Dania
Academy of Higher Education

Academy Profession (AP) Degree in Computer Science

TITLE:

Breakout in Unreal Engine 4

PROJECT PERIOD:

GrnXXdatXX,
December 2015

PROJECT GROUP:

XX

STUDENTS:

Nichlas Bruun
Mathias Forsberg
Bjarne Kristensen

SUPERVISOR:

Jonathan

COPIES: 1

REPORT PAGES: 34

APPENDIX PAGES: 1

TOTAL PAGES: 35

ABSTRACT:

Problemformuleringsspørgsmålet?

Hvorfor dette emne er spændende.

Hvad vi har lavet og fundet ud af.

Indhold

Kapitel 1

Forord

Denne rapport har til formål at fortælle om processen i skabelsen af et computerspil med navnet Unreal Breakout. Rapporten er skrevet som 4. semesters eksamensopgave på datamatiker studiet ved Erhvervsakademi Dania, en eksamen som omhandler systemudvikling. Hvert stykke i rapporten vil have forfatteren på stykket nævnt i forfatterlisten. Der vil være kildehenvisninger vist med [1] hvor kilden vil være at finde under kildelisten. Fodnoterne nederst på de individuelle sider vil være brugt til eventuelle udybninger af elementer i teksten.

Kapitel 2

Problemformulering

2.1 Problemformulering

Hvordan udvikler man et breakout type spil i Unreal Engine 4?

- Hvilke analytiske værktøjer vil være værd at benytte?
- Hvilke systemudviklingsværktøjer er gode at bruge i sådan et projektforløb?
- Hvilke Unreal Engine 4 værktøjer kan optimere udviklingen?

2.2 Afgrænsning

Da projektets varighed er 4 uger, vil et fuldt testet og balanceret spil nok ikke kunne nås at lave, samtidigt med der skal skrives en systemudviklingsrapport. Der er fokus på koden i spillet, så grafikken er ikke nødvendigvis noget der vil blive brugt tid på, lyddessign er heller ikke en prioritet. Der vil heller ikke være tid til markedsanalyse, dette er fravalgt grundet produkt og systemudvikling er i fokus. Undervisningen har bestået af en uge, og herefter et projektforløb på en uge. Derfor vil noget af tiden også blive benyttet, til at blive familiær med nogle af unreal engine 4's værktøjer.

Kapitel 3

Introduktion

3.1 Unified Process

Unified Process forkortes også UP og er en systemudviklingsmetode, der blev udviklet i slutningen af 90'erne af Ivar Jacobsen, Grady Booch og James Rumbaugh. I Unified Process gøres der brug af iterationer som projektet skal udvikles i, et større projekts iterationer varer typsik 2 til 6 uger. Når man udvikler et stykke software med Unified Process skal man igennem fire faser. Inception fasen hvor man forbereder hvilke ting der skal laves og danner et overblik over de forskellige opgaver der skal løses. Den næste fase kaldes Elaboration, i denne fase udvikles de centrale dele af programmet samt de dele der er vurderet sværest at lave, da de skal udvikles tidligt i processen så de ikke bliver udviklet under tidspres. Elaboration fasen er længere end Inception fasen, når fasen er slut vil kerneelementerne af programmet være udviklet. Efter Elaboration kommer Construction fasen, her udvikles de elementer i systemet som skal til for at programmet bliver færdigt, samt at programmet testes løbende igennem fasen. Construction fasen består af mindre iterationer, og det er også fasen hvor man begynder at alpha teste programmet. Udover tests bliver der også udført performance tuning og dokumentation i Construction fasen. Sidst men ikke mindst er der Transition fasen, dette er fasen hvor programmet bliver udgivet, Transition fasen kan have flere iterationer hvor programmet får anmeldelser og feedback inden det bliver udgivet.

3.1.1 Vores Forløb med Unified Process

I processen med at skabe Unreal Breakout har det ikke været muligt at nå alt som Unified Process indebærer, dog har det stadig været muligt at følge det. Inception fasen blev fuldført med objekt orienteret analyse og design, samt udarbejdelsen

af et Gantt-chart, som estimere tiden der skal bruges på de forskellige dele af projektet. Der blev også udarbejdet en risikovurdering i inceptionfasen som skulle vurdere de forskellige risici der er ved at lave et større projekt, samt sætte tal på hvor alvorlige problemerne der kan opstå er.
I Elaboration fasen blev det lavet én iteration for at få grundelementerne i spillet lavet, iterationen varede i alt 5 dage og resulterede i at spillet blev til et spilbart. Der blev i fasen taget højde for hvilke elementer der ville være tidskrævende at udvikle og disse elementer blev lavet først i fasen.
Efter Elaboration fasen begyndte Construction fasen. Construction fasen bestod af én kort iteration hvor der blev fortaget en Blackbox test på programmet samt rettet småfejl.
Transition fasen blev ikke taget i brug i processen med at lave Unreal Breakout da produktet ikke skulle leveres til en kunde og først får ekstern feedback til eksamen.

3.2 Process

I udviklingsforløbet er der planlagt at benytte Unified Process, og eksamens projektet kommer til at illustrerer tre små UP-iterationer. I disse iterationer vil der være fokus på planlægning og kode. Planlægningen vil bestå både af UP-, objektorienteret analyse- og objektorienterede design-værktøjer. Nogle af de værktøjer der vil være taget i brug, er f.eks. et Gantt-chart til tidsplanlægning og et analysediagram til overblik af objekter i programmet. Disse værktøjer og deres brug, vil blive beskrevet i systemudviklingsrapporten. Efter planlægningen og rapportskrivningen vil udviklingsprocsessen træde i kraft, hvor selve spillet vil blive udviklet. Efter udviklingsprocessen vil der være en kort testfase, hvor diverse fejl og mangler vil blive udredt. Til sidst vil udviklingsprocesen samt refleksioner og konklusioner blive skrevet i systemudviklingsrapporten. En kort opsummering i opremsning: Planlægning, rapportskrivning, udvikling, test og til sidst rapportskrivning.

3.3 Versionsstyring

Versionsstyring er et værktøj, der gør det nemt for flere personer at arbejde på de samme filer, og gør at der er backups man kan vende tilbage til. Ved versionsstyring er der altid mulighed for at tage en ældre version af filerne, hvis der skulle være noget som er gået galt i en nyere version. Når der er to eller flere som sidder og arbejder på samme program, gør versionsstyring det muligt at sammenflette text baserede filer automatisk, men også manuelt skulle der være konfikter på linier.

Vi har anvendt versionstyring til udarbejdelsen af rapporten, men ikke spillet. Rapporten består af mange rå tekst filer, og disse er optimale at anvende versions-

tyring på. Spillet består til gengæld af blueprints og levels til Unreal Engine 4, og ved disse har vi ikke selv kontrol over tekst linier som ved ren kode. Derfor kan vi ikke bruge vores egne versionsprogrammer til at sammenflette koden mellem 2 udgaver af den samme level eller blueprint. Unreal engine har et versionstyring integreret som hedder Perforce, så hvis man skal bruge det skal man tilmelde sig en tjeneste der passer til det. Eller selv betale for at sætte en server op og betale licenser. Det virker ikke til at være en færdig feature i Unreal Engine 4 endnu, og det ser ud til at der ofte er spærgsmål på deres forum om hvordan det virker og at det crasher eller ikke gør noget[1]. Vi har valgt at bruge Github som depot for vores rapport tekst, da det er gratis at anvende ved offentlige projekter. Skal koden være privat skal man betale for en opgraderet bruger[3]. Github har også selv deres egen git klient som man kan bruge, men man kan også bruge andre programmer der understøtter git protokollen.

3.4 Objekt Orienteret Analyse og Design

Efter problemformuleringen, afgrænsningen og spil typen; Breakout var bestemt, satte udviklingsteamet sig sammen og lavede et analysediagram. Analysediagrammet er et godt værktøj til, at få alle udviklere på samme spor og tankegang omkring projektet. Analysediagrammet lister produktets objekter, og deres relationer til hinanden. På denne måde fik udviklingsholdet snakket sammen om forholdet imellem de forskellige objekter, hvilket vil lede til en mere hensigtsmæssig udvikling, da hele udviklingsholdet får samme forståelse for hvad hvert objekt kommer til at indebære. Efter dette blev use cases udviklet, som beskriver hvilke funktioner brugeren af produktet har adgang til. Use case-værktøjet udpensler derfor hvilke funktioner produktet skal indeholde set fra brugerens synspunkt. Dette giver udviklingsholdet en sans for hvilke funktionaliteter har en sammenhæng mellem objekter. For at gøre dette helt klart, udvikles en hændelses- og funktionstabel. Disse værktøjer udmærker sig ved, at klarificere hvilke objekter i systemet der har fælles funktionalitet. Udover dette bruges værktøjerne også til at specificere funktionstyper, og kompleksiteten af disse. Til sidst i denne fase laves en systemdefinition, med værktøjet "FACTOR"til hjælp. Systemdefinitionen er en kort tekst der beskriver systemets facetter, dette kan bl.a. være systemets ansvar over for brugeren eller teknologien benyttet til at udvikle systemet. Efter brugen af alle disse værktøjer, er systemet til udvikling klarlagt og udviklingen kan begyndes.

Kapitel 4

Object Orienteret Analyse

4.1 Systemdefinition

4.1.1 FACTOR

Functionality
-Bevægelse af Paddle, Fjernelse af bricks, Scoring, Liv.
Application Domain
-Klassisk arkade spil, fokus på kode og funktionalitet.
Condition
-Udvikles til Windows styresystemer, baseret på klassisk arkade spil.
Technology
-Udvikles i Unreal Engine 4, styres med keyboard og har ikke krav til kraftig PC.
Objects
-Paddle, Ball, Bricks.
Responsibility
-Et spilbart produkt.

4.1.2 Sytemdefinition

Unreal Breakout er et klassisk arkade spil, som giver spilleren mulighed for at opleve det klassiske gameplay fra Atari Breakout. Som består af at spilleren bevæger en "paddle"i bunden af skærmen, for at forsøge at holde en bold inden for skærmens rammer, samtidigt med at der skal opnås point ved at fjerne "bricks"i toppen af skærmen. Der vil være fokus på spillets kode og funktionalitet. Spillet udvikles til Windows PC i Unreal Engine 4, og styres med keyboard. Spillet vil kunne køre på en Windows PC købt indenfor de sidste 5 år. Ansvaret over for spilleren vil være et meget basalt virkende spil.

Tabel 4.1: Funktionstabel

Navn	Kompleksitet	Type
Start Game	simpel	update
Exit Menu	simpel	update
Exit Game	simpel	update
Release Ball	medium	update / compute
Bounce Ball	medium	update / compute
Move Paddle	simpel	update / compute
Break Brick	simpel	update
Change Score	simpel	update / compute
Show Score	simpel	read

4.2 Funktionstabel

Der er blevet udarbejdet en funktionstabel (se tabel 4.1) ud fra de funktioner, som er planlagt at skulle indgå i programmets første version. Disse funktioner er inddelt efter navn, kompleksitet og type. Navnet er hvad funktionen kommer til at omhandle og kompleksitet er hvor udfordrende det vil være at lave den pågældende funktion. Typen er til at bestemme, om det er noget der skal opdateres, læses, sende et signal eller beregne noget. Der er ikke noget komplekst i spillet, der er to funktioner som menes at have en kompleksitet på medium, dette relatere til bolden interaktion med andre objekter. De resterende funktioner menes at være simple at implementere.

4.3 Analyse Diagram

4.3.1 Menu

Menu klassen kommer til at være et "level blueprint"i Unreal Engine. Den skal håndtere indlæsning af menupunkter, i form af knapper og baggrundsgrafik. Menuens blueprint kommer til at håndtere "OnClick events"hvilket i dette tilfælde er en "Play-"og en "Quit-knap". Som i samme nævnte rækkefølge, kommer til at starte spillet, og lukke spillet.

4.3.2 Camera

Camera håndtere spillets synspunkt, altså hvad spilleren kan se. Kameraet kan vælge ikke at vise spilleren nogle objekter, som der muligvis har været behov for under udviklingen af spillet. Kameraet kan beskrives som en support klasse, da

den ikke direkte ændre på nogle elementer, men bare viser eller ikke viser dem. Samme blueprint vil være brugt både på kameraet i menuen, og kameraet i selve spillet.

4.3.3 GameWorld

GameWorld vil ligesom Menu-klassen, være et "level blueprint". GameWorld kommer til at være roden til alt spillets logik, og skal bl.a. indlæse alle spillets elementer når spillet startes, og når spillet skifter bane. Klassen kommer også til at lave funktionskald til de andre klasser der er relevant i spillet f.eks. Brick og Paddle.

4.3.4 UI

UI-klassen vil holde styr på at vise og opdatere brugergrænsefladen i spillet. Som i dette tilfælde vil være at vise og opdatere spillerens point og liv.

4.3.5 Brick

Brick-klassen kommer til at ligge på alle de objekter spilleren skal forsøge at ramme med bolden. Klassen vil håndtere at når bolden kollidere med en "Brick"vil denne Brick forsvinde, og der vil blive lavet de korrekte funktionskald til at opdatere spillerens score, alt efter hvilken type af Brick bolden rammer.

4.3.6 Paddle

Paddle-klassen kan tolkes som selve spillerens klasse, denne klasse vil håndtere input fra spilleren. Det vil sige at når spilleren trykker på venstre eller højre pil, bevæger Paddlen sig i den korrekte retning. Den vil også give de korrekte funktionskald til bolden, alt efter hvilken del af Paddlen bolden rammer, og give den korrekte vinkel med til bolden. Når spillet startes eller når spilleren dør, Vil Paddlen også kunne bruges til at skyde bolden afsted, da bolden er stationær på Paddlen i begyndelsen. Bolden skydes afsted fra paddlen ved brug af mellemrumstasten.

4.3.7 Ball

Ball er bolden i spillet, og klassen vil håndtere bolden, samtidigt med de tidligere nævnte funktionskald. Bolden vil også stå for vind/tab-konditionen i spillet, ved at tjekke om den er udenfor bunden af skærmen, og lade spilleren miste liv hvis den er udenfor.

Tabel 4.2: Hændelsestabel

Events/Klasser	Paddle	Ball	Brick	Menu	Gameworld	Camera	UI
Player clicks on menu obj				X			
Player moves paddle	X				X		
Player releases ball	X	X			X		
Ball collides with object	X	X	X		X		
Score updates		X	X		X		X

4.4 Hændelsestabel

En hændelse, i objektorienteret kontekst, er en situation hvor en eller flere objekter er involveret. Det er f.eks. hændelser hvor spilleren giver et input som objekter i spillet skal agere på. Det kan også være hændelser i spillet som sker når to objekter kolliderer med hinanden og en aktion ønskes. I tabel 4.2 er der opstillet hvilke hændelser vi regner med vil ske, og hvilke objekter der kan være involveret i disse hændelser.

4.5 Use Cases

Start Spillet

Scope:
I menuen.

Description:
Spilleren trykker på *play*-knappen med musen, i menuen for at komme til *gameworld*.

Preconditions:
Spilleren skal befinde sig i menuen for at use casen kan startes.

Success Guarantee:
Når use casen er opfyldt vil spillet gå fra menuen til *gameworld*.

Main Success Scenario:
Spilleren befinder sig i menuen og klikker med musen på *play*-knappen og *gameworld* vises frem på skærmen.

Extensions:

Befinder spilleren sig allerede i *gameworld* vil det ikke være muligt at trykke på *play*-knappen da den ikke er vises i *gameworld*.

Luk Spillet

Scope:
I menuen.

Description:
Spilleren kan lukke spillet ved at trykke på *exit*-knappen med musen.

Preconditions:
Spilleren skal befinde sig i menuen for at use casen kan startes.

Success Guarantee:
Når use casen er opfyldt vil spillet blive lukket.

Main Success Scenario:
Spilleren befinder sig i menuen og klikker på *exit*-knappen med musen, hvorefter spillet vil lukke.

Extensions:
Befinder spilleren sig i *gameworld* vil *exit*-knappen ikke være vist og spilleren kan derfor ikke interagere med den.

Bevægelse af Paddle

Scope:
I *Gameworld*.

Description:
Spilleren kan ved hjælp af højre og venstre piletast bevæge *Paddlen* til henholdsvis højre og venstre.

Preconditions:
Spilleren har startet spillet igennem menuen.

Success Guarantee:

Bliver use casen opfyldt korrekt vil *Paddlen* blive bevæget til siderne i takt med at spilleren trykker piletasterne ned.

Main Success Scenario:
a. Spilleren trykker højre piletast ned og *Paddlen* bevæger sig til højre.
b. Spilleren trykker venstre piletast ned og *Paddlen* bevæger sig til venstre.

Skyd

Scope:
I *Gameworld*.

Description:
Bolden skydes fra *Paddlen* ved brug af *space*-knappen, dette kan kun gøres når spillet startes eller efter bolden har været uden for spilbanen.

Preconditions:
Spilleren har startet spillet igennem menuen og har ikke skudt endnu, eller bolden har lige været udenfor spilbanen.

Success Guarantee:
Bliver use casen opfyldt korrekt vil bolden blive skudt fra *Paddlen*.

Main Success Scenario:
Spilleren har startet spillet og bolden er stadig placeret på *Paddlen*, eller bolden har lige været uden for spilbanen. spilleren trykker på *space*-knappen og bolden bliver skudt fra *Paddlen*.

Extensions:
Bolden er blevet skudt fra *Paddlen* og har ikke været udenfor spilbanen og kan derfor ikke blive skudt fra *Paddlen* igen da den ikke er placeret der.

Gå tilbage til menuen

Scope:
I *Gameworld*.

Description:

Spilleren går tilbage til menuen fra *gameworld* ved at trykke på *escape*-knappen.

Preconditions:
Spilleren har startet spillet og befinder sig i *gameworld*.

Success Guarantee:
Bliver use casen opfyldt korrekt vil spilleren blive vist menuen igen og *gameworld* vil lukke.

Main Success Scenario:
Spilleren har startet spillet og trykker på *escape*-knappen, herefter vil menuen blive vist og *gameworld* vil stoppe.

Extensions:
Befinder spilleren sig allerede i menuen vil *escape*-knappen ikke have nogen effekt.

Kapitel 5

Object Orienteret Design

5.1 Gameplay

Unreal Breakout er et spil med et bat[1], en bold[2], en masse brikker[3], tre lukkede sidder, og en åben side i bunden. Formålet med spillet er at få bolden til at ramme alle brikkerne og sørge for at bolden ikke ryger ud af banen i bunden hvor spillerens bat og den åbne side er. Når man rammer en brik, går den i stykker eller skifter farve, når en brik rammes vil der blive tilføjet point til spillerens score. Når spillet starter har man 3 liv(bolde) og bolden er låst fast på spillerens bat nederst i midten af skærmen. Når man trykker på en dertil bestemt knap skyder man bolden afsted, og med nogle andre knapper bevæger man battet fra side til side for at undgå at bolden ryger ud af spilområdet. Hvis bolden ryger ud af spilområdet, mister man et liv(bold) og en ny bold bliver låst fast til battet og er klar til at blive skudt afsted. Når spilleren har mistet alle liv og den sidste bold ryger ud af spilområdet, slutter spillet. Får man til gengæld fjernet alle brikker, vindes banen og den samme bane kommer igen uden at ændre på antal liv spilleren har tilbage eller nulstille scoren, og bolden bliver igen fastlåst til battet.

5.2 Grafik

Spillets grafik er fundet på *opengameart.org*, som er en hjemmeside hvor folk lægger grafik op til fri afbenyttelse. Dette betyder at det er uden licens men man kan dog donere penge til de brugere der har lagt grafikken tilgængelig på siden. Nogle af grafikerne på siden vil dog gerne have deres navn nævnt hvis deres grafik bruges. Breakout er et gammelt spil udviklet af Atari, derfor er der fundet grafik som er

[1]Paddle på engelsk, bliver brugt i vores paddle klasse.
[2]Ball på engelsk, bliver brugt i vores ball klasse.
[3]Bricks på engelsk, bliver brugt i vores bricks klasse.

tro mod det klassiske spils grafik. Grafikken ligger på nogle spritesheets som er kollager af billeder som bliver brugt til at skabe de forskellige elementer i spillet. Når spritesheetet er delt op vil de forskellige brikker blive bygget op til en bane. Eftersom spilleren rammer brikkerne med bolden vil de skifte farve, farverne indikere hvor mange gange en brik skal rammes for at blive ødelagt. De grafiske elementer er ikke noget der vil blive sat stor fokus på i spillet, da det er spillets gameplay der skal være grundpillen i det.

5.3 Blueprints

Blueprints er en speciel type asset som giver en node baseret interface til at lave nye klasser og udvide klasser, så man kan scripte objekter i levels og widgets. Blueprints er et værktøj der giver designers og gameplay-programmører mulighed for hurtigt at kunne udvikle deres idéer uden at skulle skrive kode.

Det er stadig muligt selv at oprette klasser med C++ kode uden at bruge blueprints. Blueprints kan bruges til at udvide klasser som er skrevet i C++, gemme og modificere properties og håndtere objektinstanser i klasser.

Ligesom i C++ og C# har blueprints member variables/fields, member functions, og en constructor.

Der er 3 typer Blueprints:

Blueprint Class

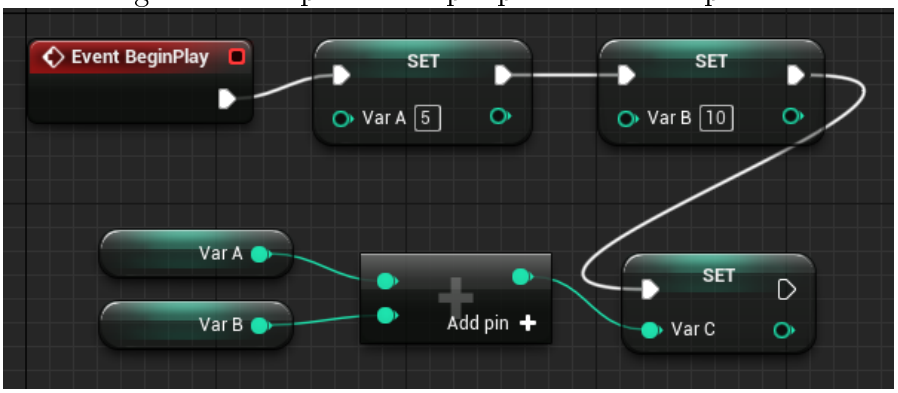
Data-Only Blueprint

Level Blueprint

Blueprint Class er det blueprint man bruger når man vil have et objekt til at gøre noget inde i et *level*. Man laver noget funktionalitet i dette blueprint og tilføjer det til et objekt i et *level*. Data-Only Blueprints indeholder kun nodes, variabler og komponenter som det har nedarvet og der kan ikke tilføjes nyt. Det den bruges til er at ændre på objekter i et *level* og lave variationer som alle har samme interface, men forskellig adfærd. Level Blueprint er et blueprint som altid findes når man opretter et nyt *level*. Det er et overordnet blueprint som eksisterer sammen med de Blueprints som sidder på objekter i banen. Man kan bruge Level Blueprint til at oprette events som andre objekters blueprints kan agere på. Man kan også oprette instanser af andre assets gennem Level Blueprint, som f.eks. et UI der skal tegnes i Camera Actor'en i banen. Se [2] for mere information.

I figur 5.1 kan ses et simpelt eksempel på hvordan noder kan sættes sammen i et blueprint. I eksemplet sættes to *private int* variabler til to forskellige tal og derefter

Figur 5.1: Simpelt eksempel på noder i blueprints.



bliver de lagt sammen og sat i en tredje *private int* variabel. Når man opretter variabler i et blueprint er de ligesom fields i C# hvor de kan defineres som private eller public. Forskellen er at blueprints altid initialiserer variabler med en standard værdi, hvor man i C# kode godt kan lade variabler være uinitialiserede.

Hvis man oversætter noderne fra blueprint eksemplet i figur 5.1 til for C# kode, vil det være noget lignende koden i figur 5.2. Unreal Engine 4 er lavet med C++ kode og alle noderne i blueprints omsættes også til C++ kode. Det er derfor at man skal *compile* hver gang man laver ændringer i noderne. Vi er ikke klar over hvordan det præcist ser ud når Unreal Engine 4 compiler et blueprint om til en klasse i C++, men vi er ret sikker på at der bliver oprettet et par hjælpe metoder og variabler i dem for at *game engine*'en kan holde styr på det hele. Der er for eksempel et *BeginPlay* event man kan gøre brug af i editoren og et *EventTick* event.

Figur 5.2: C# kode eksempel på noder i blueprints.

```
using System;

public class ExampleClass
{
    private int VarA = 0;
    private int VarB = 0;
    private int VarC = 0;

    public ExampleClass()
    {
    }

    public void BeginPlay()
    {
        VarA = 5;
        VarB = 10;
        VarC = VarA + VarB;
    }
}
```

Kapitel 6

Implementering

Noget introduktion til vores implementering.

6.1 Menu

Menuen er lavet i et Widget Blueprint hvori der er sat knapper op i en vertical box. Knapperne har tilhørende tekst som indikere deres funktion. Derudover er der sat et PNG-billede ind som baggrund for menuen. Play-knappen har et OnClick event der sørger for at skifte scene til gamemoden når der bliver trykket på den. Quit-knappen bruger en "Execute Console Command"til at lukke spillet ned nåt der bliver trykket på knappen. Se figur 6.1. Menu blueprintet ligger inde i sin egen level, hvor den er instansieret i viewporten.

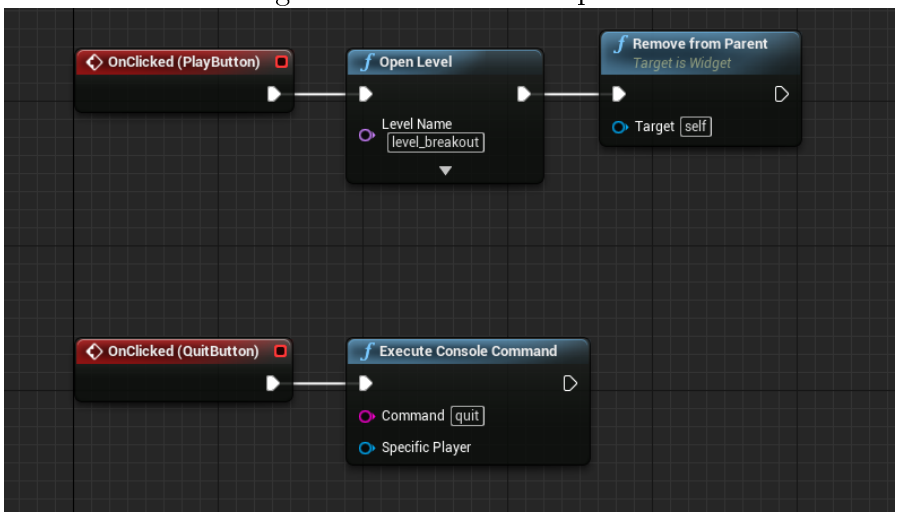
6.2 Level

Level blueprintet er banens "klasse". Denne har ikke meget, og forholdvist simpel logik. Når banen, eller selve spillet startes, bliver *EventBeginPlay* kaldt, hvilket betyder at logikken der ligger i dette event kun bliver kaldt en gang når der startes med at spille. På dette event ligger logikken til at skifte synspunktet til kameraet i selve spilbanen, og logikken til at binde UI-blueprintet til samme viewport.

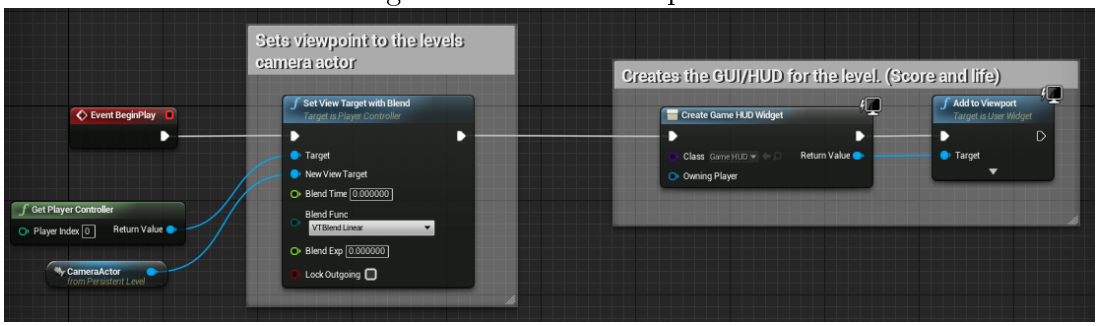
6.3 UI

GameHUD blueprint er et Widget Blueprint der viser UI mens man spiller. Den indeholder blot fire tekst felter, hvor to af dem er teksterne; "Balls"og "Score". De to andre tekst felter bliver opdateret løbende, med liv og score, når man spiller spillet. Se figur 6.3 for designet.

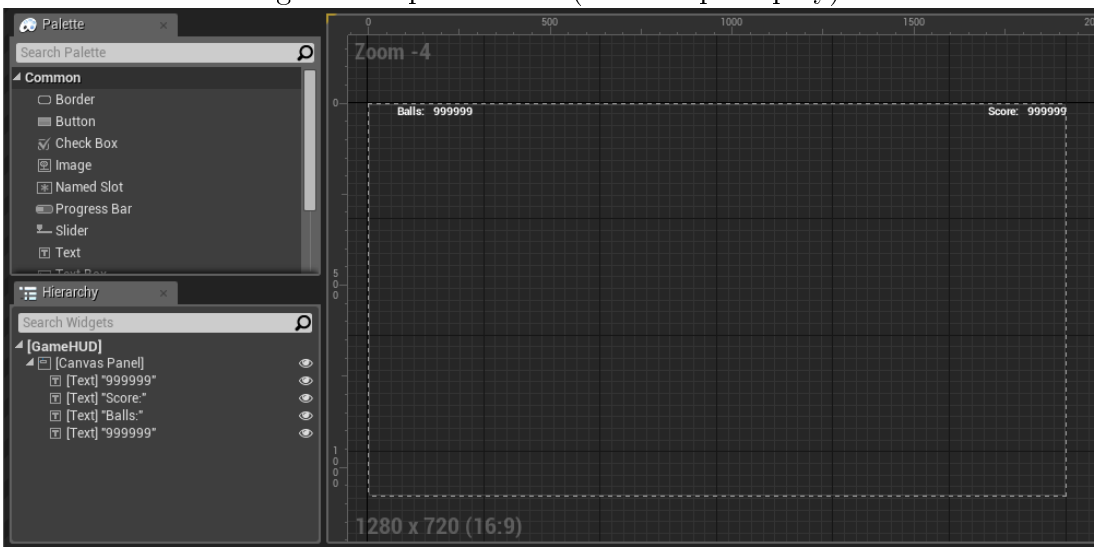
Figur 6.1: Menuens blueprint.



Figur 6.2: Banens blueprint.



Figur 6.3: Spillets HUD(Heads Up Display).

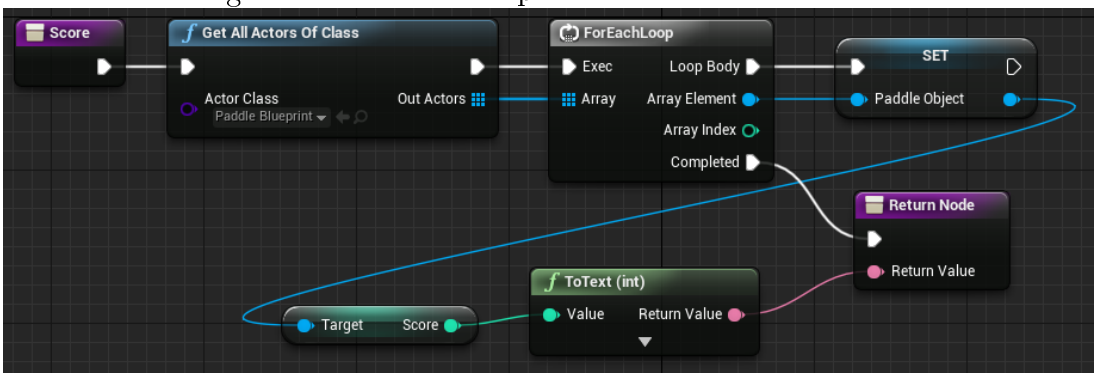


Der er to funktioner i GameHUD blueprintet; Balls og Score. Det er disse to funktioner som holder tekst felterne opdateret. Balls og Score variablerne er sat i Paddle Blueprint instansen på Paddle objektet i selve banen. For at få fat i disse variabler fra en anden instans, er man nødt til at søge efter instansen og så gemme en pointer til instansen lokalt hvor man skal bruge den. Da der kun er en paddle i banen kan vi bare tage den første instans der findes og sætte den i vores lokale variabel. Efter paddle blueprint instansen er fundet kan f.eks. Score variablen konverteres til tekst og returneres til GameHUD blueprintet hvor teksten så bliver opdateret. Se figur 6.4. Tilsvarende sker med Balls variablen i en funktion som opdaterer teksten med hvor mange liv man har tilbage.

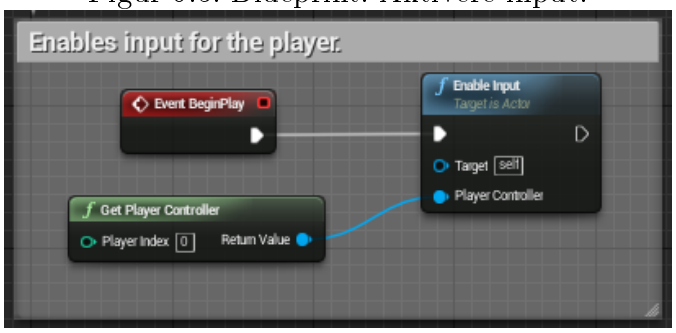
6.4 Paddle

Paddlen er inddelt i højre og venstre halvdel, dette er gjort ved lave et tjek på hvor bolden rammer paddlen. Hvis boldens X-værdi når den rammer paddlen er mindre end placeringen af paddlens midte, har bolden ramt venstre side og er X-værdie derimod større end placeringen af paddlens midte, har den ramt højre. Hele dette tjek bruges til at give bolden den rette udgangsvinkel efter kollisionen med paddlen er sket. Herefter bliver der lavet et tjek på hvor boldens tidligere placering var, hvilket bruges til at udregne hvilken retning bolden kommer fra. Et eksempel på dette kan være hvis bolden kommer fra højre side i forhold til paddlens placering og rammer højre side af paddlen vil bolden flyve tilbage i den retning den kom fra,

Figur 6.4: GameHUD opdater Score tekst funktion.



Figur 6.5: Blueprint: Aktivere input.



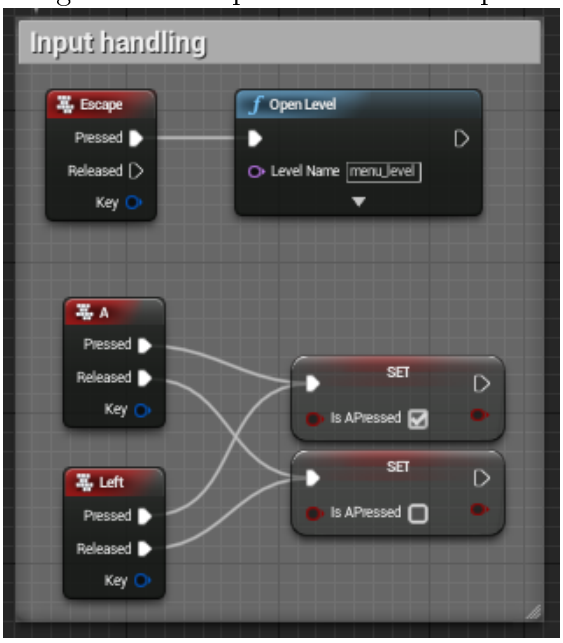
ved at vende fortegn i begge af boldens akser.

Paddlen modtager også input fra tastaturet for at kunne bevæge den fra side til side. Man skal først aktivere input på *player controlleren* for at kunne behandle input. Se figur 6.5.

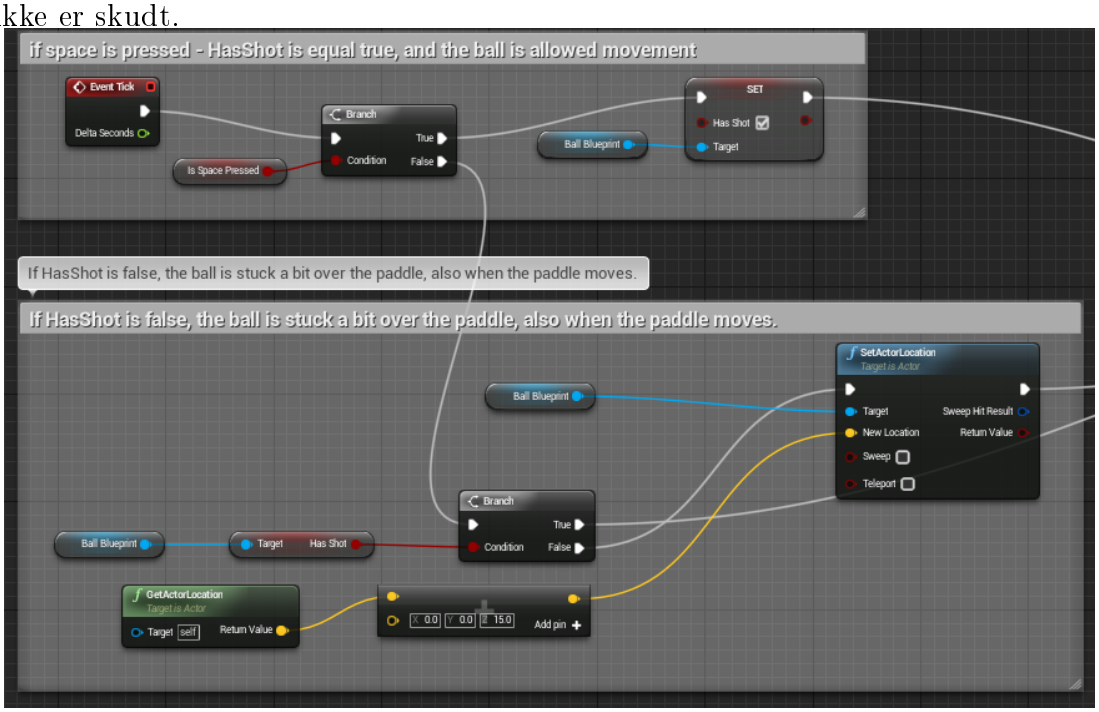
Når man trykker på enten A eller ventre pil tasterne, sætter vi en boolean variabel *IsAPressed* til true. Når spilleren så slipper tasterne igen, bliver den sat til false. Denne bliver brugt til at rykke paddlen mod ventre i banen. Se figur 6.6. Fremover bliver der refereret til A selvom det også kan være pil venstre der er trykket ned. Det er det same der bliver gjort mod højre retning med *IsDPressed*. Hvis man trykker på escape tasten loader den banen der har hoved menuen. Trykker man på space tasten bliver *IsSpacePressed* sat til true, og false når man slipper tasten.

Det første der sker i EventTick'et er at vi ser om der bliver trykket på space. Dette sætter en anden variabel så bolden kan blive skudt afsted. Hvis man ikke trykker på space og bolden ikke er skudt afsted så den flyver frit rundt, skal den låses fast lige oven over paddle'en. Se figur 6.7.

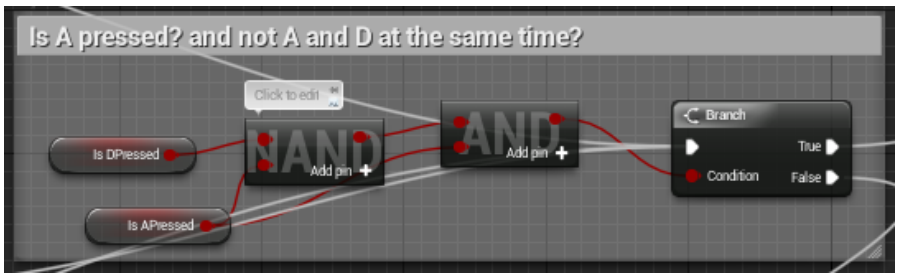
Figur 6.6: Blueprint: Håndtere input.



Figur 6.7: Blueprint: Sæt til at skyde eller få bolden til at følge battet hvis den ikke er skudt.



Figur 6.8: Blueprint: Kun aktivere hvis kun A er trykket og ikke A og B på samme tid.



Efter det kontrolleres der om A er trykket ned, men ikke A og D på samme tid. Hvis begge er trykket ned har vi ikke lyst til at den skal bevæge sig mod venstre. Se figur 6.8.

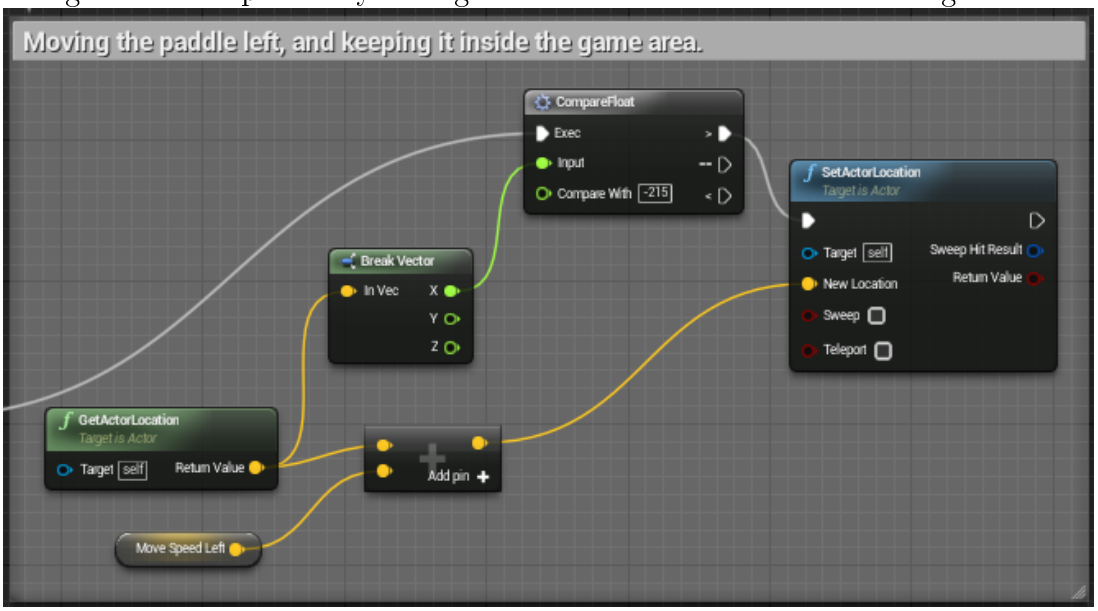
Hvis kun A er trykket kan vi bevæge paddle'en mod venstre med en given hastighed. For at den ikke skal flytte sig ud over banens sider, er der sat en minimum X værdi som den skal være over for at kunne flytte sig mod venstre. Se figur 6.9.

Noget lignende funktionalitet gør sig gældende for at bevæge sig mod højre, bare med modsatte værdier for når D er trykket ned, men ikke D og A på samme tid.

6.5 Ball

Hvis bolden ikke er blevet skudt af fra paddlen vil den være låst fast til paddlen. Derimod hvis den er skudt fra paddlen vil bolden rykke sig med den hastighed der er sat af den lokale variabel *direction vector*. Direction vector variablen bliver ændret alt efter hvad bolden kollidere med. Som tidligere nævnt i paddle afsnittet ligger logikken til kollision med paddlen i paddle blueprintet, derfor er der et tjek på om bolden kollidere med paddlen. Kollidere bolden ikke med paddlen bliver der tjekket på hit eventets *hit normal variabel*, denne værdi fortæller hvilken side af bolden kollidere. Dette kan f.eks. være hvis *hit normal X* er minus 1 er der tale om at boldens venstre side kollidere med et andet objekt. Gennem alle disse tjek kan der udregnes hvilket fortegn i hvilken akse der skal vendes i direction vectoren for at bolden får den korrekte retning efter kollision. I selve bold blueprintet ligger der også tjek på om spilleren har flere liv tilbage, eller om alle brikker er forsvundet fra spilverdenen. Dette sørger for at spilleren kommer tilbage til menuen hvis alle liv er mistet og at der bliver skabt nye brikker hvis alle brikkerne er ramt af bolden. Blueprintet indeholder også et tjek på om bolden er indenfor banen, befinder bolden sig ikke på banene mere vil der blive trukket 1 fra spillerens liv. Sidst men

Figur 6.9: Blueprint: Flytter sig mod venstre hvis over minimums grænsen.



ikke mindst ligger der også logik for inputtet til at skyde bolden fra paddlen, dette gøres ved at når der trykkes på mellemrumstasten sættes den boolske værdi i *has shot* variablen til true, og bolden går igennem det tidligere nævnte tjek.

6.6 Brick

Brick er et blueprint som nedarver fra *PaperSpriteActor*-blueprinttypen, dette vil sige at det er et blueprint der kan instansieres i verdenen med en sprite. Som sådan er det også den eneste logik der ligger på brick, årsagen til at den har et blueprint er for at Ball klassen kan kende forskel på hvad den rammer på *OnHit*-events. Da det ikke ville være hensigten at fjerne f.eks. væggen i siden af banen når den blev ramt af bolden.

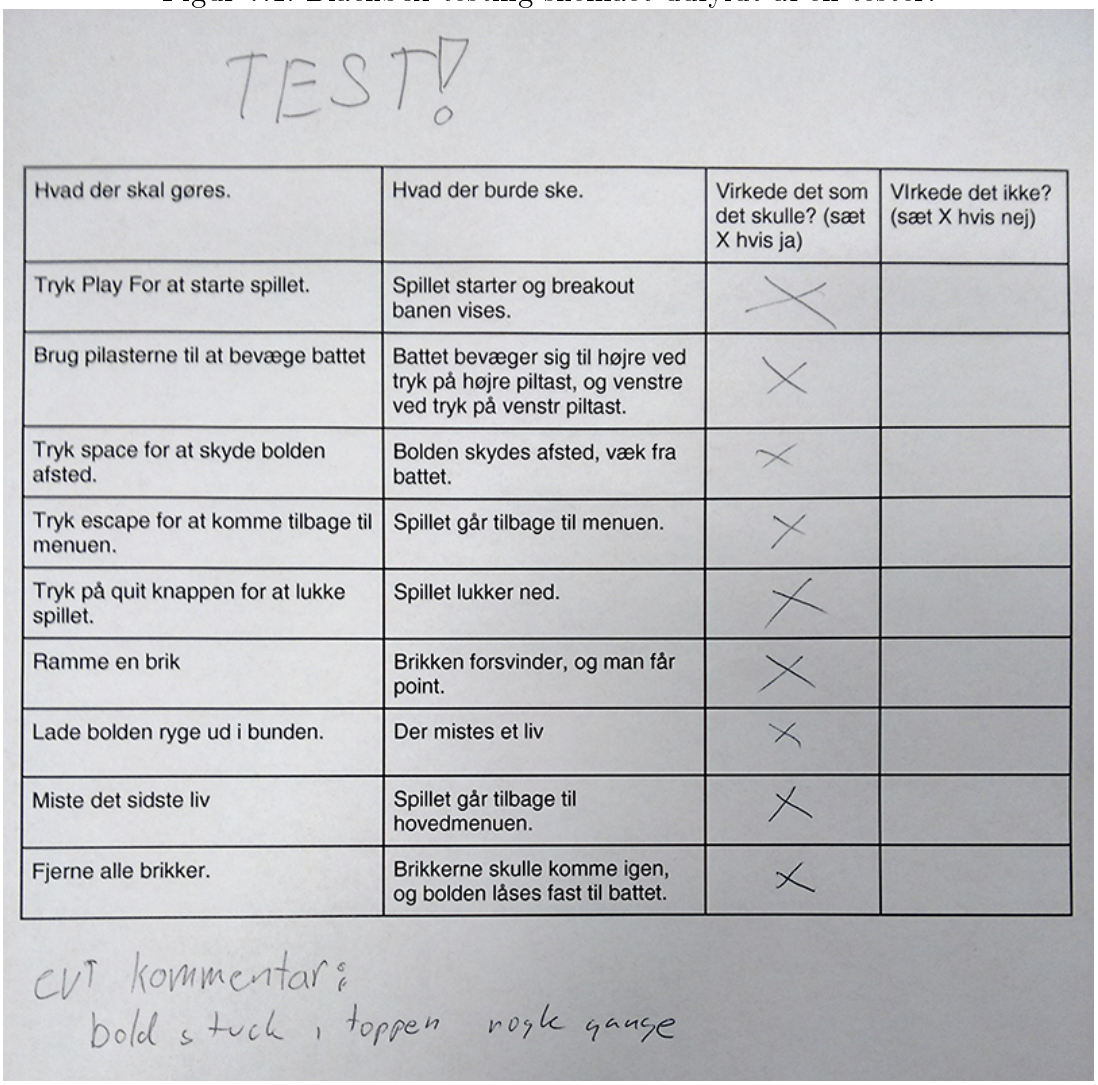
Kapitel 7

Testing

7.1 Black box test

Testningen foregik ved at have en person som ikke var del af udviklingsgruppen, til at teste specifikke elementer fra spillet. Dette blev gjort ved at lave et spørgeskema, som personen der testede spillet skulle udfylde under gennemspilningen. Se figur 7.1. Spørgeskemaet bestod af en række situationer/cases som der kunne krydses af hvis de blev opfyldt, eller eventuelt ikke blev opfyldt. Disse situationer er udviklet ved hjælp af use cases fra den objektorienterede analyse, da disse funktioner skal virke i spillet. Efter gennemspilningen har test personen krydset alle scenarierne ud, som virkende. Dog var der en kommentar om at bolden opførte sig underligt i nogle situationer.

Figur 7.1: Blackbox testing skemaet udfyldt af en tester.



TEST!

Hvad der skal gøres.	Hvad der burde ske.	Virkede det som det skulle? (sæt X hvis ja)	VIrkede det ikke? (sæt X hvis nej)
Tryk Play For at starte spillet.	Spillet starter og breakout banen vises.	X	
Brug pilasterne til at bevæge battet	Battet bevæger sig til højre ved tryk på højre piltast, og venstre ved tryk på venstr piltast.	X	
Tryk space for at skyde bolden afsted.	Bolden skydes afsted, væk fra battet.	X	
Tryk escape for at komme tilbage til menuen.	Spillet går tilbage til menuen.	X	
Tryk på quit knappen for at lukke spillet.	Spillet lukker ned.	X	
Ramme en brik	Brikken forsvinder, og man får point.	X	
Lade bolden ryge ud i bunden.	Der mistes et liv	X	
Miste det sidste liv	Spillet går tilbage til hovedmenuen.	X	
Fjerne alle brikker.	Brikkerne skulle komme igen, og bolden låses fast til battet.	X	

CVT kommentar:
bold stuck i toppen nogle gange

Kapitel 8

Refleksion

De nævnte emner i dette afsnit er noget gruppen er blevet enige om, og individuelle holdninger er der ikke blevet taget højde for i rapporten.

Sygdom
Vi har haft lidt sygdom, både en med ondt i ryggen og en med forkølelse. Da vi arbejdede meget på flextid, har det ikke påvirket processen særligt meget.

Blueprints
Taget lang tid at finde ud af at gøre nogle bestemte ting, men generelt set er det gået godt.

Rapportskrivning
Det gik godt i starten af projektet, men langsommere da produktet var lavet og der skal skrives om implementeringen og rettes til i rapporten.

Hjemmearbejde
Det har virket okay, på flextid har vi kunne arbejde når vi bedst var inspireret. Kommunikationen er lidt langsommere end hvis man sidder sammen i skolen.

UP
Vi har efter bedste evne forsøgt at holde UP-arbejdsgangen og det er lykkedes ganske udmærket.

Gruppens Samarbejde
Vi har ingen konflikter haft i gruppen. Vi har holdt opsamlingssamtaler et par gange om ugen for at høre hvor langt hver enkelt gruppemedlem var nået med sine opgaver.

Tidsplanen
Vi har lavet et Gantt-diagram som vi har forsøgt at følge. Det er gået godt med de fleste af opgaverne på tidsplanen, men den sidste rapport skrivnings periode er skredet lidt ind over julen, hvor vi gerne ville have haft den færdig inden jul.

Kapitel 9

Konklusion

Det lykkedes os at lave en prototype af et Breakout type spil i Unreal Engine 4.

Under analysefasen har vi brugt use cases, analyse-klassediagram, system definition(FACTOR), hændelsestabel, funktionsliste. Disse har været meget hjælpsomme til hurtigt at skyde projektet i den rigtige retning. Dette sørger for at alle i projektet har samme idé om hvad der udvikles. Til design af spillet har vi lavet en gameplay-beskrivelse og hentet grafik fra *Opengameart.org*.

Unified Process har givet os rigtig gode værktøjer til at planlægge projektets forløb med, så man ved hvor meget tid og arbejdskraft man har at gøre med. Hvis noget i tidsplanen så skrider, har vi hurtigt kunne reagere på det.

Til at lave selve spillet har vi brugt Unreal Engine 4 med blueprint-, sprite-, og level-editor. Sprite-editoren har gjort det let for os automatisk at splitte spritesheet billedfiler op i mindre elementer. Level-editoren har gjort det let at lave hovedmenuen og banen ved visuelt at placere objekter. Blueprint-editoren er en visuel måde at repræsentere kode(scripts), ved at oprette noder med specifik funktionalitet og forbinde dem sammen.

Referencer

[1] Epic Games. *Blueprint Merge Topics*. 2015. URL: https://answers.unrealengine.com/questions/topics/merge.html.

[2] Epic Games. *Blueprint Overview*. 2015. URL: https://docs.unrealengine.com/latest/INT/Engine/Blueprints/Overview/index.html.

[3] GitHub. *GitHub Terms of Service*. 2015. URL: https://help.github.com/articles/github-terms-of-service/.