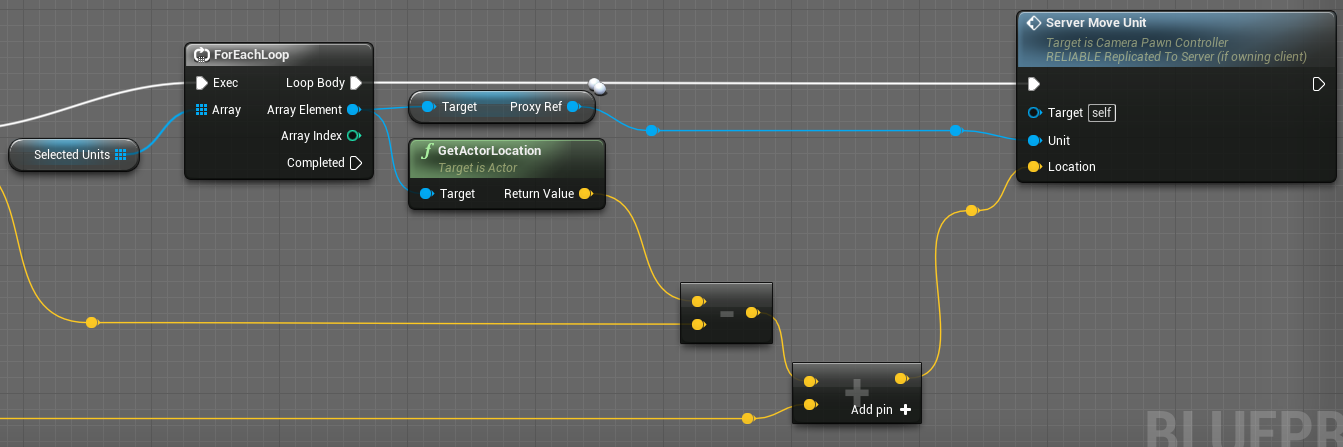


Når spillerne endelig giver slip på venstre museknap sker anden del af den forrige metode, IsDrawing bliver sat til at være *false* hvilket betyder, at boksen forsvinder. Samme tid med at boksen forsvinder vil de enheder som er inden for den grænse også blive tilføjet til et nyt array af enheder og det gamle array, som tilføjer vores markerings boks, vil blive tømt.

Det skal også være muligt for spilleren, at vælge enkelte enheder af gangen. Dette gør vi ved at se på hvor musen er og laver en linje ned gennem verden og ser på om en enhed bliver ramt af linjen. Når vi marker en enkelt enhed ønskes der også, at alle andre enhed ikke længer er markeret. Derfor tømmes vores array for enheder inden den enkelte enhed tilføjes. Der skal også gøres muligt, ikke at have enheder markeret, så nar man klikker et sted på skærmen, og det ikke rammer en unit vil castet fejle og arrayet tømmes for enheder.

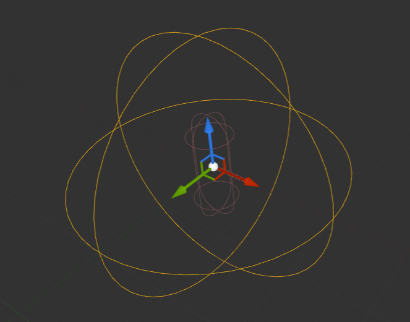
# Bevægelse

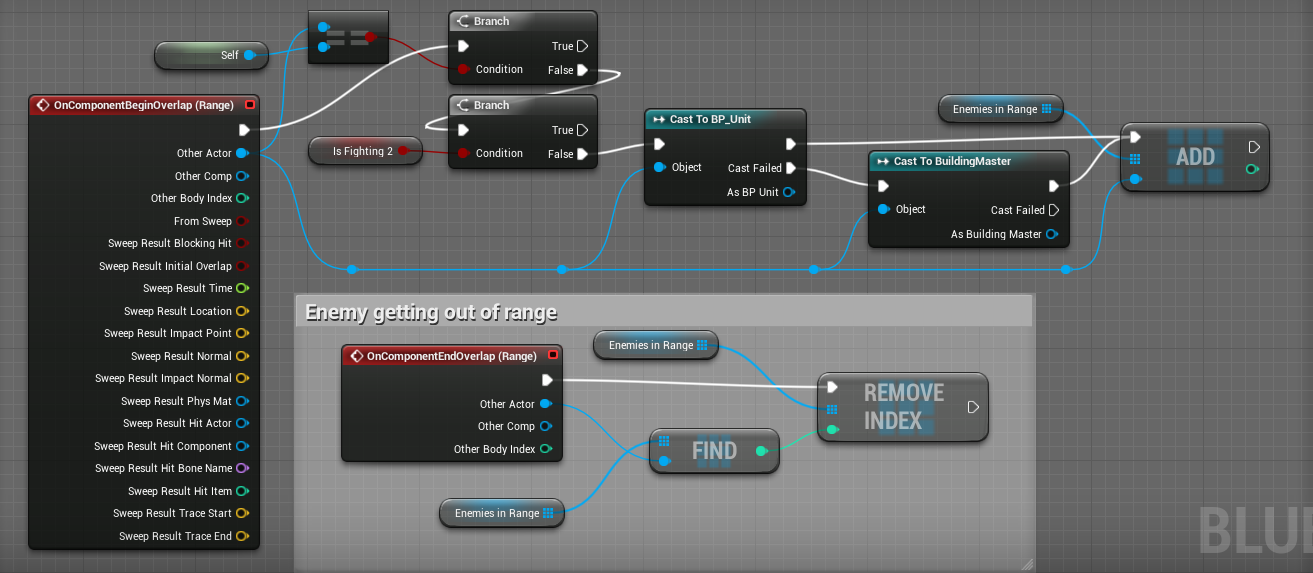
For at får spilleren til at bevæg sig bruges højre museknap. Når knappen trykkes bliver der lavet en linje fra kameraets position ned til det sted hvor der trykkes på skærmen. Hvis en eller flere enheder er markeret vil der blive sendt en besked til alle enheder om, at de har fået en ny lokation de skal bevæg sig til. Dette giver dog umiddelbart et problem, da alle enheder så vil prøve at stille sig på den præcise samme lokation, så derfor laves der et midtpunkt mellem alle de markerede enheder og dette gøres til der er et sammenligs punkt. Dette gøre ved at oprette en nyt variable, *centroid*, den sættes ved at loop igennem alle de markeret enheders lokationer og ligger dem sammen, og når den er færdig divider man det samlede resultat med antal enheder.

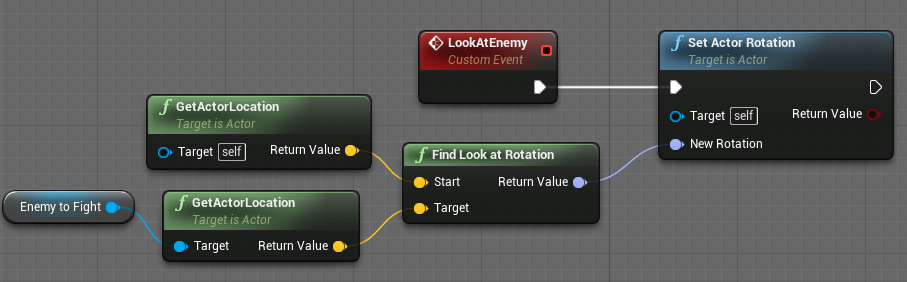
For så at få enheden til at bevæg sig i den korrekte retning trækkes værdien af *Centroid* fra aktørens nuværende lokation, derefter ligges den til den lokation som man får, når man trykker på skærmen. Når vi endelig har dette, kan man sende det til *ServerMoveUnit* som skal bruge denne information. Der sendes også en reference fra den proxy enhed som sørger for, at serveren opdaterer positionen for begge spiller. Dette bliver uddybet i afsnittet server meddelelser.

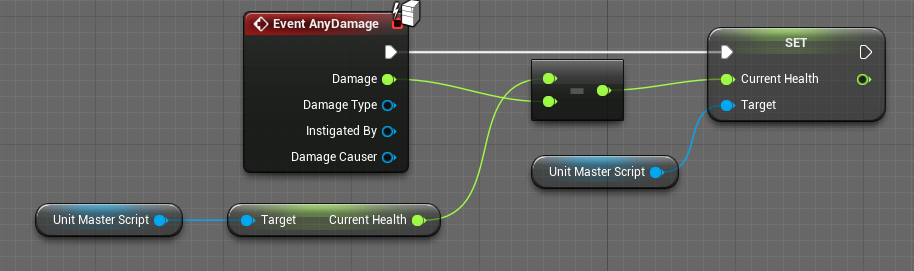
# Kamp mellem enheder

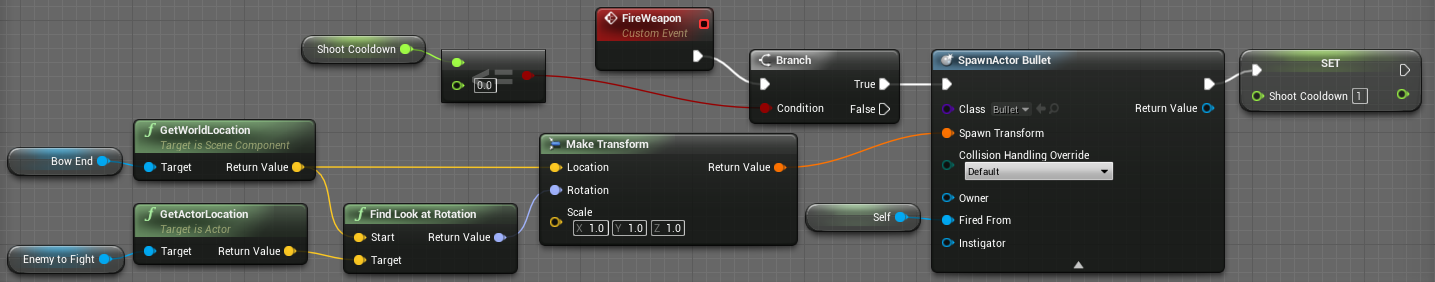
Når der laves kamp mellem enheder er der mange ting der skal tages i betragtning, og det er derfor også en af de mere komplekse del af spille. Der skal for eksempel tages højde for enhedens skudvidde, hvilken modstander den skal angribes og hvornår, finde en ny fjende når den gamle dør samt en række af andre ting.

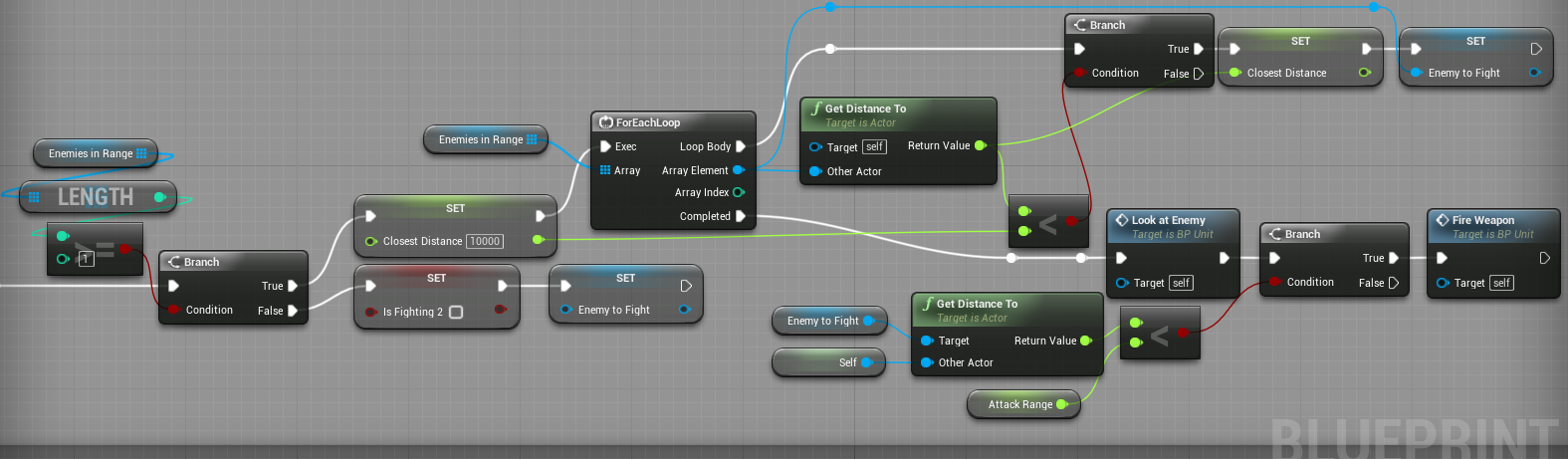
Det første der skal gøres er, at tjekke på om noget er inden for rækkevidde. Dette gøres ved at lave en kollisions cirkel på vores enheder. Man kan nu bruge denne cirkel til at se på om der er overlap mellem to enheders kolliders, og hvis den gør, der kan den angribes hvis det er en fjende.

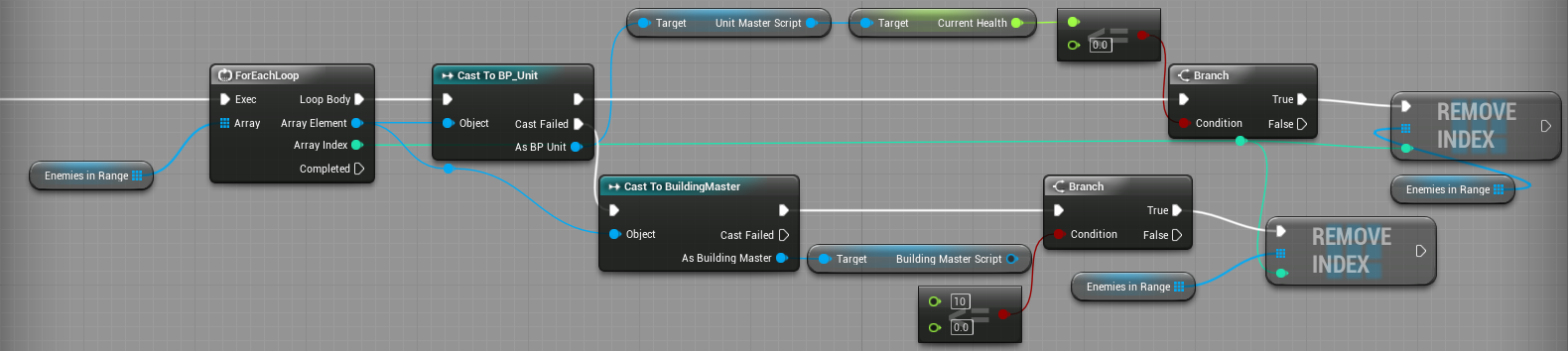
For at fjerne og tilføje enheder bruges en begivenhed som tilfører kollisions komponentet som hedder *On Component Begin Overlap* og *On Component End Overlap*. Når kollisionen begynder tjekkes der først om den kollider med sig selv, da dette er muligt, hvis kollisionen ikke sker med sig selv skal den kontrollere, om det den kollider med er en fra af dens eget hold. Når begge af disse statements er falske kan en enhed eller bygning tilføjes til et array af enheder i rækkevidde. Hvis *On Component End Overlap* bliver aktiveret findes den specifikke enhed og fjernes fra arrayet.

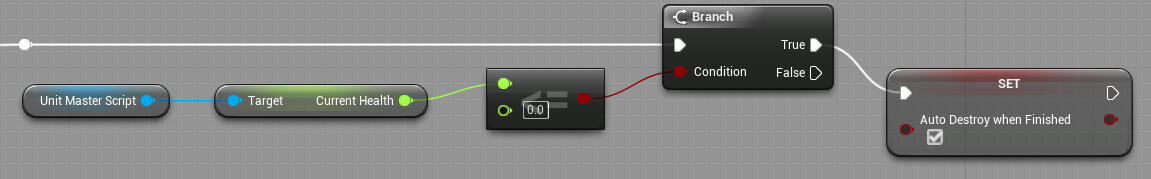
*Look at Enemy* er et event som sørger for, at den kigger på den enhed som den skyder på, hertil bruger den *set Actor Rotation*.

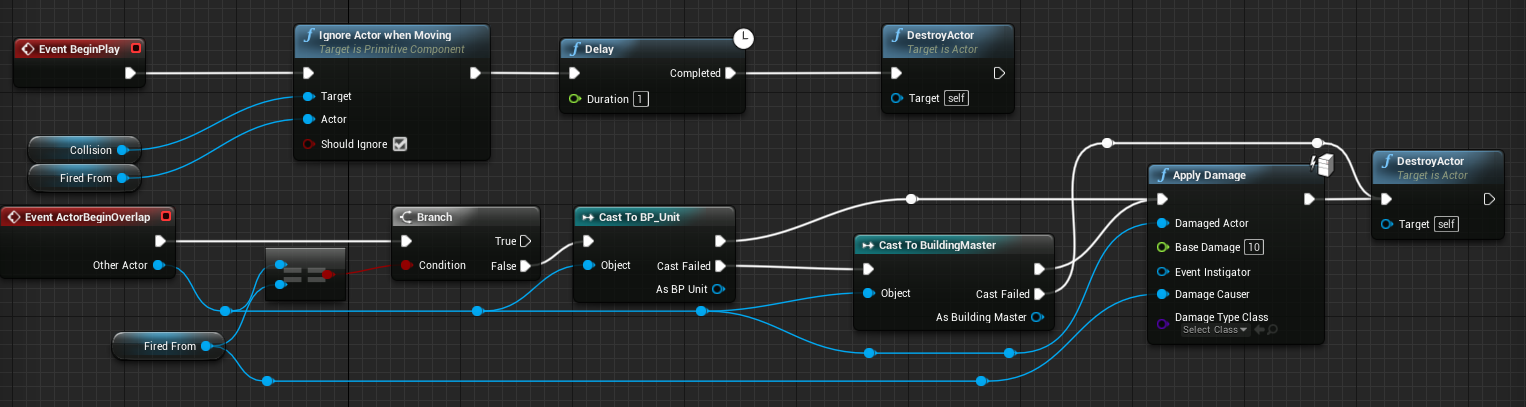
*Event anyDamage* bruges til at ændre vores *health* når den bliver ramt af noget som giver *damage*. Her hentes en reference fra *unit master scripts* som indeholder disse variabler.

Det sidste event som skal bruges inden det hele skal slås sammen er *fire Weapon*, denne bruges til at lave en ny instans af en kugle som skal affyres mod fjenden. Når den gøre dette tjekker den om dens nedkøling er færdig, dette gøres så der ikke bliver lavet nye kugler konstant. Når kuglen bliver lavet vil den automatisk bevæg sig i den retning den blev skud imod, derfor skal der bruges et *spawn transform* som kan bestem hvorfra, og i hvilken retning, den skal laves. Lokationen bestemmes af en komponent som ligger på enheden, og retning af den fjende den nu kigger på.

Nu hvor vi har de nødvendige metoder, kan der endelig laves noget funktionalitet for kamp. Det første der gøre er at tjekke på om der er noget at skyde på, så man tjekker arrayet igennem af fjendtlige enheder i rækkevidde, og vis dens længde er større end eller lige med 1, så må der være noget at skyde på. Når vi gøre dette vil vi finde den enhed der er tættes på ved at sammenligne afstanden fra den selv, og alle enheder i rækkevidde. Når den endelig har fundet enheden, der er tættest på, bruger den *Look at enemy* til at vende sig mod fjenden og begynder at skyde ved at bruge *Fire Weapon.*

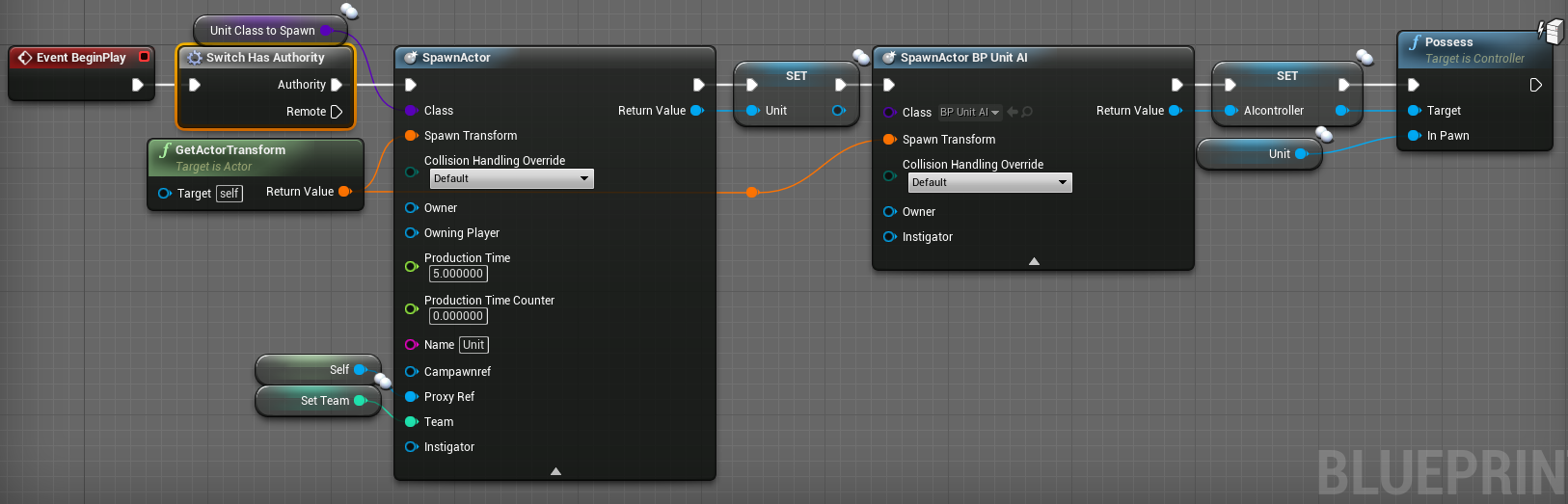
Nu mangles der bare at gøres to ting, at tjekke på om fjenden er død eller en bygning er ødelagt, og om man selv er død. Det der skal gøres er at tjekke på om fjenden er død. Dette gør vi ved at se alle enheder i vores array igennem, og se om ens liv er under eller lige med nul, og hvis dette argument er sand vil den specifikke enhed blive fjernet fra lisen af sete fjender. Det samme gøres på bygninger.

Til sidst er der en funktion som tjekker på enhedens egen liv. Hvis de kommer under eller er lige med 0, vil enheden blive fjernet fra spillet når den ikke længer har reference eller kald liggende. Dette sørger *Auto Destroy when Finished* for som er en del af *Unreal*. Grunden til at man ikke bare kan bruge *destroy* er, at der så vil være null referencer til den enhed.

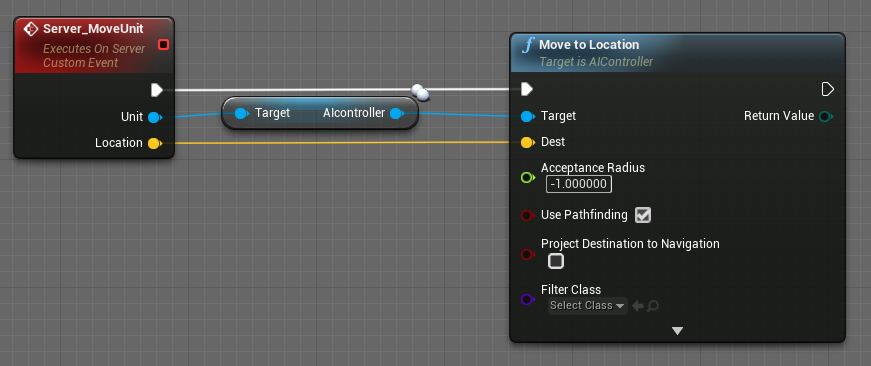
*Bullet* er også en essentiel del af kamp, og det objekt som betemmer hvor meget skade en enhed tager når de bliver ramt. Det første der skal sørges for er, at kuglerne ignorer den enhed den er affyret fra, da dette er en mulighed, skal kuglerne destrueres så de ikke bare hænger i spilverdenen og optager hukommelse. Dette gøres via et delay så den ikke destruer sig selv med de samme. Det forrige sker dog kun hvis kuglerne ikke rammer en enhed. For at tjekke om kuglerne har ramt en enhed har den en kollisions cirkel på sig, og med den kan man bruge *Actor Begin Overlap.* Igen skal der tjekkes om den kollider med den enhed den er skudt fra. Hvis dette argument er falsk, vil den give skade til enten en enhed eller en bygning alt efter hvad den rammer, hvis den rammer andet vil den også forsvinde.

# Serverside Meddelelser

I det at spillet er online multiplayer spil er det også vigtig at sørge for, at information som enheds positioner, antal, hvilke bygninger og hvem der er på bliver sendt frem og tilbage mellem spilleren og serveren. For at gøre dette bruges nogle *proxys* som kan replikker data fra klienterne og sende det videre.

Denne *proxy* har to opgaver, at tilføje vores enhed til verden, og give den en AI controller som skal sørge for, at enheden kan bevæge sig rundt i banen. Dette event begynder når spilleren bygger en enhed. Den starter med først et bestemme, at den kun er serveren der kan have kontrol over den ved at bruge *Switch has Authority*. Derefter spawner den en Actor og bruger *Unit Class to spawn* som er en variabel af *BP\_Unit*, her får den også tildelt et team og en reference til sig selv. Denne reference kaldes når man trykker for at bevæg enheden. For at proxyen kan finde den unit den tilhører sættes variablen *Unit* til at være den nu spawned actor her efter. Nu spawnes der en *BP\_Unit\_Ai* som er det blueprint som indeholder AI controlleren, og bliver igen sat over i en variabel. Denne variabel bliver sat til at overtage kontrollen af enheden.

Når proxyen er lavet kan den nu kaldes i *CameraPawnController*, den kan man bruge i en *Move to location*, som tager den tidligere AIContoller variabel og destinationen som bliver sat når man klikker.



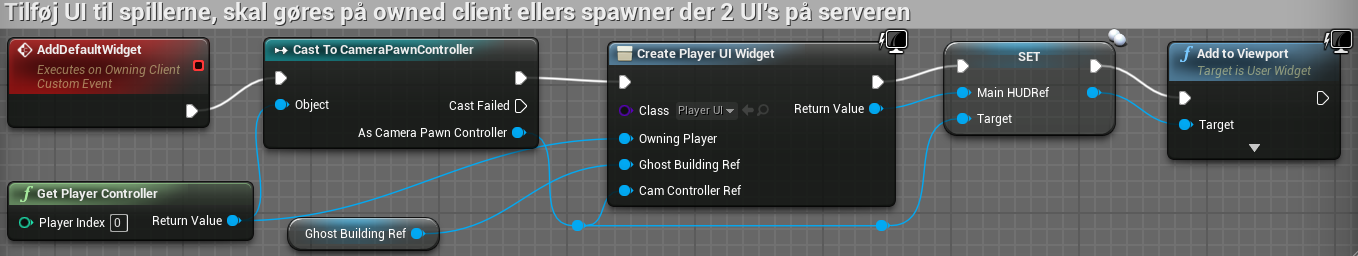
*Server Spawn unit* kaldes når der laves en unit, som før sætter serveren til at være den eneste der må gøre dette. Efter den har gjort dette spawner den en BP\_unit\_proxy som er beskrevet tidligere i afsnittet. Efter dette *castes* det hele til en BP\_unit og tilføjes til et array af ens egen enheder.

# 

Ud over server spawn unit er der også serverside for spawning af units, bygninger, tildeling af hold samt at flytte enheds position når den er færdig med at bygge. Disse ligge sig meger op af de andre og derfor er der valgt ikke at gentage der i dybden. Grundlæggende gøre de alle brug af en proxy som kun serveren kan bestem position og egenskaber for og derefter bindes en enhed eller en bygning til proxyen.

## Begin play i CameraPawnController

Dette event er rimlig langt så læser bedes se med i blueprintet[[1]](#footnote-1) i stedet for.

Til at starte med deaktiveres tick*[[2]](#footnote-2)* eventet efterfulgt af en branch*[[3]](#footnote-3)* hvor der tjekkes om player controlleren[[4]](#footnote-4) på index 0 er den samme som den controller man er i, altså *CameraPawnController*. Da det er sandt, køres der et custom event[[5]](#footnote-5) som sørger for at spillerne kommer på forskellige hold. Derefter deaktiveres kamera navigation midlertidigt og *tick* eventet aktiveres igen. Den går så ud i en sekvens med adskillige events. Det første event sætter klienternes referencer og får fat i den *pawn[[6]](#footnote-6)* der er bliver kontrolleret af player controlleren på index 0. Den returnerer så den *pawn* og kaster den til en *CameraPawn* og sætter så referencen. De to efterfølgende events sørger for at spawne en *Construction Manager* og en *GhostBuilding* aktør og sætter deres referencer, og det gøres ved, at en af eventsene *castes* til *CameraPawnController* som får player controlleren på index 0 med. Den kan begge sekvenser så bruge til at sætte deres referencer. Det næst sidste event sørger for, at der spawnes en kirke som er spillerens hovedbygning. Det sker ved hjælp af *ServerSpawnBuilding* eventet hvor der defineres hvilken bygning der skal bygges, hvor den skal bygges og hvilket hold man er på, og til sidst sættes kamera lokationen til det samme sted hvor ens hovedbygning bliver bygget. Til sidst er der en sekvens som håndterer user interfacet. Her er der en funktion som sætter nogle regler op for inputs i et UI[[7]](#footnote-7). Den får en reference til player controlleren og *Lock mouse to viewport* variablen bliver sat til *true* så musen bliver inden for viewporten[[8]](#footnote-8). Funktionen går ud i et custom event som tilføjer et UI til viewporten og til sidst aktiveres kamera navigation og *tick* igen.

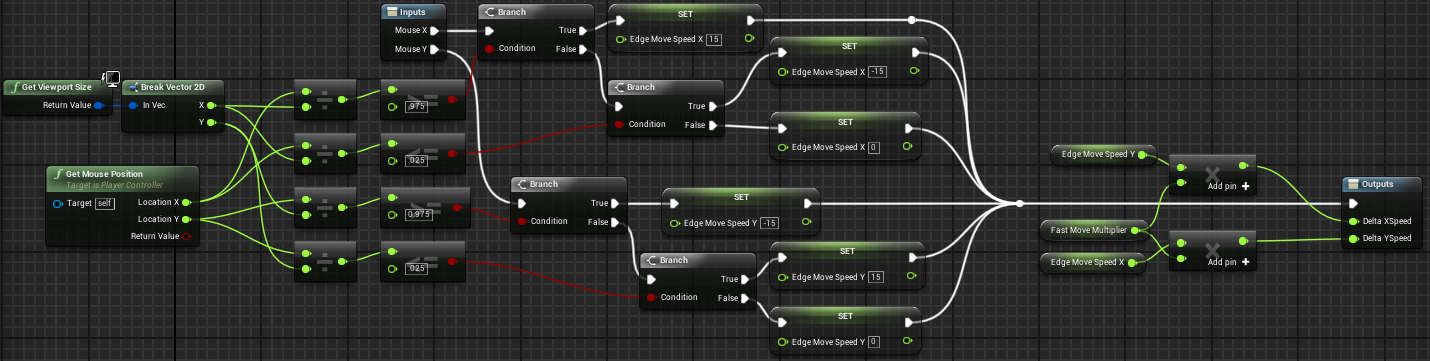
I eventet kastes player controlleren på index 0 til *CameraPawnController* hvorfra der laves et widget[[9]](#footnote-9). Den tager controlleren fra index 0, en reference fra *GhostBuilding* og en fra *CameraPawnController*, og widgeten der skal laves er *PlayerUI*. Til sidst sættes der en reference til UI’et og det tilføjes så til viewporten.

## Mouse edge movement



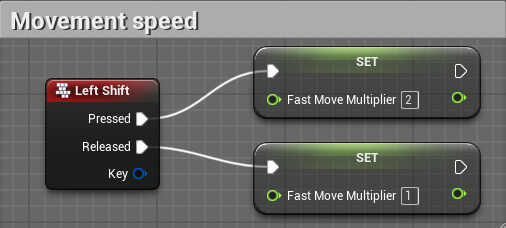
Her er funktionaliteten til at flytte kameraet når man bevæger musen ud til kanten af skærmen.

Til at starte med er der to events som sørger for at få fat i musens placering på X og Y aksen en gang hver frame. Derefter er der en branch, ækvivalensen til en *if* statement, hvor der tjekkes om bool’en *Disable Cam Nav* er *true* eller *false*. Er den *false*, kommer man til et makro[[10]](#footnote-10) som sørger for udregningerne.



I makroen hentes størrelsen på spillets vindue og destruerer dens vektor så man kan få fat i X og Y værdien. Derefter hentes musens X og Y position og så divideres musens X akse med viewportens X akse og ligeledes med Y værdien. Så tjekkes der så efterfølgende om den dividerede værdi er større end 0.975 eller mindre 0.025. Er den større end 0.975 sættes variablerne *EdgeMoveSpeed* til 15 som er farten som kameraet bevæger sig med og er den mindre end 0.025 vil variablerne blive sat til -15. Til sidst i makroen ganges der en multiplier, *Fast Move Multiplier*, til *EdgeMoveSpeed* variablerne så man kan lave funktionalitet til at sætte farten på kameraet op. Til sidst er der en funktion hvor der vælges hvilket objekt som skal gøre brug af overnævnte funktionalitet.

## Movement speed

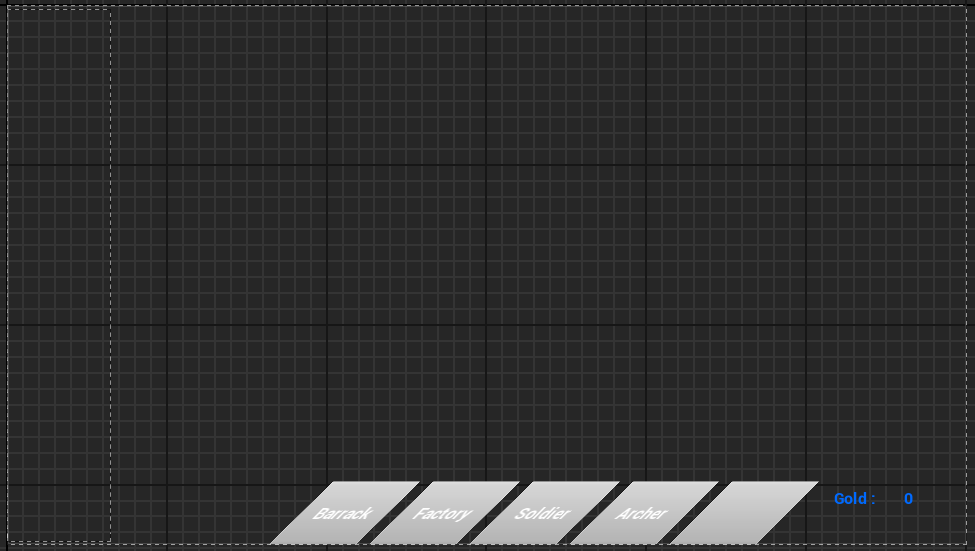


Dette er et simpelt event som ændre multiplieren så man har mulig for at bevæge kameraet hurtigere. Her tjekkes der for om den venstre Shift tast er trykket nede eller ej. Trykkes den ned vil *Fast Move Multiplier* variablen blive sat til 2, og er den ikke trykket ned vil den blive sat til 1.

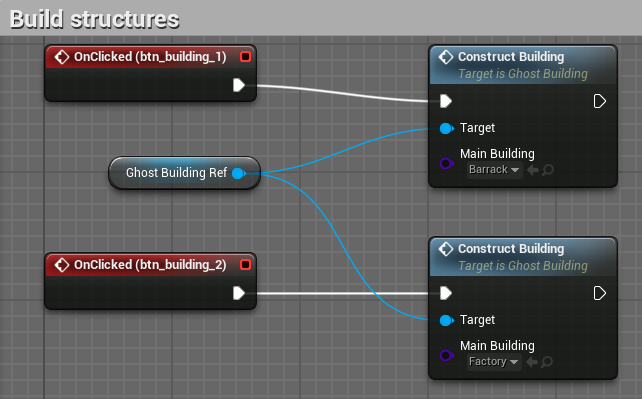
## Unit production

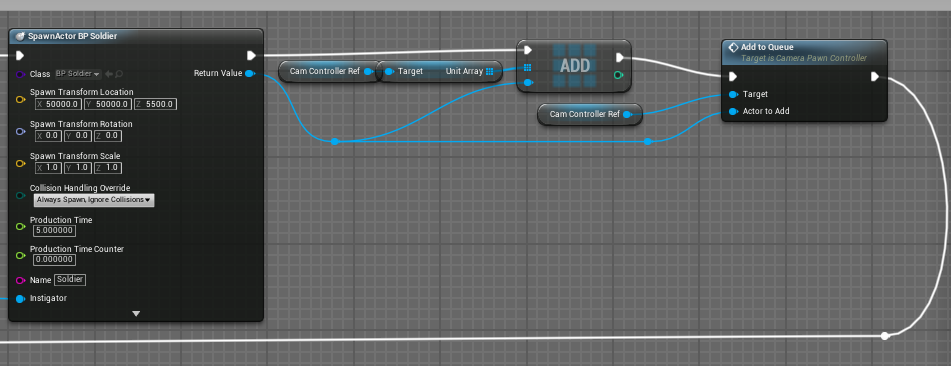
Unit produktionen fylder meget så læseren bedes referere til kommentar boksen *Build Units* i *CameraPawnController* blueprintet. Produktionen starter som den første sekvens i blueprintets *tick* event og går ind i en foreach lykke med et break[[11]](#footnote-11) som går igennem alle elementer i *UnitArray* som indeholder referencer til alle unit aktører som er i kø for at blive lavet. Lykken munder ud i en sekvens med to events. Første event munder ud i en branch hvor der tjekkes om objektet er nedarvet fra klassen *BPSoldier*, og er den det, køres der en ny foreach lykke igennem alle elementer i *MyBuildings* arrayet som indeholder referencer til alle bygningerne. Derefter køres der igen en branch hvor der tjekkes om elementet er nedarvet fra *Barrack* blueprintet, og skulle det ikke være tilfældet vil der ikke sket noget, men er det tilfældet kommer der endnu en branch som tjekker om barak referencen, som man har fat i, er igang med at blive bygget. Er den det sker der intet, men er den ikke det kommer der en branch der tjekker på, om barakken er igang med at bygge en unit. Er den det sker der intet, men er den ikke det sættes variablen *MyUnitToBuild*, fra *BuildingMasterScript* komponenten, og får *UnitMasterScript* komponenten med i dens unit input. Den munder så ud i *Server Change Unit Position* eventet som fortæller serveren hvor, at soldaten skal spawne. I dette tilfælde på den *billboard* komponent som barrack blueprintet har, og til sidst findes indekset i *UnitArray* og bliver fjernet derfra. Da en *Archer* skal igennem nødagtig den samme proces, burde det kun være nødvendigt at forklare den ene for at kunne forstå den anden.

## PlayerUI



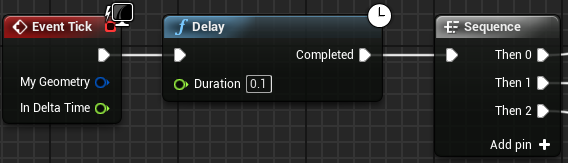
*PlayerUI* er et widget, og det indeholder knapper og andet brugbart information så som ressourcer, tid eller bygninger man kan bygge på skærmen.

Her køres der to events, et til hver af de to bygninger der er i spillet. Her gives der en *Ghost Building* reference og der vælges en bygning til den hvert sit *OnClick* event.

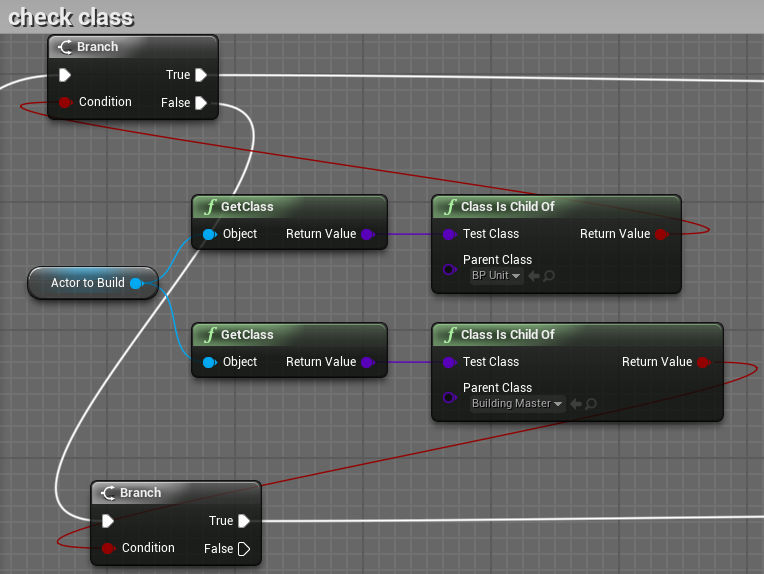
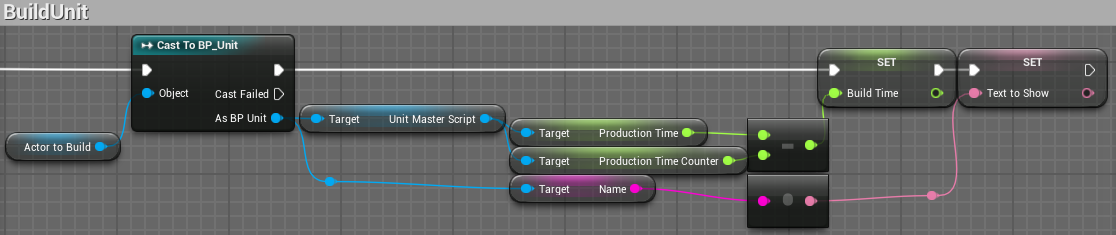


I dette *OnClick* event, produceres der en soldat. Processen for at producere en archer er nødagtig den samme. Den starter med at køre ud i en branch hvor der bliver kørt et tjek på størrelsen på *Unit Array* variablen. Er der mere end ti elementer i arrayet sker der intet, men er der mindre end ti elementer gås der ind i et *foreach* loop hvor *My Buildings* variablen gives med. Den indeholder alle bygninger og *castes* til en *barrack*, eller en *factory* hvis det er en *archer*. Der tjekkes efterfølgende om den er gyldig, og er den det, laves der en ny aktør af typen *Soldier*, der gives en position og en produktionstid med. Derefter tilføjes den nye *Soldier* til *Unit Array* variablen og sættes i kø via. *AddToQueue* funktionen fra *CameraPawnController* blueprinted.

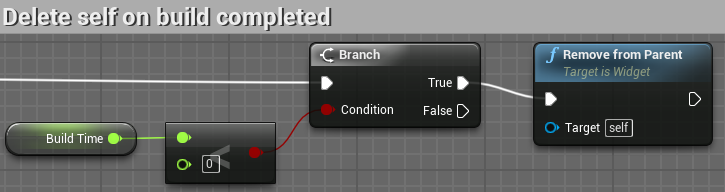
## PlayerUIButton

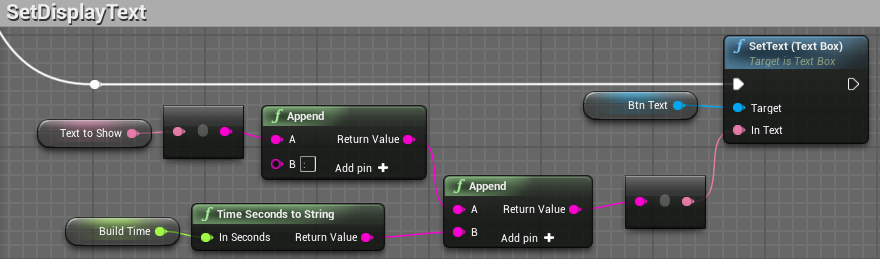
Dette widget blueprint sørger for, at der er et display når man bygger bygninger og soldater.

Der startes ud med at bruge en funktion til at køre en kort forsinkelse, da vi fandt ud af at det havde en stor påvirkning på antallet af fps’en. Derefter er der en række sekvenser.

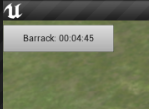
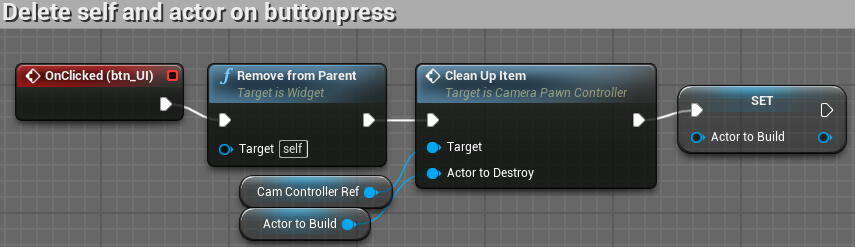
Den første sekvens køres ud i en branch hvor der tjekkes om der er nogle aktive aktører som har nedarvet fra klassen *BP\_Unit*. Er den det, vil den blive *casted* til en unit og dens produktions tid vil blive skrevet ud til skærmen, men har den ikke nedarvet, bliver der kørt en ny branch hvor der tjekkes om aktøren har nedarvet fra klassen *Building Master*. Er den det, vil den gå igennem samme proces som en *BP\_Unit* og blive skrevet ud til skærmen, men har den ikke nedarvet vil der ikke ske noget.

Skulle en aktør være en *BP\_Unit*, vil den blive kastet til en *BP\_Unit* som får en *Actor[[12]](#footnote-12)* reference. Fra dens *As BP\_Unit* output findes *Unit Master* komponenten som ligger på *BP\_Unit* blueprinted, og fra det komponent findes *Production Time* variablen som bliver fratrukket med *Production Time Counter* variablen som håndterer produktions tiden på en bygning eller soldat, og den bliver sat ind i en ny variabel som kaldes *BuildTime*. Derudover hentes *Name* variablen fra *BP\_Unit* som bliver konverteret til *TextToShow* variablen.

I denne sekvens tjekkes der om *BuildTime* er mindre end 0, og er den det, vil widgeten blive fjernet fra skærmen.



Denne sekvens sørger for at navn, konstruktions- og produktions tider bliver skrevet ud til skærmen i et forståeligt format.

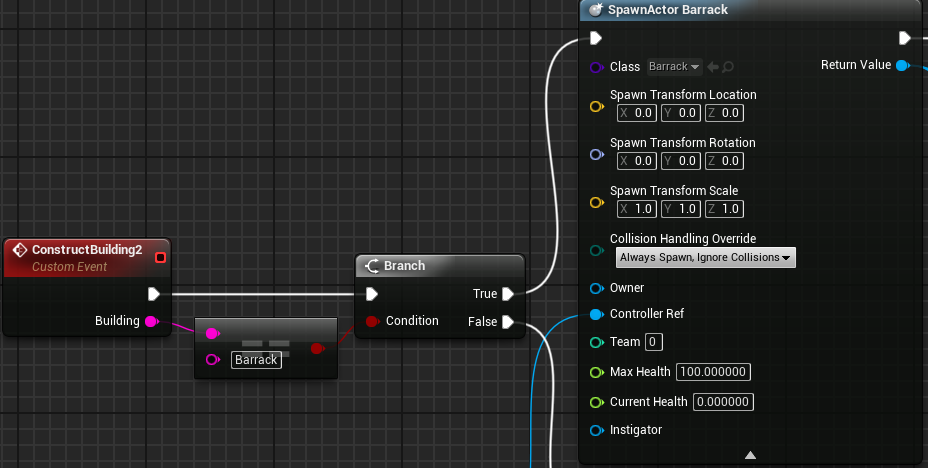
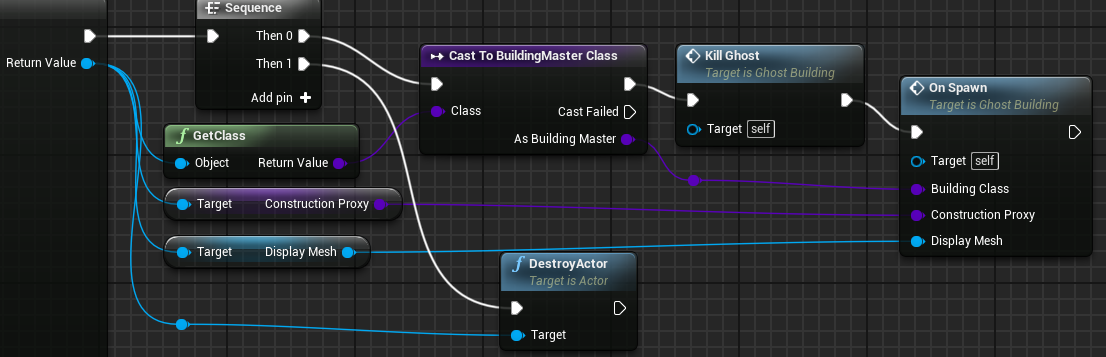
Først oprettes en *SetText* funktion og får en reference til tekst boksen som der skal gøres brug af. Derefter fås indholdet fra *TextToShow* og *BuildTime*. Først bruges en *Append* funktion som konverterer tekst variablen til en streng og indsættes i input *A*, og i *B* indsættes et semikolon for at adskille navnet på bygningen eller soldaten. Funktionen leder så videre til en ny *Append* funktion og bliver placeret i input *A* så det er det der forekommer først i tekst boksen. *BuildTime* variablen bliver sendt igennem en funktion der formaterer værdien til noget mere læsbart, og bliver efterfølgende indsat i input *B*. Funktionen konverteres så fra en streng til en tekst som så gives med i *In Text* inputten, og så har man en tekst der ligner eksemplet på billedet til højre.

Dette event sørger for, at man kan stoppe med at bygge hvis man nu skulle fortryde.

Ved at trykke på venstre musetast, imens musen holdes over en produktions bar, bliver widget komponenten fjernet via *Remove from Parent* funktionen og derefter kører *Clean Up Item* funktionen som tjekker om det er en bygning eller soldat der er i gang med at blive bygget, fjerner den fra dens retmæssige array og fjerner aktøren fra hierarkiet.

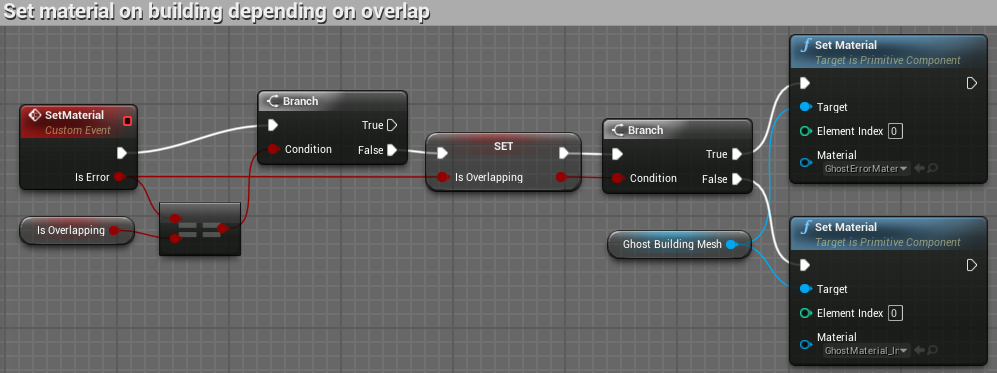
## GhostBuilding

*GhostBuilding* blueprinted er ment til at hjælpe spilleren med at finde ud af hvor man kan og ikke kan bygge sine bygninger. Dette indebærer bl.a. at tilføje en aktør med en transparens model af bygningen som man har tænkt sig at bygge, at tjekke om der er overlap med andre bygninger eller units, sætte et ny materiale hvis der er overlap eller ej og destruere aktøren når den ikke længere skal bruges.

Her er der et custom event som tager en streng som input. Så køres der en branch hvor der tjekkes om hvad der står i strengen, og det vil enten blive *Barrack* eller *Factory* afhængig af hvilken knap man trykker på. Er det en *Barrack* spawnes der sådan en.

Der køres efterfølgende en sekvens hvor *Barrack* kastes til *BuildingMaster* via *GetClass*. Efterfølgende køres *KillGhost* eventet som sørger for at lukke porten nulstille ghostets referencer, mesh og variablerne der hjælper med at finde ud af om der bl.a. allerede er, eller ikke er, et aktivt ghost og om man kan markere en unit. Til sidst kører *OnSpawn* eventet som sætter referencer, mesh og variabler der fortæller, at der er et aktivt ghost. Skulle strengen returnere noget andet end *Barrack*, vil der spawne en *Factory* som der så vil gennemgå den samme proces som barakken skulle igennem.

Dette er *GhostBuilding*’s tick event som starter med, at køre ind i en *Gate[[13]](#footnote-13)* som der er lukket fra starten. For at kunne komme videre er man nød til at åbne den, og det gøres via. custom eventet *OpenTickGate*, og den lukkes igen med *CloseTickGate* eventet. Begge events vil blive forklaret efterfølgende. Når denne *gate* er åben bruges der en funktion til at sætte lokationen på en aktør. Her gives der en reference til *GhostBuilding* med, og da man allerede befinder sig i *GhostBuilding* blueprintet kan man bruge *self* referencen i stedet for. I *New Location* bruges en custom funktion som sørger for, at den definerede aktør følger musen og, at aktørens mesh bevæger sig som var den på et grid. Efterfølgende kommer der en branch hvor der køres et tjek på om der er overlap mellem to aktører via. en custom funktion. Der køres så et custom event, både for sandt eller falsk, som bruges til at sætte et materiale til at visualisere hvor der er og hvor der ikke er overlap. Her er der også en IsError variabel som bruges til at afgøre hvilket materiale der skal på meshen.

Her er custom eventet som tager sig af hvilket materiale der skal på meshen hvis der er, eller ikke er, overlap.

Her køres der en branch på IsError variablen for at teste, om der er overlap mellem ghost aktøren og andre aktører i hierarkiet. Der bliver sat en reference til *GhostBuilding* meshen, og hvis der er overlap, vil meshen få et rødt, transparens materiale til at illustrere, at der er overlap, men er der ikke overlap vil meshen få et gråt, transparens materiale.

1. <https://docs.unrealengine.com/latest/INT/Engine/Blueprints/GettingStarted/index.html> [↑](#footnote-ref-1)
2. <https://docs.unrealengine.com/latest/INT/Engine/Blueprints/UserGuide/Events/#eventtick> [↑](#footnote-ref-2)
3. <https://docs.unrealengine.com/latest/INT/Engine/Blueprints/UserGuide/FlowControl/#branch> [↑](#footnote-ref-3)
4. <https://docs.unrealengine.com/latest/INT/Gameplay/Framework/Controller/PlayerController/> [↑](#footnote-ref-4)
5. <https://docs.unrealengine.com/latest/INT/Engine/Blueprints/UserGuide/Events/Custom/> [↑](#footnote-ref-5)
6. <https://docs.unrealengine.com/latest/INT/Gameplay/Framework/Pawn/> [↑](#footnote-ref-6)
7. <https://docs.unrealengine.com/latest/INT/Gameplay/Framework/UIAndHUD/index.html> [↑](#footnote-ref-7)
8. <https://docs.unrealengine.com/latest/INT/Engine/UI/LevelEditor/Viewports/Basics/> [↑](#footnote-ref-8)
9. <https://docs.unrealengine.com/latest/INT/Engine/UMG/UserGuide/WidgetBlueprints/index.html> [↑](#footnote-ref-9)
10. <https://docs.unrealengine.com/latest/INT/Engine/Blueprints/UserGuide/Macros/> [↑](#footnote-ref-10)
11. <https://docs.unrealengine.com/latest/INT/Engine/Blueprints/UserGuide/FlowControl/#forloopwithbreak> [↑](#footnote-ref-11)
12. <https://docs.unrealengine.com/latest/INT/Programming/UnrealArchitecture/Actors/> [↑](#footnote-ref-12)
13. <https://docs.unrealengine.com/latest/INT/Engine/Blueprints/UserGuide/FlowControl/#gate> [↑](#footnote-ref-13)