HOGESCHOOL ROTTERDAM

Hoe realiseren we een ontwikkel omgeving waarin niet-programmeurs, zoals 3D artiesten en level designers, efficint unieke virtual reality ervaring kunnen creren in Unreal Engine 4

door

Mark Arts

Een scriptie geschreven voor het halen van de
bachelor of science Mediatechnologie

voor

Mediatechnologie
Instituut voor Communicatie Media en Informatietechnologie

April 2016

“I choose a lazy person to do a hard job. Because a lazy person will find an easy way to do it.”

li Bill Gates

HOGESCHOOL ROTTERDAM

Abstract

Mediatechnologie
Instituut voor Communicatie Media en Informatietechnologie

Bachelor of Science

door Mark Arts

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too...

Acknowledgements

Als eerste wil ik DPI bedanken voor het mogelijk maken en het faciliteren van deze scriptie. Extra dank gaat naar Boy .... die ons met enthousiasme en vertrouwen begeleid heeft.

Naast DPI ben ik dankbaar voor het zelfde enthousiasme en vertrouwen van Emiel Bakker onze afstudeer begeleider vanuit het HRO.

Randy Dijkstra, die zijn scriptie aan de andere kant van mijn bureau schreef, wil ik bedanken voor zijn vriendschap. Zonder jou had ik aanzienlijk minder motivatie gehad om een uitdagend onderwerp te kiezen, laat staan het succesvol afronden ervan.

Als laatste bedank ik mijn ouders voor het dak boven mijn hoofd, mijn avondeten, het wassen, het opruimen en natuurlijk voor alle mogelijkheden die ze mij gegeven hebben om mijzelf te ontwikkelen.

Contents

Abbreviations

HRO	**H**ogeschool **RO**tterdam
VS	**V**isual **S**cripting

Voor all mijn vrienden die om onverklaarbare redenen graag met mij omgaan

Chapter 1

Inleiding

In een periode van 18 weken zal ik proberen een reeks van basis componenten voor de Unreal Engine 4 te ontwikkelen waarmee niet-programmeurs efficint interactie aan virtual reality demos kunnen toevoegen.

De Unreal Engine zal gebruikt worden omdat deze een uitgebreid VS systeem heeft die het makkelijk maakt om simpele logica, zoals wanneer en waar iets moet plaats vinden, makkelijk uit te drukke. Het VS systeem maakt het mogelijk voor niet-programmeurs om simpele logica zelf toe te voegen zonder dat hier een programmeur voor nodig is. Naast een efficintere workflow maakt dit het ook makkelijk om als designer, 3D artist te experimenteren.

1.1 Probleemstelling

Momenteel is het toevoegen van interactieve elementen in virtual reality een intensieve programmeer taak omdat het vaak ontwikkelt wordt voor een specifieke use case. Het is daardoor vaak lastig om de geprogrammeerde logica te hergebruiken in een ander project.

Hierdoor is er altijd een programmeur nodig om de gewenste functionaliteit toe te voegen of aan te passen. Vooral in gameontwikkeling zijn er veel sprongen gemaakt in het oplossen van dit probleem. Een van deze oplossingen is het gebruikt van VS om interactie in een spel te programmeren. Er zijn hier al veel sprongen in gemaakt maar het implementeren van VR gerelateerde componenten is nog niet standaard aanwezig.

1.2 Doelstelling

Het creren van een reeks basis componenten voor Unreal Engine 4 waarmee niet programmeurs interactieve virtual reality demos kunnen maken die toegevoegde waarde hebben aan de workflow van DPI.

Aan het eind van dit traject zal het mogelijk zijn om met de gemaakte componenten een bestaande omgeving zonder programmeren geschikt te maken voor VR en om een nieuwe omgeving op te zetten zonder behulp van een programmeur.

1.3 Onderzoeksvraag

1.3.1 Hoofdvraag

Hoe realiseren we een ontwikkel omgeving waarin niet-programmeurs, zoals 3D artists en level designers, efficint unieke virtual reality ervaring kunnen creren in Unreal Engine 4.

1.3.2 Deelvragen

- Hoe kan er een koppeling met VS gemaakt worden die intutief is voor niet programmeurs.
  - Wat is de huidige kennis van computer logica onder de werknemers van DPI
  - Is er extra training nodig om met de ontwikkel omgeving aan de slag te gaan
- Welke componenten zullen gemaakt moeten worden?
  - Wat zijn de huidige taken van een programmeer in het maken van een Virtuele omgeving
  - Welke tools bestaan er all
  - Wat zijn de juiste abstracties van de componenten
- Is het mogelijk de componenten cross-platform te maken
  - Welke platformen zijn relevant?
  - Wat zijn de verschillende mogelijkheden van deze platformen

1.3.3 Onderzoeksmethoden

1.3.3.1 Hoe kan er een koppeling met VS gemaakt worden die intutief is voor niet programmeurs?

De hoeveelheid abstractie van de componenten zijn afhankelijk van de bestaande intutie voor programmeren. Als een gebruiker namelijk snapt hoe de als dit dan dat constructie werkt kan er meer vrijheid in de componenten gecreerd worden.

Als eerst zal er onderzocht moeten worden wat de huidige kennis is van de niet-programmeurs. Dit kan door middel van interviews en een aantal vragen.

Vervolgens zal er onderzocht moeten worden wat een haalbaar moeilijkheidsgraad is en of er baat is van een extra training voor de niet-programmeurs.

Als er sprake is van toegevoegde waarde van een extra training dan zal deze parallel aan het project gegeven worden en iteratief ontwikkeld.

1.3.3.2 Welke componenten zullen gemaakt moeten worden?

Dit zal beginnen met een onderzoek naar de huidige taken vaan programmeur tijdens het maken van een virtuele reality omgeving. Aan de hand hiervan zal er gekeken worden welke taken geautomatiseerd kunnen worden of versimpeld naar virtual scriptring.

Als er een duidelijk beeld is van de benodigde componenten zal er een onderzoek gedaan worden naar bestaande tools die ingezet kunnen worden om de workflow te verbeteren.

Als het duidelijk is welke componenten zelf geschreven moeten worden en hoeverre de niet-programmeurs met VS om kunnen gaan kan er bepaald worden welke abstractie de componenten zo breed inzet baar mogelijk houd terwijl het intutief blijft voor de niet-programmeurs

1.3.3.3 Is het mogelijk de componenten cross-platform te maken?

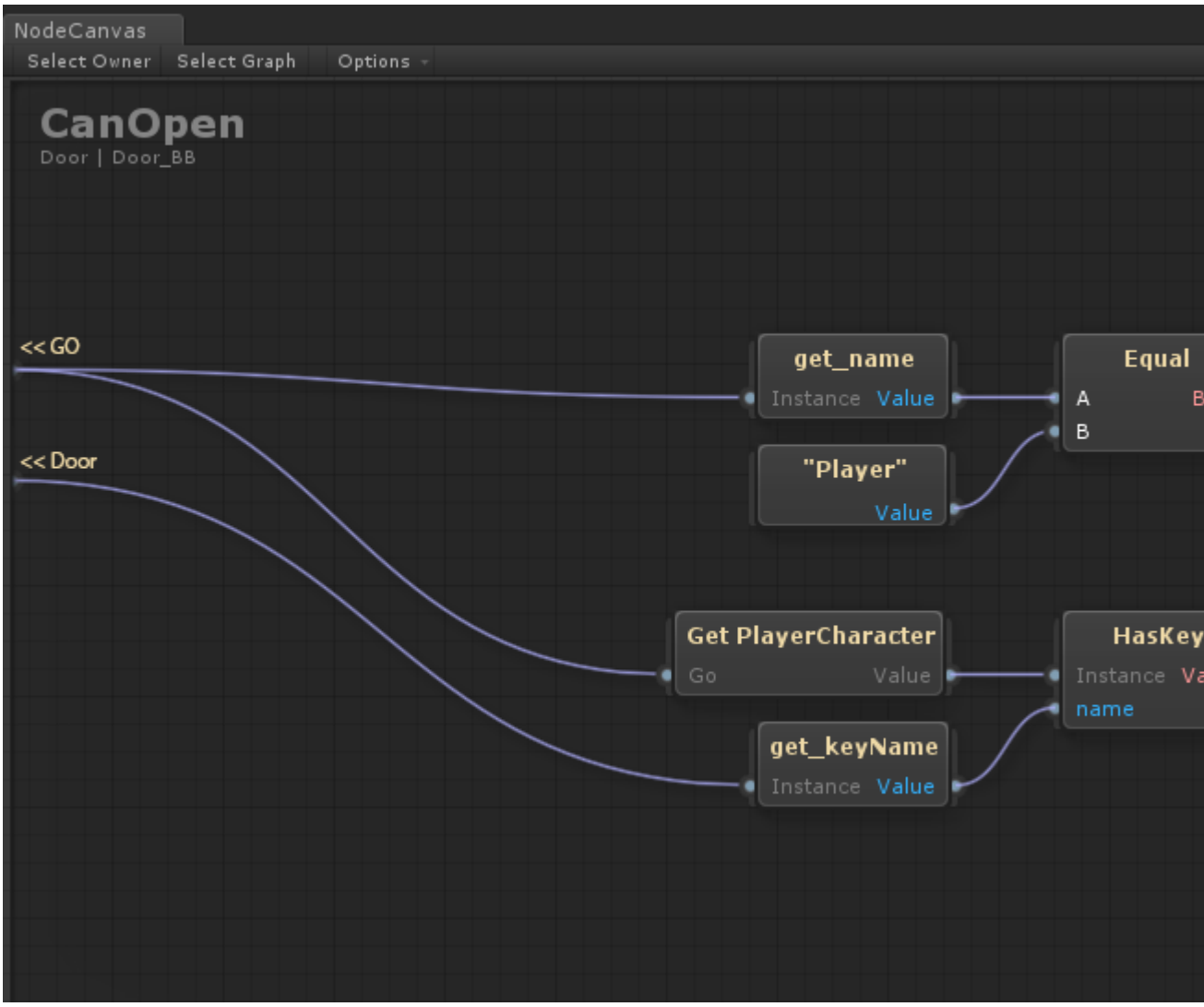
Er zal hier onderzocht moeten worden welke platformen relevant zijn en wat de mogelijkheden voor elk platform is.

Er zal daarna een beslissing gemaakt moeten worden voor elk component of het een alternatieve versie moet hebben of dat cross-platform in het component zelf meegenomen kan worden.

Chapter 2

VS

VS maakt het mogelijk om logica op een visuele manier inplaats van textuel te schrijven. Er is minder kennis van de onderliggende werking van computer systemen nodig dan voor tekstueel programmeren [?]. Hierdoor kan er zonder kennis van computers en hun werking al snel mee gewerkt worden door niet programmeurs.



Daarnaast word de mogelijkheden van de visuele programmeer taal ook vaak gelimiteerd om het de gebruiker makkelijker te maken en het programma te beschermen van de gebruiker. Meestal als de limitaties van de visuele scripting taal een probleem worden had de logica beter in een tekstuele taal geschreven kunnen worden.

2.1 Use Case's

Door de jaren heen zijn er voor verschillende doeleinden visuele programmeer talen gemaakt. Een aantal voorbeelden zijn

- Data flow in applicaties
- State flow voor onder andere animaties
- Geluids effecten programmeren

- Gameplay Logica
- Programeren leren aan beginners

In deze scriptie word er gefocust op het gebruik van VS voor gameplay logica.

2.2 Gameplay Logica

Het programmeren van gameplay in een spel bestaat voor een groot gedeelte uit de vragen Wanneer moet iets gebeuren, Wat moet er gebeuren en Hoe moet dit gebeuren. De vragen Waarom moet iets gebeuren en Waar moet dit gebeuren komen het meeste voor in de gameplay logica van een spel, vaak hebben deze vragen ook een makkelijk antwoord. Helaas zijn deze vragen lastig om te beantwoorden in textuele code.

2.2.1 Het afvuren van een projectiel

Een voorbeeld van deze drie vragen in code is als volgt, voor het voorbeeld laten wij de logica zien van het schieten van een projectiel in c++.

2.2.1.1 Tekstuele Implementatie

In een c++ header bestand registreren wij de volgende functie in onze speler classe.

```
void OnFire();
```

Dan tijdens het opzetten van de input voor ons karakter laten we weten wanneer we een projectiel willen schieten.

```
if( EnableTouchscreenMovement(InputComponent) == false )
{
  InputComponent->BindAction(
    "Fire",
    IE_Pressed,
    this,
    &Adpi_unreal_colosseumCharacter::OnFire
  );
}
```

Maar omdat de fire actie niet werkt op een touch interface moeten we de onfire zelf afvuren als iemand het scherm aanraakt (De logica achter het registeren van touch events word niet getoond maar is wel aanwezig in de speler klasse).

```
void Adpi_unreal_colosseumCharacter::EndTouch(const ETouchIndex::Type
    FingerIndex, const FVector Location)
{
  if (TouchItem.bIsPressed == false)
  {
    return;
  }
  if( ( FingerIndex == TouchItem.FingerIndex ) && (TouchItem.bMoved ==
    false) )
  {
    OnFire();
  }
  TouchItem.bIsPressed = false;
}
```

Vervolgens definiren we OnFire als volgt

```
void Adpi_unreal_colosseumCharacter::OnFire()
{
  // try and fire a projectile
  if (ProjectileClass != NULL)
  {
    const FRotator SpawnRotation = GetControlRotation();
    // MuzzleOffset is in camera space, so transform it to world space
    before offsetting from the character location to find the final muzzle
    position
    const FVector SpawnLocation = GetActorLocation() + SpawnRotation.
    RotateVector(GunOffset);

    UWorld* const World = GetWorld();
    if (World != NULL)
    {
      // spawn the projectile at the muzzle
      World->SpawnActor<Adpi_unreal_colosseumProjectile>(ProjectileClass,
    SpawnLocation, SpawnRotation);
    }
  }

  // try and play the sound if specified
  ...

  // try and play a firing animation if specified
```

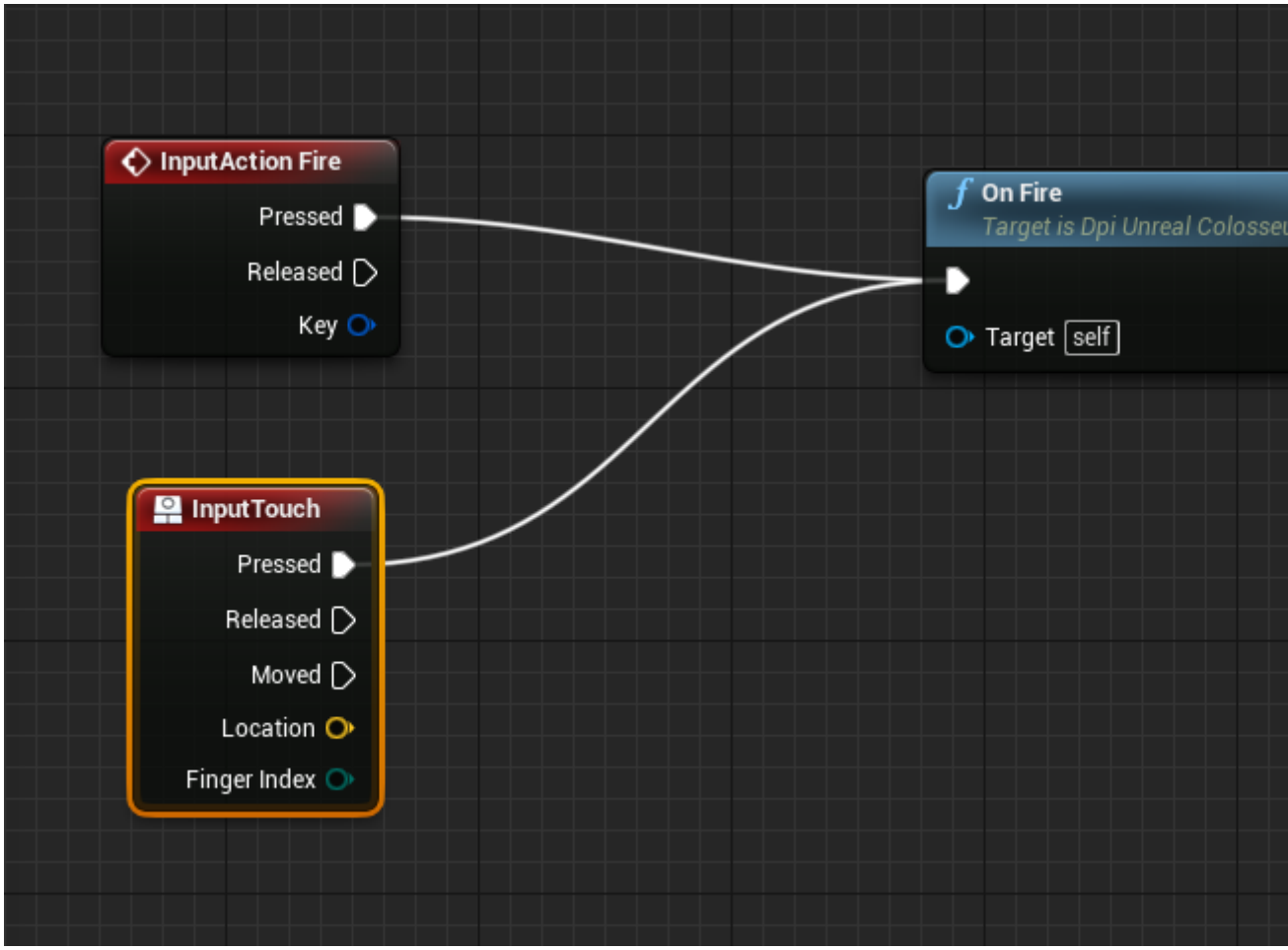
```
22    . .
}
```

Om de logica te implementeren voor het vuren vuren van de kogel hebben wij nu op vier verschillende plekken de logica moeten verspreiden, hiervan zit de implementatie verspreid in een bestand van 218 regels (zie bijlage standaard playercharcter van Epic).

2.2.1.2 Tekstuele Implementatie

De Hoe moet dit gebeuren is een complexe vraag die prima beantwoord word in code. Vooral omdat er verschillende logica achter elkaar plaatsvind, geluid afspelen / animatie starten, en een aantal wiskundige berekeningen.

Maar de Wanneer vraag is makkelijk te beantwoorden. Namelijk als de fire actie uitgevoerd word of als er op het scherm gedrukt word. De logica in blueprints ziet er als volgt uit (in de event graph van een playerCharacter).



In tegenstelling tot de c++ code is de logica van de Wanneer vraag dit maal niet verspreid en is het in een oog opslag duidelijk wat er voor zorgt dat er een projectiel geschoten word.

Chapter 3

Unreal Engine 4

3.1 Waarom de Unreal Engine

Tijdens het kiezen van een Engine waren er drie rand voorwaarden namelijk

1. gratis voor educatie, of goedkoop genoeg voor DPI om aan te schaffen
2. Oculus en Gear VR ondersteuning
3. Een VS systeem

De volgende Engines zijn naar gekeken

- Unity
- CRYENGINE
- Unreal Engine 4

3.1.1 Unity

3.1.1.1 Licentie

Unity heeft een educatie licentie waaronder het grootste gedeelte van de engine gratis gebruikt kan worden maar rekening houdend met de interesses van DPI, en eventuele vervolg projecten die zei willen ondernemen, zal de licentie minimaal 75 per maand worden met daar de koste van Android builds bij.

3.1.1.2 Oculus en Gear VR ondersteuning

Van de drie engines heeft unity de meest uitgebreide platform support. Daarnaast is Unity vaak een van de eerste keuzes om nieuwe technieken mee te implementeren. De reden achter deze keuze is het gemak waarop plugins gemaakt kunnen worden en de voordelige / gratis pricing van Unity wat het een populaire keuze maakt voor experimenteren.

3.1.1.3 VS

Unity heeft zelf niet een ingebouwd VS systeem maar via een aantal plugins kan dit wel worden toegevoegd. Deze plugins zitten wel vast aan de limitaties van een Unity Plugin. Daarnaast word dit niet officieel gesupporterd door Untiy zelf.

3.1.1.4 Gebruiks gemak

Op gebruiksgemak, in serieuze projecten, fidelity en performance loopt Unity ver achter op de Unreal Engine 4 en CryEngine. Onderhoudbaar en uitbreidbaarheid van Unity is meestal rampzalig door de manier waarop code voor Unity geschreven word. Vaak is Unity daarom ook niet de keuze voor grote games van hoge kwaliteit.

Het programmeren word namelijk in een component style in c-sharp gedaan. Het programmeren in c-sharp is voor veel mensen fijn omdat dit vaak een bekende taal is. Maar het component systeem en de manier waarop Unity relaties afhandelt zorgt dat er extreem snel spaghetti code ontstaat.

3.1.2 CRYENGINE

3.1.2.1 Licentie

De CRYENGINE licentie bestaat uit een Pay what you want model. Daarnaast beid Crytek een CRYENGINE Insider Membership aan voor 50 of 150.

3.1.2.2 Oculus en Gear VR ondersteuning

Op het moment van schrijven ondersteund de CRYENGINE wel de Oculus maar niet de GearVR. Android ondersteuning is mogelijk maar niet officieel ondersteund door Crytek. Dit zal het lastig maken om Gear VR te ondersteunen.

3.1.2.3 VS

De CRYENGINE heeft een ingebouwd VS systeem genaamd Flow. Dit systeem heeft helaas geen ondersteuning voor overerving en de koppeling met c++ bestaat alleen uit het maken van nieuwe nodes en het aanspreken van graphs in c++.

3.1.2.4 Gebruiks gemak

De CryEngine staat bekend als een moeilijke Engine om in te beginnen maar heeft ondertussen bewezen een goede keuze te zijn voor game ontwikkelaars en grotere teams.

Door het gebrek aan persoonlijke ervaring kan ik hier minder over zeggen. Maar door het gebrek aan Gear VR ondersteuning is er hier niet meer in verdiept.

3.1.3 Unreal Engine 4

3.1.3.1 Licentie

Het gebruik van de Unreal Engine 4 is compleet gratis voor educatie en niet-game applicaties zoals architectuur, visualisatie, films etc. Dit betekend dat zowel tijdens het afstuderen als mogelijke vervolg projecten van DPI er geen licentie kosten betaald hoeven te worden.

3.1.3.2 Oculus en Gear VR ondersteuning

Unreal Engine 4 ondersteund zowel de Oculus als Gear VR. Daarnaast ondersteund het ook de meeste randapparatuur voor VR zoals Leap Motion en Kinect. Updates hiervoor komen vaak wel later dan voor Unity. VS Unreal heeft een uitgebreid VS systeem wat nauw samenwerkt met c++. Alle functies die in c++ beschikbaar zijn zijn beschikbaar via Blueprints. Van elke Blueprint kan ook direct de source code ingesprongen worden, zelfs voor de functies uit de Unreal Engine 4 libraries.

Het koppelen van c++ aan blueprints is vrij simpel en ondersteund ook het creen van events in c++ en functionaliteit hieraan via blueprints te kopellen.

3.1.3.3 Gebruiks gemak

De leer curve van Unreal Engine is redelijk stijl maar binnen een paar weken is het mogelijk de belangrijkste te leren. Een punt om rekening mee te houden is dat het zomaar

iets maken in de Unreal Engine 4 vaak verkeerd uitpakt. Daarom is het belangrijk de documentatie, van het onderdeel waar je mee bezig bent, goed te lezen.

Als de basis principes eenmaal onder de knie zijn kan er extreem snel ontwikkeld, en geprototyped, worden met de unreal engine door de implementatie van VS en een strakke en goed uitgedachte UI.

3.1.3.4 Conclusie

De CRYENGINE valt af vanwege zijn gebrek aan Gear VR support. Dan blijft de keuze over tussen Unity en Unreal Engine 4. Untiy heeft een minder style learning curve maar geavanceerde 3D technieken zullen uiteindelijk toch geleerd moeten worden om een hoge kwaliteit te behalen.

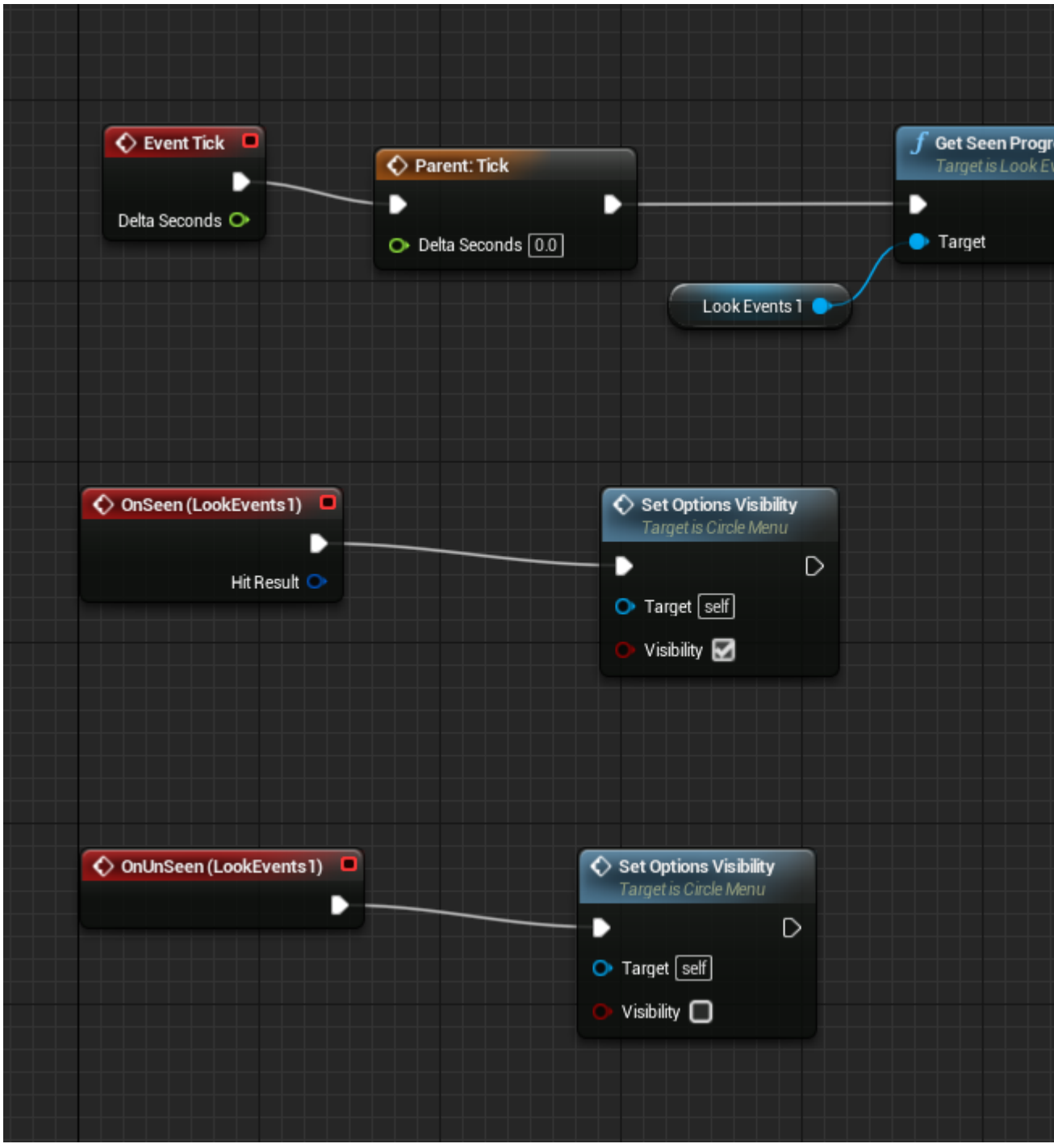
Het visuele scripting systeem van Unity is beperkt en mist de kracht en flexibiliteit die nodig is om deze scriptie succesvol af te ronden. Daar in tegen beid het visuele scripting systeem van Unreal Engine 4 wel deze mogelijkheden.

3.2 Blueprints

Blueprints is het visuele scripting systeem van Unreal Engine 4 en wat deze scriptie mogelijk maakt. De beschrijving van Unreal zelf is als volgt:

Blueprints are special assets that provide an intuitive, node-based interface that can be used to create new types of Actors and script level events; giving designers and gameplay programmers the tools to quickly create and iterate gameplay from within Unreal Editor without ever needing to write a line of code.

Blueprints ziet er als volgt uit:



Een hele kort samenvatting zou zijn dat de rode nodes Events zijn en dus aangeven wanneer iets gebeurd, de witte lijnen bepalen de volgorde waarin de nodes worden uitgevoerd, de blauwe nodes zijn functies en de rest zijn variabelen of pure transformatie van data.

Chapter 4

Blueprints en C++

In dit hoofdstuk word gameplay logica in drie aspecten verdeeld en word voor elk aspect naar de voor en nadelen van C++ en Blueprints gekeken.

Aan de hand van deze vergelijkingen word een workflow gekozen voor het programmeren van deze logica waarin wij het beste uit C++ en Blueprints proberen te combineren.

4.1 Gameplay

Om de scheiding tussen c++ en Blueprints concreet te maken verdelen wij gameplay logica in de volgende vragen:

- Wanneer moet iets gebeuren
- Wat moet er gebeuren
- Hoe moet dit gebeuren

Door deze scheiding wordt het makkelijker om de keuze tussen een C++ en een Blueprint implementatie te maken en kunnen er een aantal richtlijnen opgezet worden.

4.1.1 Wanneer

De wanneer vragen zijn vaak makkelijk te beantwoorden, bijvoorbeeld als de speler geraakt word door een projectiel wil ik dat geluid x afgespeeld word, maar moeilijk te coderen door hun asynchrone natuur. Een van de krachtigste voordelen van een Visuele

programmeer taal is dat de flow van een programma uitgedrukt kan worden door middel van de lijnen tussen nodes.

Om het verschil tussen asynchrone logica in C++ en blueprints duidelijk te maken implementeren wij hieronder het afspelen van een geluid nadat een speler dood gaat.

4.1.1.1 C++

We registeren eerst een functie die aangesproken kan worden door de timeout en het geluid wat afgespeeld word in de header van de character.

```
/** Plays a sound x seconds after the death of the player*/
void AfterDeathSoundTimeOut();

/** Sound to play each time we fire */
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category=Gameplay)
class USoundBase* DeathSound;
```

Vervolgens word deze functie gemplementeerd in de character.cpp

```
void Adpi_unreal_colosseumCharacter::AfterDeathSoundTimeOut()
{
  UGameplayStatics::PlaySoundAtLocation(this, FireSound, GetActorLocation()
    );
}
```

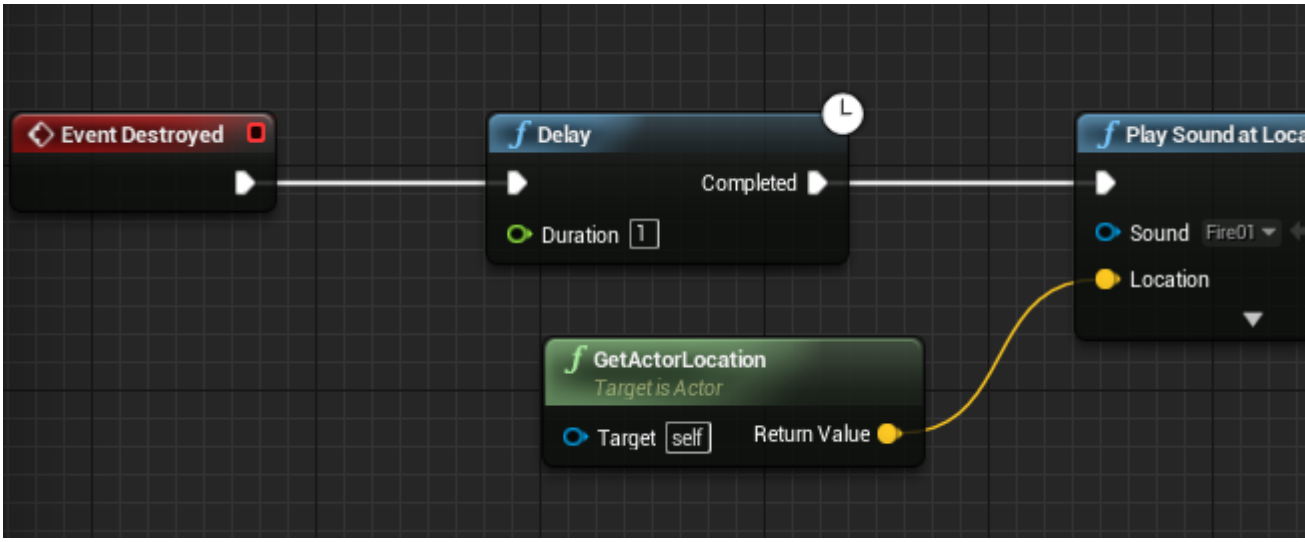
En word de timeout voor het geluid gezet tijdens het doodgaan van de speler.

```
void Adpi_unreal_colosseumCharacter::OnDeath(const FDeathReason Reason)
{
  // death logic
  ...

  FTimerHandle UnusedHandler= FTimerHandle();
  GetWorld()->GetTimerManager().SetTimer(
    UnusedHandler,
    this,
    &Adpi_unreal_colosseumCharacter::AfterDeathSoundTimeOut,
    1.0f
  );
}
```

We zien hier dat de gerelateerde code op drie verschillende plekken komt te staan, tussen niet relevante code in. Pas na het lezen van de code op deze drie plekken word het duidelijk wat de complete functionaliteit is. Er zijn hier natuurlijk hulpmiddel voor zoals opmerkingen boven code te plaatsen maar bij elke extra taak die, asynchroon, uitgevoerd moet worden word de code complexer en moeilijker te begrijpen. De lezer moet namelijk alle gerelateerde functionaliteit in zijn geheugen hebben.

4.1.1.2 Blueprints

In Blueprints zou deze logica er als volgt uit zien:



In de blueprint implementatie is het in een oog opslag duidelijk dat er een geluid afgespeeld word op de locatie van de speler een seconde nadat deze deze dood gaat. De logica bevind zich op de dezelfde plek en de witte lijnen geven de flow van de logica aan.

Een ander groot verschil dat we hier zien is dat er voor iets simpels als een vertraging in tekstuele code naast de standaard kennis van de c++ syntax ook kennis nodig is van de volgende concepten:

- Pointers
- Pass by reference
- Function references
- Namespaces
- Out parameters (de UnusedHandler)
- Floats

- Types
- Macros.

Terwijl in Blueprints all deze concepten verborgen zijn in de nodes. Het verbergen van de onderliggende werking van functionaliteit is een thema wat vaker terugkomt in programmeren en wordt aangemoedigd. Het verbergen van deze logica heeft als gevolg dat er zonder programmeer kennis de wanneer logica gemplementeerd kan worden.

4.1.2 Wat

Het plaatsen van de wat logica in Blueprints of c++ is lastiger om te bepalen. Voor logica die de niet-programmeurs schrijven is c++ geen optie en moet dit wel in Blueprints maar voor programmeurs moet er een afweging gemaakt worden.

Het probleem ontstaat voornamelijk bij complexe conditionele logica.

4.1.2.1 Conditionele Logica

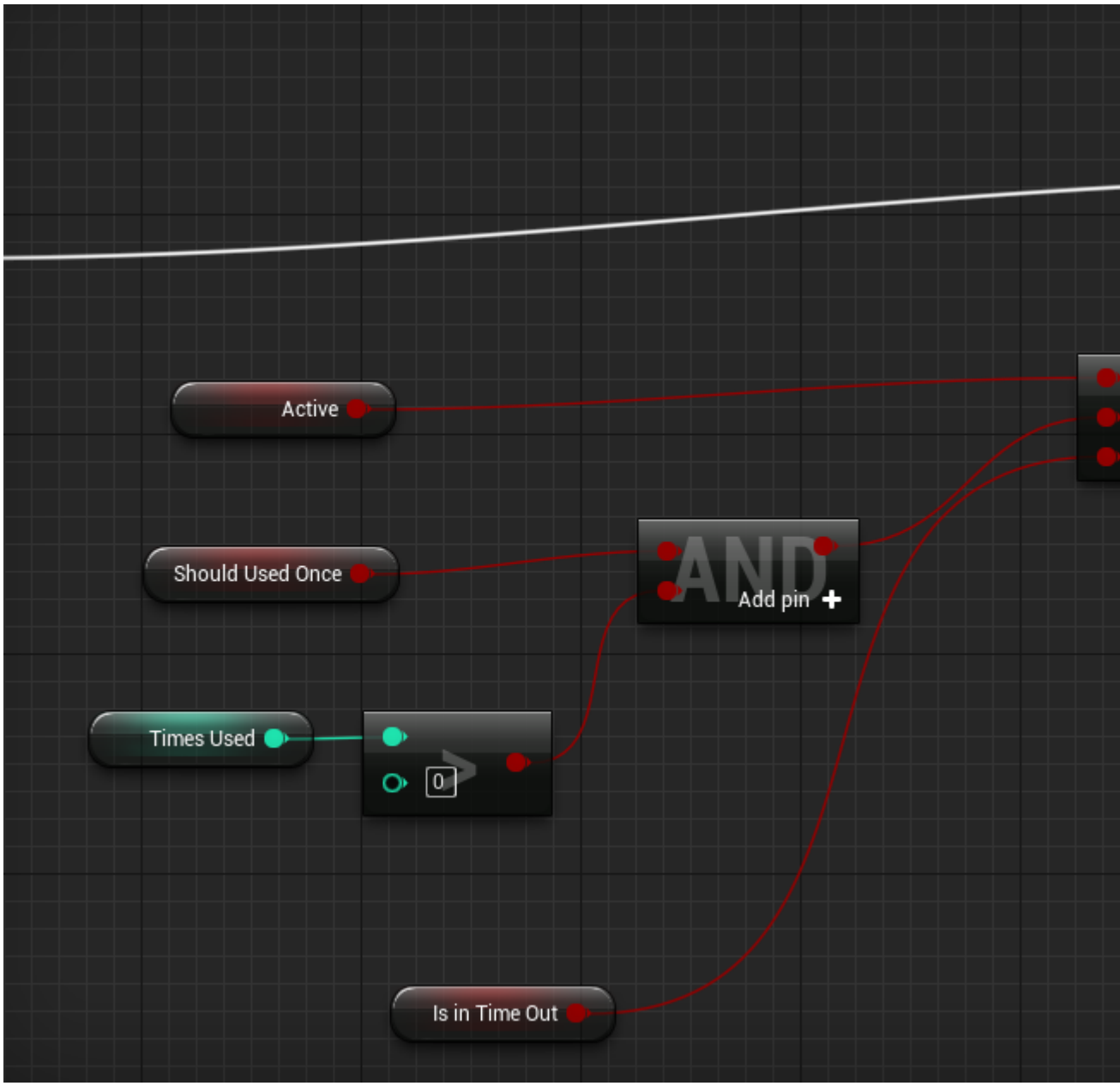
Als we de conditionele logica van het afvuren van de events van een. LookEventsComponent[zie hoofstuk ?] in c++ en blueprints met elkaar vergelijken.

- Bijlage 1: Tick functie van de LookEventsComponent A
- Bijlage 2: Conditional logic van Tick functie van LookEvents in Blueprints B

Zijn beide varianten moeilijk te lezen. Voor iemand die niet codeert ziet de Blueprints variant er waarschijnlijk begrijpgaarder uit maar de complexiteit komt voornamelijk door de logica zelf en in tekstuele code zijn er een aantal manieren om dit soort constructies kleiner te maken zoals:

```
1 if (bActive != true || (bShouldUsedOnce && TimesUsed > 0) || bIsInTimeOut
    == true)
{
3   return;
}
```

In vergelijking met

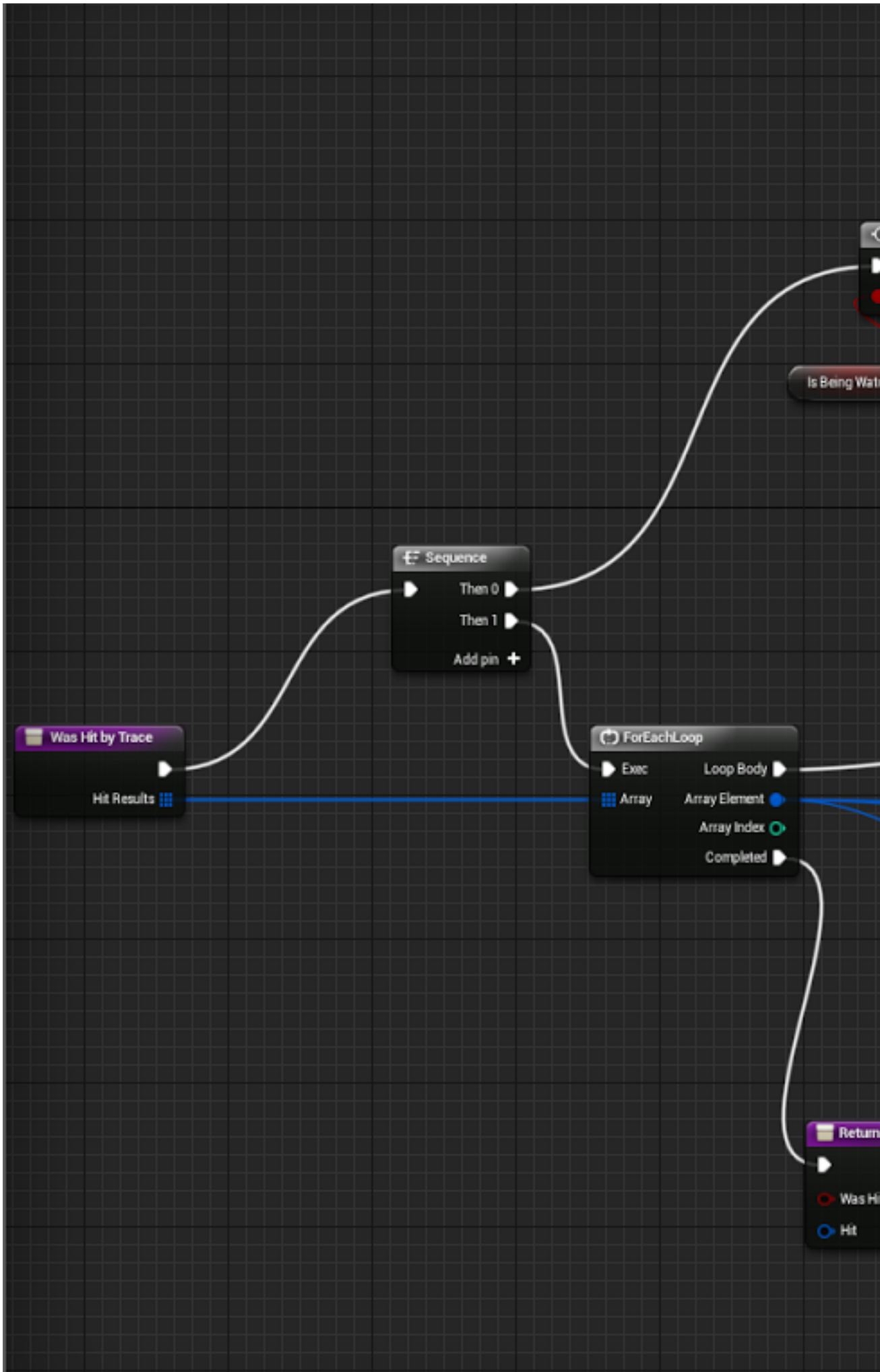


Een ander voorbeeld is de volgende functie die bepaald of de trigger van een LookEventsComponent onderdeel was van een trace.

```
FHitResult ULookEventsComponent::WasHitByTrace(const TArray<FHitResult>
    HitResults)
{
  UObject* Trigger = (bIsBeingWatched) ? UnSeenTrigger : SeenTrigger;

  for (auto& HitResult : HitResults)
  {
    UObject* HitObject = (Trigger->IsA(AActor::StaticClass())) ? Cast<
    UObject>(HitResult.GetActor()) : Cast<UObject>(HitResult.GetComponent()
    );

```

```
    if (HitObject == Trigger)
    {
      if (bShouldDrawTriggerDebug) {
        DebugTrigger(HitObject, 0.1f);
      }
      return HitResult;
    }
  }
  return FHitResult();
}
```

In vergelijking met:


Sequence
Then 0
Then 1
Add pin
Was Hit by Trace
Hit Results
ForEachLoop
Exec
Array
Loop Body
Array Element
Array Index
Completed
Hit

Voor complexe conditionele logica is c++ sneller te begrijpen en geeft meer mogelijkheden om dit te versimpelen.

4.1.3 Hoe

Met de hoe vraag word de interne werking van een functionaliteit bedoelt waaronder ook de integratie met de rest van het programma, denk aan integratie met de rest van de engine, lifecycles, overerving en performance.

Complexe algoritmes zijn altijd makkelijker in C++ voor zowel leesbaarheid als onderhoudbaarheid. Daarnaast is de scheiding van algoritmes met de rest van de code belangrijk voor het scheiden van complexiteit.

4.1.4 Onderhoud en Performance

Naast het schrijven van de logica zijn performance en onderhoud twee onderwerpen die door elk aspect van een spel verwikkeld zijn en daarom ook onderdeel moeten zijn voor onze keuze van workflow.

4.1.4.1 Onderhoud

Naast een kleine verbetering in leesbaarheid heeft het schrijven van logica in c++ een ander belangrijke voordelen. Namelijk de voordelen van een code editor. Dit geeft uitgebreide debug, zoek en meta informatie.

De errors die de Unreal Engine 4 voor je genereed zijn niet altijd even duidelijk, voornamelijk errors die pas tijdens het packagen van een project ontstaan, en het vinden van een foutieve Blueprint kan extreem lastig zijn. Bijna elk element in de Unreal Engine 4 kan Blueprints code bevatten. En elk element op zich kan die Blueprints weer in kleinere elementen verdelen. Deze elementen bevinden zich tussen alle andere soorten assets zoals Meshes, Animaties, Deciscion Trees, Materials, Textures etc en het zoeken van een onbekende fout hierin is extreem lastig.

Als de foutieve code in c++ had gestaan kan er altijd door middel van een stack trace gekeken worden in welke functie het fout gaat en kan er gezien worden welke functie die functie aansprak en verder omhoog.

Het auto aan vullen van functie namen en informatie tonen over functies door middel van de JavaDoc opmerking notatie versimpelt het schrijven van complexe code die gebruikt maakt van de Unreal Engine 4.

4.1.4.2 Performance

Een ander belangrijk onderdeel van de hoe logica is performance. De performance van Blueprints is namelijk lager dan dat van c++. In de hoe en wat vraag is dit verschil compleet onmerkbaar maar in code wat vaak, bijvoorbeeld in de loop van een game, word uitgevoerd kan het verschil merkbaar worden. In c++ is er ook veel meer controle over de manier waarop de computer logica interpreteert, in Blueprints is dit een stuk minder duidelijk. Dit zorgt ervoor dat op een lager niveau optimalisaties mogelijk zijn.

Daarnaast is het makkelijker om complexe optimalisaties te schrijven op een onderhoudbaar manier. Als er bijvoorbeeld een performance probleem ontstaat in het aantal raytraces wat de LookEventsComponents nodig hebben is het mogelijk om een cache te schrijven voor de ray traces waar de LookEventsComponent een trace uit vraagt als deze trace dan al eerder door een andere LookEventsComponent berekend is krijg hij de resultaten van die trace inplaats van een nieuwe trace te maken.

4.1.5 Conclusie

Voor de wanneer vraag is Blueprints voor zowel de programmeur als de niet-programmeur altijd de beste optie. Niet alleen is het leesbaarder maar ook flexibeler doordat de logica op een logische manier achter elkaar staat.

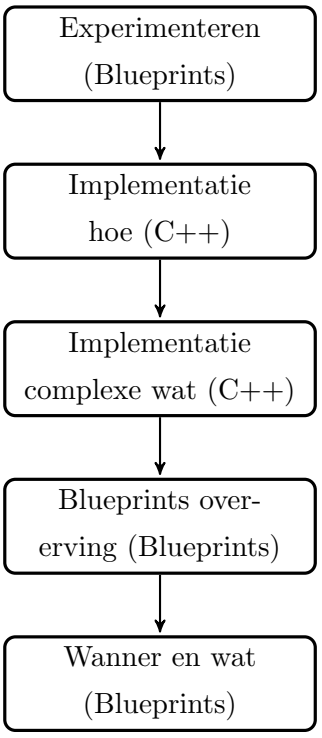
Voor de hoe vraag is C++ altijd de beste optie. Onderhoudbaarheid en performance spelen hier een belangrijke rol.

Voor de wat vraag heeft de niet-programmeur geen optie maar de programmeur wel. Voor triviale code kan er gekozen worden voor Blueprints maar voor complexe conditionele logica heeft C++ de voorkeur.

4.2 Workflow

Voor onze workflow willen wij de complexiteit voor de niet-programmeurs zo laag mogelijk houden met zo veel mogelijk vrijheid. Daarnaast is flexibiliteit belangrijk voor het makkelijk en veilig experimenteren en itereren.

Gebaseerd op het verdelen van de gameplay logica in de drie vragen hoe, wat en wanneer word er voor de volgende workflow gekozen.



Elk component zal beginnen als een imperfecte Blueprint logica. In dit stadium is de performance en onderhoud niet belangrijk, en hoeft het niet eens compleet functioneel te zijn. Er kan direct ge-experimenteert en getest worden. Dit kan niet alleen waardevolle informatie opleveren voor het design aspect van een project maar plaats de logica ook in de context van het project. Vaak komt hier gewenste functionaliteit uit die tijdens het bedenken over het hoofd gekeken is.

Nadat het duidelijk is dat het component gebruikt gaat worden in het project en de gewenste functionaliteiten duidelijk zijn kan er een fatsoenlijke en efficinte C++ implementatie geschreven worden.

De C++ implementatie word vervolgens overerft door een Blueprint die de aanvullende logica toevoegt.

Ook in gevallen waarin de Blueprint niet iets toe te voegen heeft is het belangrijk om toch een Blueprint te maken van de C++ code. Er kan hierdoor namelijk makkelijk mee gexperimenteerd worden en niet-programmeurs hebben hierdoor ook de mogelijkheid om iets toe te voegen of een standaard waarde te veranderen.

4.3 Versie Controle

Versie controle is een belangrijk onderdeel van elk ICT project en al valt dit buiten de scope van deze scriptie word er hier een korte vermelding van gemaakt.

Versie controle is een belangrijk onderdeel in samen werken aan code en speelt een belangrijke rol in het onderhoud van code. Er zijn tools in Unreal Engine 4 om versie controle tools zoals git en svn op Blueprints, en assets, toe te passen. Voor deze scriptie is de plugin ontwikkeld met behulp van git maar is er niet gekeken naar een manier waarop niet-programmeurs hier ook mee kunnen werken.

Tijdens de interviews[?] werd het duidelijk dat er geen bestaande kennis van versie controle aanwezig was in de visuele teams van DPI. Het kiezen, opzetten en leren van versie controle en de integratie hiervan met de Unreal Engine 4 valt buiten de scope van deze scriptie.

Appendix A

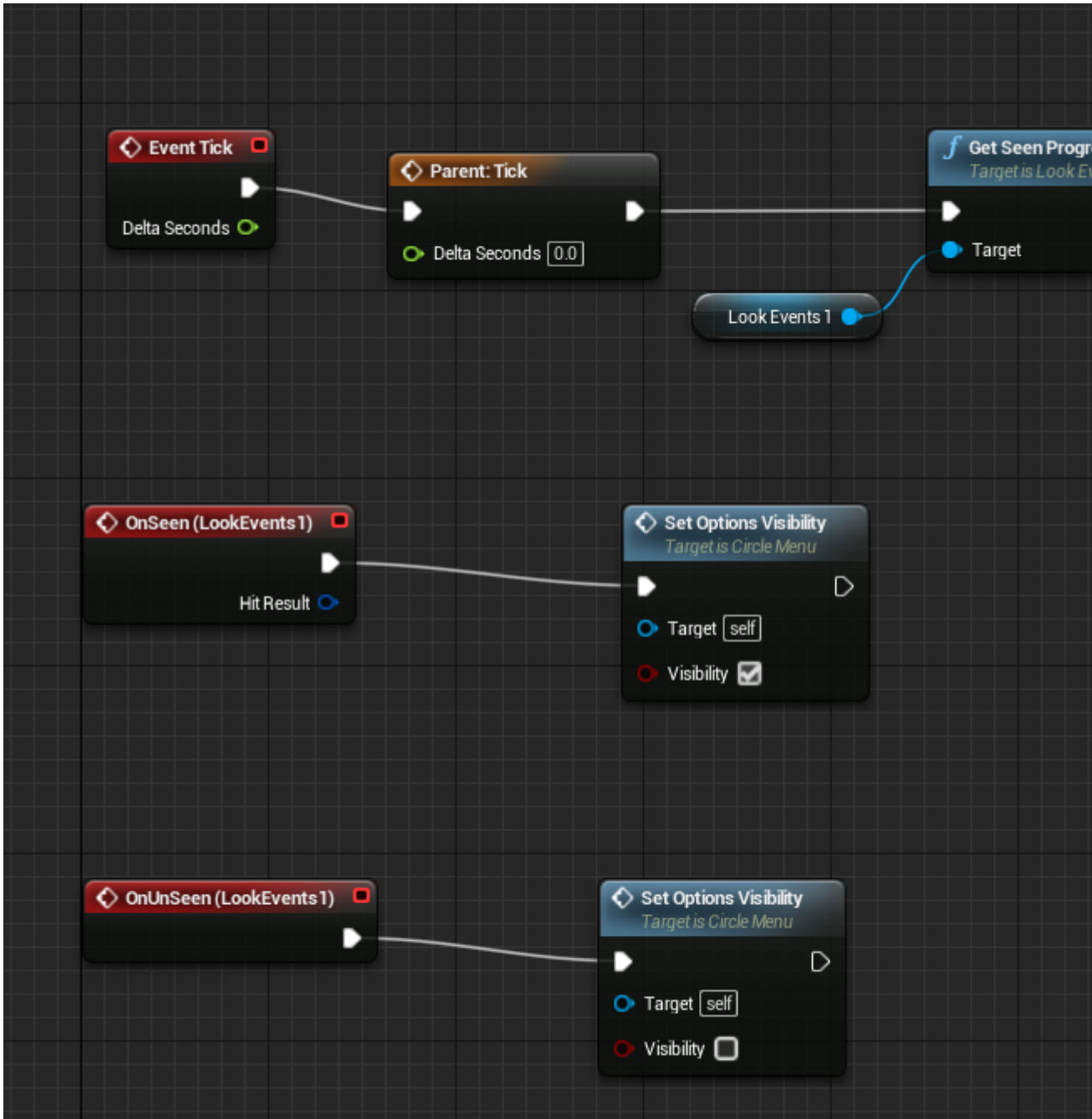
Conditional logic van Tick functie van LookEvents in c++

```
void ULookEventsComponent::TickComponent(float DeltaTime, enum ELevelTick
    TickType, FActorComponentTickFunction *ThisTickFunction)
{

  if (bActive != true || (bShouldUsedOnce && TimesUsed > 0) || bIsInTimeOut
     == true)
  {
    return;
  }

  FHitResult HitResult = WasHitByTrace(Trace());

  // TODO: This should be a proppert check if the HitResult was a hit
  if (HitResult.Actor.IsValid())
  {

    if (bIsInUnSeenDelay)
    {
      // If we are in a un seen delay we should clear the handler and
    pretend it never happend
    }

    if (!bIsInSeenDelay)
    {
      if (SeenDelay > 0 && !bIsSeenDelayFinished)
      {
      }else if (!bIsBeingWatched)
      {
      }
    }
```

```
28    }
      else if (bIsInSeenDelay)
30    {
      }
32    else if (bIsBeingWatched)
      {
34      if (UnSeenDelay > 0)
        {
36        if (bIsUnSeenDelayFinished)
          {
38        }
          else if(!bIsInUnSeenDelay)
40        {
          }
42      }else
        {
44      }
      }
46  }
```

Appendix B

Conditional logic van Tick functie van LookEvents in Blueprints



Bibliography