

Dynamisk Sværhedsgrad i Videospil - DDU

Eksamensprojekt

Mathias Esmann & Bruce Esplago
ZBC Ringsted - 3.Q
DDU

Maj 2018

Resumé

This document covers the conceptualization and creation of the final exam project for "Teknik A - DDU." The project is focused on creating a video game, with the purpose of developing and including a Dynamic Difficulty Adjustment (DDA) system within it and using it to find out and answering the questions: to which degree can we change the playstyle of a player and which effect does this have on their engagement? Through an analysis of currently existing DDA systems, we found inspiration in how to create our own. This lead to the entire game being built with the DDA system in mind, the final game produced is that of a top down shooter, using the DDA system to dynamically change predefined levels, and dynamically change the content inside the levels, according to player performance and playstyle. The final result is unfortunately somewhat unreliable, given the low amount of test subjects, so it should be taken with a grain a salt. The final result showed that using the DDA system that was developed helped push players towards a more diverse playstyle by a degree of 23.75%. Suprinsingly this lead to a decrease in their overall engagement and therefore enjoyment of the game. The conclusion being that we unfortunately don't have enough test data to accurately answer our problem.

1 Titelblad

Titel: Dynamisk Sværhedsgrad i Videospil - DDU Eksamensprojekt

Navn(e): Mathias Esmann, Bruce Esplago

Fag: Teknik A - DDU

Eksaminsterming: Juni 2018

Skole navn: ZBC Ringsted, HTX

Klasse: R3q

Oplag: 3

Vejleder: Mikkel R. Nielsen & Uffe Thorsen

Afleveringsdato: 22. maj 2018

Underskrift: _____

Mathias Esmann

Underskrift: _____

Bruce Esplago

Indhold

1 Titelblad	2
2 Indledning	5
3 Baggrund	6
3.1 Problemstilling	6
3.2 Problem afgrænsning	6
4 Problemformulering	7
5 Analyse	8
5.1 Gantt og Scrum	8
5.2 Dynamisk eller halv dynamisk levels	8
5.3 Måling af engagement i computerspil	11
6 Teknisk	13
6.1 Kravspecifikation el. Feature List	13
6.1.1 Need To Have	13
6.1.2 Nice To Have	14
6.2 Design	14
6.2.1 Gameplay	15
Stealth vs Rambo	15
6.2.2 Level Design	16
Level design i dette spil	16
6.3 Level Design	16
Stealth level design	16
Rambo level design	18
6.3.1 DDA System	19
6.3.2 Fjende AI	21
6.3.3 Grafik	22
6.4 User Interface (UI)	22
7 Implementering	26
7.1 Fundament	26
7.1.1 Unity	26
Komponenter & Scripts	26
7.1.2 Den Typiske Scene	26
7.2 Game Manager	28
GameManager.cs	29
DynamicDiffManager.cs	30
7.3 DDA Systemet	31
7.4 AI & Tilhørende	32
7.4.1 EnemyAI	32
7.4.2 Waypoints	33
7.4.3 Lys og AI'en	34

7.5	Spiller Karakteren	35
7.6	Grafik	36
7.6.1	Post Processing Stack	37
7.7	Level Design	38
7.7.1	Level Handler og Opgaver	38
8	Test	40
8.1	Problem test 1	41
8.2	Problem test 2	41
9	Konklusion	43
10	Refleksion	44

2 Indledning

Mange videospil benytter sig udelukkende af en lineær sværhedsgrad, hvor spilleren som regel har valget mellem nogle forskellige forbestemte grader: easy, normal, hard. Valget har til formål at gøre spillet tilgængeligt for en større gruppe af spillere, fra nybegyndere til veteraner. Det er en nem løsning, der sørger for at spilleren som regel er udfordret hvor spillet ikke er for nemt eller for svært, dog kan det ske at den valgte sværhedsgrad ikke passer perfekt til spilleren. Der er mange forskellige holdninger til, hvordan man bedst laver sværhedsgrad i videospil, hvor nogen mener at man ikke skal have et valg, og andre, at man skal have mange forskellige valgmuligheder. Hvorvidt det ene er bedre end det andet, kan kun bestemmes af spilleren selv, og hvad deres opfattelse af spillet er. Nogle spillere er stædige og foretrakker en udfordring, hvor de er villige til at starte forfra utallige gange, for at være den bedste og opnå de største resultater, så de vælger den sværeste sværhedsgrad. Andre, ville gerne opleve spillet, og har måske ikke tiden til at starte forfra hele tiden, og vælger derfor at spille på en lettere sværhedsgrad, for nemmere at kunne bevæge sig gennem spillet. Der findes dog en anden måde at beslutte sværhedsgrad, der kun tager spillerens input indirekte.

Dynamic Difficulty Adjustment(DDA) er en tilgang lavet til løbende og dynamisk, at ændre på sværhedsgraden af et givet spil. Som regel, fungerer det ved at tage stilling til forskellige variabler fra spilleren, såsom hvor tit de dør, og hvor godt de rammer eller ligende, hvor der så bliver udført justeringer til spillets mekanismer for at kompenserer for dette. Dette kan f.eks. være langsommere fjender, desto værre spilleren rammer.

Bemærk: De fleste fodnoter indeholder beskrivelser af forskellige ting gennem dokumentet. Disse beskrivelser er der som regel ingen kilder på, da de er baseret fra egen viden. De er der udelukkende for at hjælpe forståelsen af teksten, og skal ikke direkte ses som fakta. Derudover er ”baggrundsresearch” delen i dette dokument en del af analyse delen, da det giver mest mening i forhold til struktureringen af denne rapport.

3 Baggrund

3.1 Problemstilling

Her er problemstillingen fra projektoplægget givet til os:

”Udfordringen med at fastsætte sværhedsgraden i et spil er et tilbagevendende problem i alle færdighedsbaserede spil. En løsning på denne udfordring er at lade sværhedsgraden tilpasse sig spillerens evner eller hvor godt det indtil nu er gået spilleren. Dermed opstår der dog nye udfordringer: Giver det overhovedet mening at gøre sig umage når spillet så bliver sværere? Er det det samme spil vi spiller? Hvordan finder man ud af hvordan sværhedsgraden skal fastsættes? Hvordan måler man hvor godt spilleren klarer sig? Hvordan man ændrer sværhedsgraden er også et åbent spørgsmål. Er det noget spilleren er bevidst om og hvilken effekt kan det have? Man kan ændre på for eksempel AI'en for spillets modstandere, på level-designet, eller for spillets regler.”[5]

3.2 Problem afgrænsning

På grund af at vi har et bestemt tidsrum som dette projekt skal fuldføres i, er vi nød til at lave en projektafgrænsning der gør at vi har et realistisk syn på hvad vi kan nå på den tid vi har. Vi skal derfor finde ud af hvilke elementer der mindst skal til, for at vores produkt kan testes og spilles optimalt. Vi har derfor en ”Need to have”feature list og en ”Nice to have”feature list, der er opdelt i de elementer som vi har brug for i vores projekt, og de elementer som vi forhåbentligt kan få med.

4 Problemformulering

Mange har præferencer når det kommer til videospil, hvor man selv må bestemme hvordan man vil gennemføre spillet. Nogle foretrækker f.eks. at snige sig igennem en bane, mens andre hellere ville gå amok. På grund af dette hælder spilleren sig typisk til den ene retning fremfor den anden, og holder sig måske fra den ene spillestil.

I hvilken grad kan vi påvirke spilleren til at ændre spillestil, og hvilken effekt har dette på spillerens engagement?

5 Analyse

5.1 Gantt og Scrum

Vi har tænkt os at benytte en form for SCRUM planlægning til dette projekt. Dvs. Vi opdeler hele projektet op i forskellige *sprints*, hvor vi får lavet X antal opgaver. Alt implementering bliver delt op i disse opgaver, og de bliver tildelt forskellige sværhedsgrader, hvilket hjælper os til at danne et overblik over, hvor lang tid de individuelle opgaver tager at udføre. Alle opgaver er delopgaver, ud fra vores kravspecifikation, og det er i enden, den vi sigter efter. Derudover benytter vi os af et simpelt gantt diagram, for at danne et overblik over hele forløbet.

For at holde styr på vores SCRUM planlægning, har vi tænkt os at benytte online hjælpemidler, såsom Trello. Her kan vi nemt gå ind og sætte opgaverne op, på en nem og overskuelig maner.

Vores Gantt diagram og Scrum opsætning kan ses i bilaget.

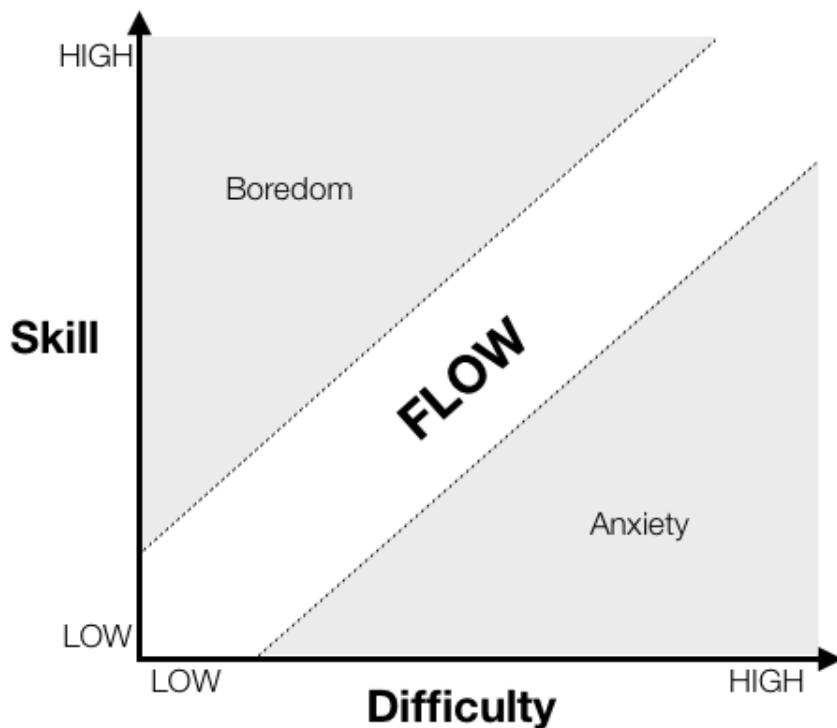
5.2 Dynamisk eller halv dynamisk levels

Fordi at vi i vores problem formulering skal finde ud af ”**I hvilken grad** kan vi påvirke spilleren til at ændre spillestil”, skal det gøres klart hvordan vi mäter graden. Det giver mening at antallet af opsatte levels stiger parallel med kvaliteten af test svarene, men på grund af vores begrænsede tid og ressourcer, har vi valgt at opstille 6 antal levels op, som er en betydelig mængde i forhold til at få svar på vores problemformulering. To metoder gav mest mening i forhold til vores spil: ”halv” dynamisk level og dynamisk level.

Den ”halv” dynamisk metode er at skabe færdiglavede levels, der giver muligheden for at opsætte alle scenarier op i et system. Med denne metode opstiller man parametre som mäter spillerens spillestil. Som et simpelt eksempel, kan man kigge på om spilleren har dræbt en modstander som **var** i ”alerted”(at fjenden har set spilleren) tilstand og som **ikke var** i alerted tilstand. Man tager så de tal fra de forskellige parametre, og finder ud af hvad spilleren har flest af. Det næste level er så bestemt ud fra de tal som spilleren har flest af. Dette vil sige at man kan finde ud af alle de forskellige ruter som hver spiller kan tage. Ved at kigge på antallet af levels som er med i vores spil, kan vi finde ud af hvor mange mulige ruter der er. For hvert antal af levels som spilleren kan spille, stiger antallet af mulige ruter med 2, som vist på det sidste bilag. I dette tilfælde er der 6 levels som spilleren skal igennem. Vi kan så ud fra denne metode finde ud af hvor stor en grad det var at vi fik påvirket spilleren på, ved at kigge på hvor tæt på de landede de opsatte ideelle ruter. Ved at give hver rute et tal fra 1-32, hvori de ideelle er 11 og 22, kan man tage alle testpersonernes slutrute tal, finde gennemsnittet, og finde ud af hvor effektivt vi har fået spilleren til at ændre spillestil.

Den anden metode er den dynamiske metode. Med denne metode tager man de samme parametre fra den forrige metode, men den her gang blive næste level skabt ud fra de tal som spilleren har givet spillet. På denne måde blive

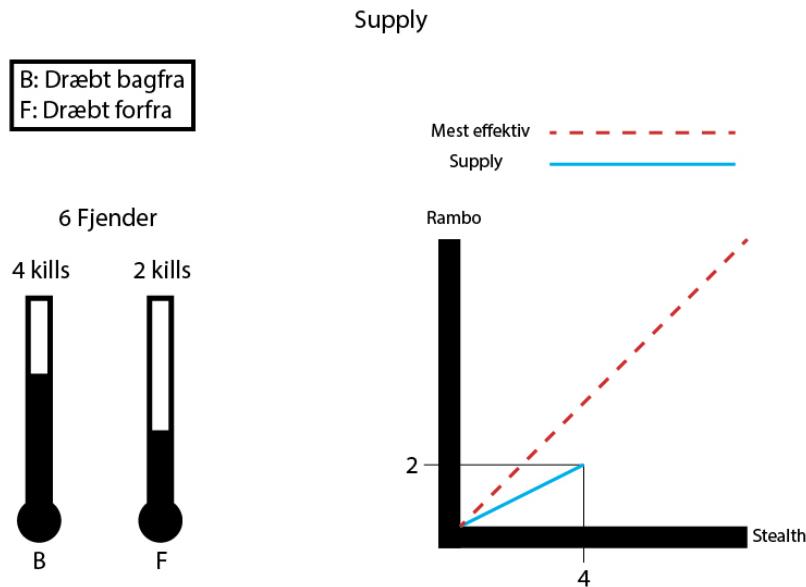
næste level skræddersyet til den spillestil som spilleren har spillet. Måden man så finder ud af hvor stor en grad vi kan få spilleren til at ændre spillestil med denne metode, er at tage parametrene fra de to spillestile, og sætte dem op i et difficulty flow chart. Man bruger et "supply and demand" system, hvilket der også bliver gjort brug af i "Dynamic difficulty adjustment through parameter manipulation for Space Shooter game" [4]. I dette paper bruger de et difficulty flow chart til at finde ud af hvor mange og hvilke ressourcer der skal uddeles til spilleren, baseret på hvor de ligger på et flow chart i det bestemte tidspunkt.



Figur 1: Difficulty flow chart, tilpasset fra Baron, 2012[1]

I dette eksempel skal vi igen bruge en parameter der mäter om spilleren vælger enten den ene spillestil eller den anden, derfor tager vi som eksempel parameteren der mäter om man dræber en fjende forfra og bagfra. De parametre som der nu er sat op som eksempel er spillerens supply. Figur 2 opstiller dette.

Det er så spillet der skal levere den demand som får spilleren til at holde sig indefor den "Mest Effektiv" linje. Spillet ved nu hvor mange og hvilke ressourcer der skal placeres i næste level, for at få spilleren til at prøve alle spillestile. I dette tilfælde kan det være ved at placere et skjold bag på 4 ud af de 6 fjender på næste level. Man kan så derefter holde øje med hvor effektiv metoden er, ved



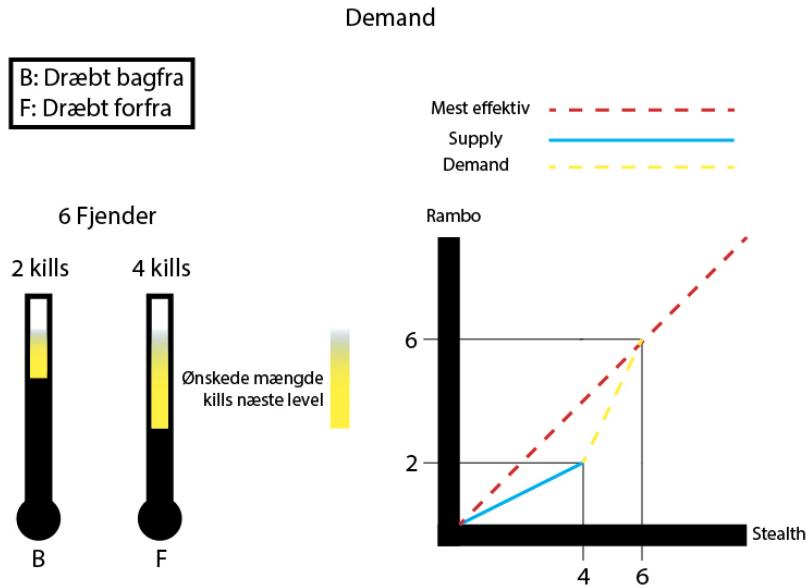
Figur 2: Model der viser spillers supply i forrige nævnte eksempel

at kigge på hvor tæt på spilleren ligger sig op til ”Mest Effektiv” linjen. Figur 3 opstiller dette

På denne måde får man en metode som er mere dynamisk end den første metode.

Baseret på den mængde tid og ressourcer tilgængeligt, valgte vi at skabe et system som er en blanding af de to forrige nævnte systemer. Dette system tager parameterne fra det dynamiske system, og tilpasser dem til de fjender som man kommer til at møde i det næste level. På samme tid med det, bliver der ud fra de parametre gennem et system, valgt 1 ud af de 3 baner indefor den bestemte spillestil som næste level er baseret på. Hver af disse baner bliver også tildelt et objective (mål), som passer med den spillestil som det næste level er baseret på. På denne måde bliver både baner og fjender påvirket af de opstillet parametre. Dette bliver opstillet i figur 4.

Metoden vi bruger til at finde ud af i hvilken grad spillet har været i stand til at ændre en spiller spillestil, er ved at tage antallet af gange som spilleren har brugt en af de to tilgængelige spillestile, sammenligne dem, og finde hvor meget spilleren har ændret sin spillestil i procent. Dette gør vi med et system som har et DDA system implementeret, og et uden. Til slut vil disse procenter blive trukket fra hinanden, for så at finde ud af hvor stor en ændring der har været, målt i procent.

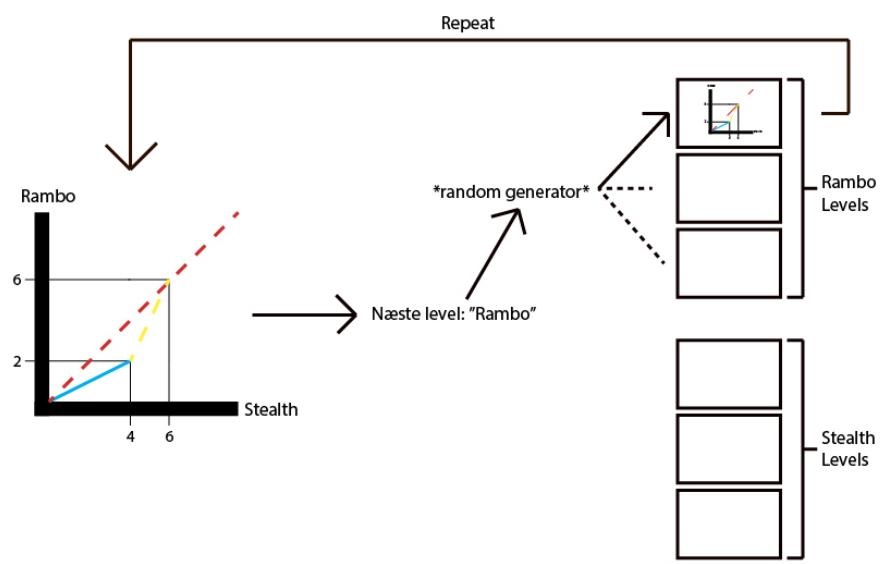


Figur 3: Model der viser spillers demand i forrige nævnte eksempel

5.3 Måling af engagement i computerspil

2. del af denne rapports problemformulering lyder således: "Hvilken effekt har dette på spillerens engagement?". Det kan være svært at måle en spillers engagement, da det er bredt og ikke har nogen konkret metode at måle det på. Derfor er der skabt forskellige metoder at måle engagement på, af forskellige videnskabelige artikler.

En af disse metoder kommer fra artiklen "Five Approaches to Measuring Engagement: Comparisons by Video Game Characteristics[3]". Som titlen siger, har de valgt at måle en spillers engagement, ved at måle på flere aspekter i en spiller. "survey self-report, content analyses of player videos, electro-dermal activity, mouse movements, and game click logs[3]". De kommer så med den konklusion at spillerens bevægelser (med musen, da dette var det eneste spilleren kunne interagere med), spillerens "presence" og spillerens endelige svar på deres "survey self-report" havde en korrelation da det kom til at måle spillerens engagement. Ved at vælge denne metode til at måle engagement på, skal man opstille noget der kan tage den data og sætte det i sammenligning. Dette kan man gøre ved at tage spillerens svar fra personens spørgeskema, og opsætte det i forhold til de data som vi modtaget gennem bevægelse og tilstedværelse. Denne data vil så blive opdelt i et point system, der vurderer hvor stor et engagement niveau spilleren har haft.



Figur 4: Model der viser det endelige system brugt i spillet

6 Teknisk

6.1 Kravspecifikation el. Feature List

Her er kravspecifikationen for projektet, også kaldet for en *feature list*.

6.1.1 Need To Have

- DDA System
 - Dette er kernen bag projektet. Dette system skal implementeres tæt sammen med den fjendtlige AI, da det er den primære ting DDA systemet skal påvirke. Det er planlagt at systemet skal kunne justere på de forskellige sværhedsgader i spillet, gennem en simpel talværdi. Denne talværdi skal være en forsimppling af de ændringer der sker gennem systemet, for at gøre det nemmest muligt at justere på værdien når der testes.
Systemet **skal** som minimum kunne ændre på AI'en.
- Fjende AI
 - Spillet skal indeholde en simpel fjende AI, som skal have nogle forskellige funktioner. Primært skal den være i stand til at blive justeret af DDA systemet, f.eks. Mere skade eller større synsvidde. Dette skal være en universal fjende, og der er ikke nødvendigvis brug for mere end en.
- Grafik
 - Simple grafiske elementer
 - * Spiller
 - * Fjender
 - * Level (væg, gulv)
 - * Projektiler
- UI
 - Knap der skifter mellem prædefineret system & DDA system
 - Reset knap
- Level Design
 - 1 Level
 - * Spillet skal som minimum kun indeholde et individuelt level. Der er ikke brug for flere forskellige levels for at teste fundamentet af projektet, dog er det en ting som ligger i planerne.

6.1.2 Nice To Have

- DDA System
 - Elementer fra et levels design, der bliver påvirket af spillerens spil-
lestil
 - * Placering af vægge
 - * Lys
 - * Health packs
 - * Traps
- Musik & Lyd
 - Adaptiv Musik (musik der skifter flydende i.flt. sværhedsgrad)
 - Lydeffekter (skridt, projektiler, død, nærkamp, UI)
- Grafik
 - Gennemgående grafik
 - * Spiller
 - * Fjender
 - * Level
 - * Projektiler
- UI
 - Gennemgående UI Tema
 - Main Menu
 - Pause Menu
- Level Design
 - Dynamisk levels eller prædefineret levels tildelt til de variabler ud fra
spillerens spillestil

6.2 Design

Mange tanker er gået til at skabe dette projekt, her vil de blive beskrevet. Det er også her, at en detaljeret beskrivelse af alle elementer af projektet kan findes. Der beskrives tankerne bag designet af elementer såvel som hvad projektet generelt indeholder. For at finde information omkring hvordan de forskellige elementer er lavet og fungerer, se sektion 7.

6.2.1 Gameplay

Gameplay har været et vigtigt element i dette projekt af en meget vigtig årsag: hvis gameplayet er uinteressant og el. kedeligt, bliver det svært finde svar på vores problemformulering. For at få svar på vores problemformulering, kræver det at vores testpersoner tager spillet seriøst, hvilket er svært med dårligt gameplay. Derudover er det kritisk at spillet er engagerende. Med det i minde, har gameplay været det største fokus, efter DDA systemet.

Spillet fungerer som et simpelt *top down shooter*, altså et spil hvor man styrer en karakter på en bane, med et perspektiv som var kameraet en drone og hvor man som regel skyder på ting. I vores tilfælde er spillet opbygget i forskellige levels, som man spiller igennem en efter en. Spillet er uændeligt, dog er der ikke uendelige baner. Der er designet **otte** baner, hvor man spiller gennem **seks** af dem gentagende. Et level fungerer som tutorial som man selv har valgmuligheden om at spille, et andet, er et intro level som man alting skal spille igennem når man starter et spil. De sidste seks er delt op i to kategorier: rambo og stealth, som forklaret i sektion 6.2.2.

De seks levels indeholder hvert især et ud af tre mulige opgaver som spilleren skal løse for at komme videre til næste level: **Assassination, Item & A to B**

Assassination går ud på at spilleren skal ind og eliminere en specifik fjende. Det er tilladt at dræbe andre fjender, dog låser udgangen til næste level ikke op, før denne fjende er elimineret.

Item giver spilleren opgave at hente en kasse, for så at tage den med ud af levelet igen. Her starter spilleren altså ved udgangen som er låst, og skal vende tilbage med denne kasse, for at låse den op. Når spilleren bæger på kassen, bliver deres hastighed reduceret. Der har været forskellige design ideer til denne opgave. Det var planlagt at kassen skulle udsende lyde hvert *x sekunder*. Dette blev dog ikke medtaget, som et resultat af tidsbesparelse.

A to B som navnet hentyder, er denne opgave simpel. Spilleren skal fra A til B. Her er spillerens eneste opgave bare at komme ud a levelet, hvor udgangen er låst op fra starten.

Det var planlagt at der skulle være ligeså mange opgaver som levels, for at skabe en større mulighed for DDA systemet at vælge et level OG en opgave. Dette blev dog ikke til noget for at spare tid. Ideen var at alle levels skulle være designet til at kunne passe til alle opgaver.

Stealth vs Rambo Som sagt så er en af de vigtigste elementer i spillet, gameplay kontrasten mellem de to typer: stealth og rambo. Spilleren har muligheden for at spille enten den ene eller den anden måde, eller begge på samme tid. Hele præmissen med problemformuleringen er at se om man kan ændre spillerens spillestil, derfor er det vigtigt at det er et frit valg for dem. Måden stealth og rambo defineres på er som følger.

Stealth er når spilleren sniger sig rundt, og ikke bliver opdaget af fjenderne. Når en spiller bruger stealth stilten, handler det om at komme igennem et level uden at affyre sit våben, men bruge kniven for ikke at larme, og tiltrække fjender.

Det er dog ikke ligetil at snige sig rundt, for når man bevæger sig laver man lyd, og man kan nemt blive set hvis man er i lyset.

Rambo er det stik modsatte af stealth. Her bruger spilleren sit gevær, og larmer. De fleste fjender finder hurtigt ud af hvor spilleren er, og jager dem til de er døde, eller forsvinder i skyggerne. Det interessante ved rambo spillestilen, er måden som skud virker på i spillet. Skud rammer nemlig ikke sit mål, samme sekund som de er affyret, det er langsomt. Dette gælder både for spiller og fjender. Dette giver mulighed for spilleren til at undvige fjendtlige skud, hvilket gør spilleren har en chance mod de mange fjender der kan fremkomme senere i spillet.

Et andet element for gameplay delen af spillet, er måden spillet slutter på. Idet at det er uændeligt, har der været nød til at indsætte noget til at afslutte spillet, for ikke at efterlade spilleren med en følelse af meningsløshed. Med dette blev et livsystem implementeret. Dette gør at spilleren har X antal liv, hvor de mister et hver gang de dør. Når spilleren har mistet alle sine liv, slutter spillet og de bliver smidt tilbage til hovedmenuen. Dette ledte også til at spillere var mere forsigtige, da der var en egentlig konsekvens for at dø.

6.2.2 Level Design

Level design i dette spil

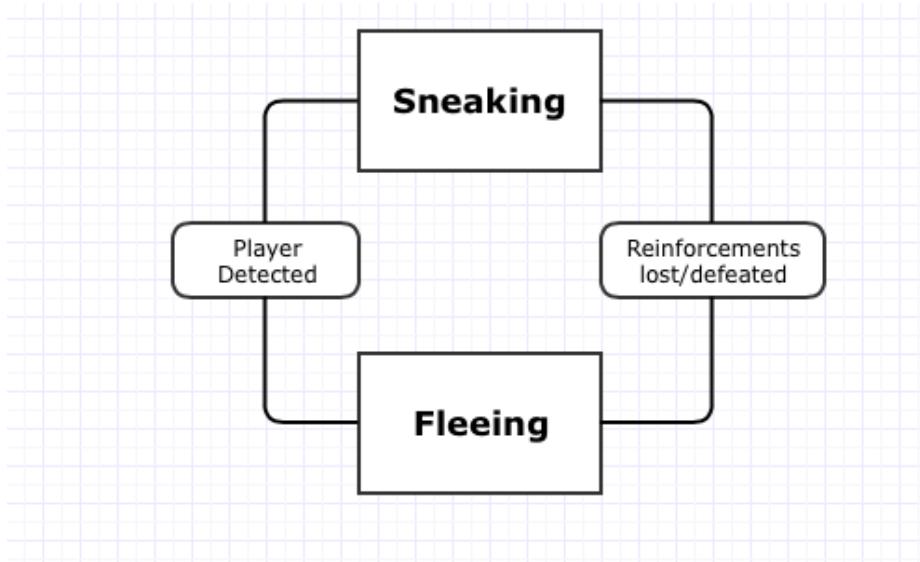
6.3 Level Design

Level design kan være et vigtigt punkt når man prøver at ændre en spillers spillestil. Level designet kan nemlig tælles med som en parameter i forhold til det forrige nævnte supply og demand system. Der findes nemlig bestemte måder at designe levels på til de forskellige spillestile, så derfor er der også bestemte måder at skabe level design til vores valgte spillestile: rambo og stealth.

Når man designer et level, skal man først kigge på hvilke værktøjer man har med at gøre. I dette tilfælde er det ting som væg, lys, AI position, AI rute og alle de små parametre som disse AI'er er blevet tildelt. Man skal så derefter kigge på hvad målet er med det level som man designer. I dette spil er der implementeret objectives, som er skabt for at hjælpe effektiviteten af spillerens ændring i spilletstil. Måden som disse objectives fungerer på, er at der på hver af de 6 levels, er et bestemt objective knyttet til det bestemte level. 3 af disse objectives er fokuseret på at ændre spillerens spillestil til rambo, og de andre 3 objectives er fokuseret på at ændre spillerens spillestil til stealth. Fordi at der i dette spil skal være mulighed for at kunne spille begge spillestile i alle levels, skal disse levels skabes ud fra principper og elementer fra både stealth level design og rambo level design.

Stealth level design Når man designer et stealth level, er der mange parametre der er i spil. Et af de vigtigste og helt basale, er de to tilstande som stealth spil generelt er baseret på. Dette kan ses på figur 5.

Der er i stealth spil 2 vigtige undersystemer som er med til at skabe et level:



Figur 5: Model der viser de to tilstade som mange stealth spil er baseret på, tilpasset fra Wong, 2014[6]

- **Grunts/ Enemy AI**

Den fjendtlige AI har en stor rolle i mange stealth spil. Sådan er det også i vores spil. Det der gør fjender i stealth spil specielt, er at de (i et godt designet stealth spil) er en del af selve det level man spiller. En vigtig ting med stealth levels, er at de skal være designet til at kunne **bevæge** sig igennem, i stedet for at **kæmpe** sig igennem, hvilket typisk er tilfældet for normalt level design i action spil. Dette princip gælder specielt når det kommer til fjender. Fjendernes AI i stealth spil, består typisk af 3 tilstade. **Idle, Searching og Alert.** Fjender i stealth spil har desuden også typisk en ”line of sight”. Hvis spilleren er indefor den kegle, går fjenden ind i alert tilstanden.

- **Player advantages**

Spillere har i mange stealth spil fordele, der gør det muligt for spilleren at kunne gennemføre et level på flere og mere kreative måder. Dette gælder dog ikke kun for stealth levels, player advantages finder man også i levels som er designet til andet end stealth, såsom rambo. Der er 3 typiske advantages som spillere har i stealth spil:

- **Mobility**

Mobility er ting som giver spilleren en fordel i forhold til bevægelse såsom at kunne bukke sig, dedikeret stealth mode, liste, kravle.

- **Hiding**

Hiding hænger sammen med mobility i den forstand at man via mo-

bility kan kan gemme sig. Man kan så buge dette til for eksempel at kunne kravle sig under små steder, hvor fjenden ikke kan se, eller kender til.

- Intelligence

Intelligence er bl.a. den generelle intelligens som spilleren har i forhold til de fjendtlige AI. Ting som viden om levels layout, fjenders placering og fjenders liv er også noget som er en del af intelligence.

Rambo level design Det kan være svært at definere ”rambo”spillestilen, så derfor er det også svært at skabe bestemte rammer indefor hvordan man laver et godt rambo level. Der er derfor heller ikke særligt mange artikler (hvis der overhovedet er nogen) om hvordan man level designer til denne spillestil. På grund af dette vil der i stedet tages et kig på hvordan andre spil af samme genre og spillestil, har designet deres levels, og derefter anvende det på vores levels.

Hotline Miami er et af de mest velkendte spil indefor top down shooter genren. Dette spil er bl.a. også kendt for dets hurtige tempo gameplay, og velfungerende level design, der opfordrer spilleren til at gennemføre banen med rambo spillestilen. En ting som gør dette spil anderledes i forhold til f.eks. et spil med fokus på stealth spillestilen, er dets fokus på rum til rum kamp. Næsten hvert level i Hotline Miami er opdelt i forskellige rum. Det vil altså sige at når man hopper ind i det ene rum og dræber sine fjender, er det op til spilleren hvornår han vil gå videre til næste rum. Dette er anderledes i forhold til stealth baseret spil, da der typisk er fokus på at man igennem et level bevæger sig flydende igennem, uden nogen bestemt system for hvornår man erude af skjul. Fjender i dette spil har også en øjeblikkelig reaktion til spillerens handlinger, hvor man i stealth baseret spil, ofte ser at det tager tid før fjenden opdager spilleren.

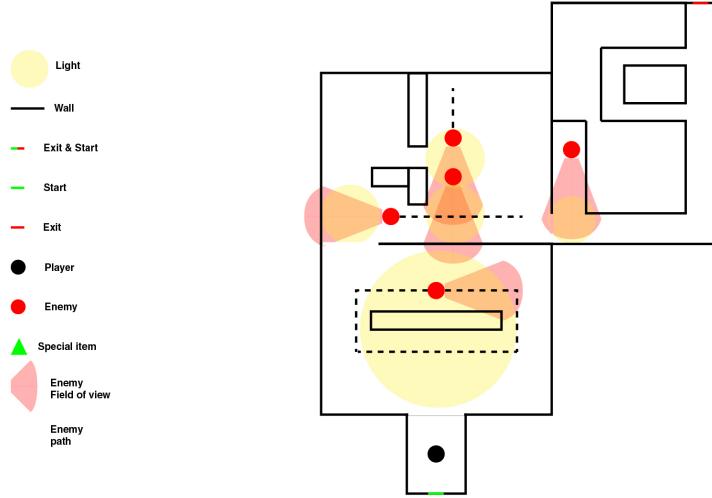
Alle disse ting er vigtige ting man skal have i sinde når man skal designe et level til rambo spillestilen.

Når man så er bevidst om hvilke værktøjer man har med at gøre, og man er klar på hvad ens mål med sit level er, kan man begynde at designe sit level.

I dette spil er der 8 levels. Der vil dog kun blive givet eksempler på 3 af dem, da det er de samme teknikker og metoder som bliver brugt igen på de andre levels. Der er i dette spil ikke meningen at **tvinge** spilleren til en bestemt spillestil, men at **opfordre** dem til en, så derfor skal hvert level give spilleren muligheden for at køre den spillestil de ønsker, uanset hvad spillet ønsker. Det kan derfor være svært at tage metoder og teknikker direkte fra det forrige afsnit om level design.

Det første level som kan ses på 6, er et level som spilleren kun går igennem en gang, i forhold til de andre levels, som spilleren sagtens kan gennemføre flere gange. Dette level er specifikt designet til at bestemme hvilken spillestil som spilleren hælder sig mest mod. Efter spilleren har gennemført dette level, er der nok information om hvordan næste level burde se ud som.

På 7 kan man se et ud af de tre rambo levels. I dette level er der kun en rute man kan tage for at nå sit mål. Vejen til målet er delt op i forskellige rum hvor der ikke er mere end 2 fjender i hvert. So nævnt skal detvære muligt



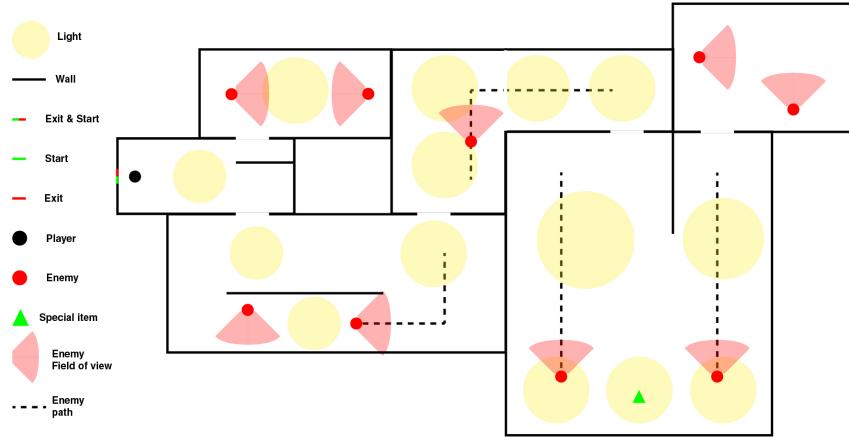
Figur 6: Det første level spilleren skal igennem efter tutorial level(2D kort)

for spilleren at spille den spillestil som spilleren ønsker, så derfor kan man ikke tage den nemme udvej, og tvinge spilleren ind i et rum hvor fjenderne kigger direkte på den dør der forventes at spilleren kommer igennem. I stedet er der i dette level med vilje sat et bestemt objective på, der opfordrer spilleren til at gå rambo. Ideen med dette level er at spilleren går igennem det meste af dette level uden at blive set. Når spilleren så når målet (den grønne trekant), begynder genstanden at udsende en lyd, der fungerer som en alarm. Alarmen har dog en bestemt radius, og afspiller kun lyd indefor et bestemt tidsrum, der igen er designet på den måde der gør det muligt for spilleren stadig at gå stealth på vej tilbage til udgangen.

På 8 kan man se et ud af de tre stealth levels. Dette level er ligesom det andet, baseret på researchet fra dets forrige afsnit. I dette level er der mere fokus på bevægelse fremfor ren kamp. Dette er der bl.a. gjort ved at gøre det enkelte level til 1 stort rum. Der er desuden tilsat flere fjender per kvadrat(afstanden i spillet) der yderligere opfordrer spilleren til at spille stealth spillestilen, fordi at alle fjender ville kunne høre hvis spilleren affyrede et skud, eller løb. Der er også sat flere forhindringer, og flere fjender med bestemte stier, der får spilleren til virkelig at tænke over hvilken vej spilleren skal tage. Dette levels objective er det simple ”Gå fra punkt A til punkt B”. Dette objective igen noget som kan spilles på begge måder, men som hælder mod stealth spillestilen, da man kan nå gennem banen uden at skulle dræbe alle fjender.

6.3.1 DDA System

Som der står i kravspecifikationen, så er dette kernen bag projektet og spillet. Dette er hvad problemformuleringen omhandler og fungerer som det primære



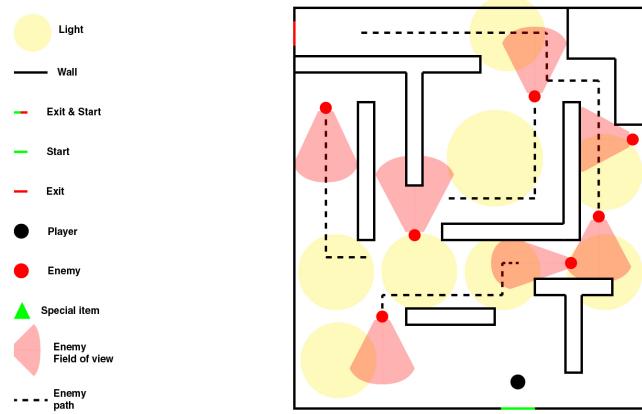
Figur 7: 1/3 rambo levels i spillet(2D kort)

fokus for projektet. DDA står for *Dynamic Difficulty Adjustment* hvilket på dansk, oversætter til Dynamisk Sværhedsgrads Justering. DDA systemet i dette spil har nogle simple funktioner, og laver nogle simple ændringer til spillerens forløb gennem spillet.

DDA systemet tager udgangspunkt i to primære ting som spilleren gør: hvor mange drab spilleren har fortaget sig og på hvilken måde. Der er to måder spilleren kan dræbe fjender på: kniv eller gevær. Hvis spilleren dræber en fjende med kniven, tæller det som et stealth drab, med gevær, et rambo drab. Efter hvert level ser DDA systemet så på, hvilken værdi der er højest, rambo- eller stealth drab. Hvis stealth er højest, får spilleren et *point* i stealth kategorien. Det samme gælder for rambo.

Når et nyt level starter, ser et separat system på hvilken kategori der er flest point i, og skaber et level baseret på dette. Hvordan disse systemer fungerer under hjelmen bliver gennemgået i detalje, under **implementering** sektionen.

Der bliver ikke benyttet mange forskellige variabler til DDA systemet, dog var dette planlagt, med variabler som hvor meget spilleren rammer ved siden af, hvor mange gange er spilleren blevet opdaget, hvilken hastighed gennemføre spilleren levels i, osv. Der har dog været et brug for at indskærpe alle disse variabler til de to mest simple, både for at ramme deadline, men også for at gøre evaluering og test af systemet nemmere. Havde der blevet brugt tid på at udvikle adskillige variabler til måling af spillerens type, ville det være en del mere tidskrævende at teste systemet, både for fejl, men også hvordan det påvirker en spiller.



Figur 8: 1/3 stealth levels i spillet(2D kort)

6.3.2 Fjende AI

Et top down shooter spil er ikke fuldendt uden noget at skyde på. Det er her AI fjenderne kommer ind i billedet. Dette er det første element af spillet som blev designet og implementeret, da det er kritisk både for gameplay og DDA system. Uden fjender ville der intet spil være.

Fjenderne er relative simple, og kan ikke mange ting. Dog fungerer de fint til formålet, og er lavet fra bunden under projektet. De har nogle fejl som desværre ikke er blevet fikset, men disse fejl præsentere ingen større problemer for spillet. Den største og mest irriterende fejl, er at de ikke tager tilbage til deres originale position, når de hører en lyd, eller jager spilleren, hvis de ikke tidligere havde nogen rute. Dette er dog ikke et problem for de patruljerende fjender. Mere detalje komme i implementerings afsnittet.

Der findes tre typer af AI fjender i spillet, som alle fungerer på ens måder, dog med nogle undtagelser. Opførelsen for alle typer af fjender er den samme, altså hvordan de reagere på ting, og hvordan de jager spilleren. Dog er der forskelle. De tre typer er: **Normal, Heavy & Fast**

Normal fjendetypen er den som de fleste fjender er. Den er standarden for hvad en fjende i spillet er, og der er intet specielt ved den.

Heavy denne fjendetyp er som navnet hentyder, stor og tung ift. de andre. Den er ca. dobbelt så stor i størrelsen som normal fjenden. Det der adskiller den ift. gameplay er at den har mere liv og skyder hurtigere. Dog er den langsom til at bevæge sig, og dens skud er langsmmere end andre for at balancere den.

Fast er den hurtigste af de tre fjendetyper. Den er halv så stor som normal fjenden, og er dobbelt så hurtig. Den bliver sjældent brugt i spillet da den er super stærk. Den skyder med en lidt langsmmere hastighed en normal fjenden,

dog er dens skud virkelig hurtige, og derfor næsten umulige for spilleren at undvige.

Det var først sent inde i spillet disse forskellige typer af fjender blev introduceret. Det var blevet bedømt efter adskillige test at en fjendetype ikke var tilstrækkeligt for at holde spillet interessant efter de første par levels.

6.3.3 Grafik

Siden de tidligste stadier af projektets idegenerering var det bevidst at grafikken var et element som der ikke ville blive sat særligt meget fokus på, medmindre der var ekstra tid. Spillet består derfor af meget simpel grafik. Dog er simpel ikke det samme som dårlig og spillet fremstår med en hvis visuel identitet, selvom der aldrig var en planlagt. Spillets endelige grafik er alt et resultat af løbende små ændringer, samt en markant opgradering mod de sidste versioner.

Der var dog noget som var bestemt siden starten: grafik stilen skulle være mørk og dyster, for at skabe god plads for at spilleren kunne snige sig. Generelt var ideen at skabe en kontrast mellem lys og mørke. Det endelige resultat kan ses på figur 9, og figur 10.

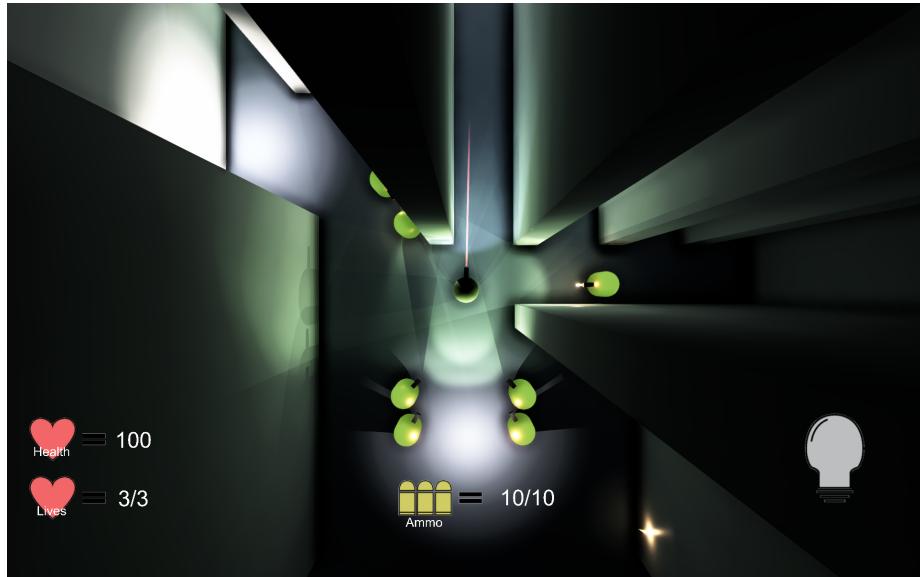
Som kan ses på figurenene, så består spillet af simple former og grafik, dog med en grad af realisme og detalje i lys og farver. Idet spillet er opbygget omkring det lyse og mørke, med mange lamper i smalle korridorer, begyndte fokus for grafikken at være at få lyset til at se godt ud. Det blev bedømt at den stil og stemning der var blevet opbygget, gjorde det til et ikke-problem at spillerens karakter og fjender bestod af pilleformer, der det gav en vis atmosfære til spillet. Der blev lagt laser lyde på fjendernes våben, så de lyder som robotter, for at komprimere deres underlige form.

Den største ændring i grafikken kom med brugen af *image effects*¹, og en ændring i højden på alle vægge i spillet. I tidlige versioner var det muligt at se over vægge (se figur 12,) hvilket blev bedømt som en uønsket mulighed, for at øge sværhedsgraden i spillet, da det var for nemt at se hvor alle fjender var. Dog bragte dette et problem ift. at kunne se hvor ens mål for levelet befandt sig, derfor blev der skabt et nyt redskab til spilleren: muligheden for at zoome ud, og se hele banen (se figur 11.)

6.4 User Interface (UI)

Dette er menuerne og information spilleren ser. Dette indeholder hovedmenuen og det som spilleren ser inde i selve spillet. Det har aldrig været en prioritet at få dette lavet, da det ikke tilføjer meget til selve spillet. Det blev dog vurderet at det var noget som godt kunne blive implementeret hen mod slutningen af produktionen, for at hjælpe med at kommunikere nogle forskellige elementer af

¹ **Image Effects** refererer her til en officiel assetpakke lavet af Unity. Den giver mulighed for at lave en masse ændringer på måden kameraets syn bliver renderet på, såsom bloom, depth of field, vignette og mere. For mere information kan pakken *Post Processing Stack* findes på linket: <https://assetstore.unity.com/packages/essentials/post-processing-stack-83912>

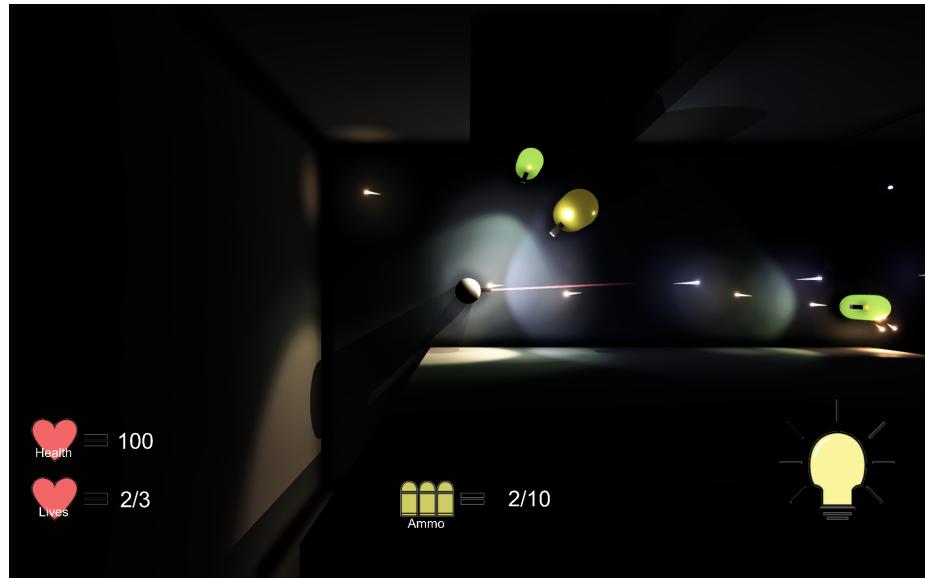


Figur 9: Billede af spillet i gang.

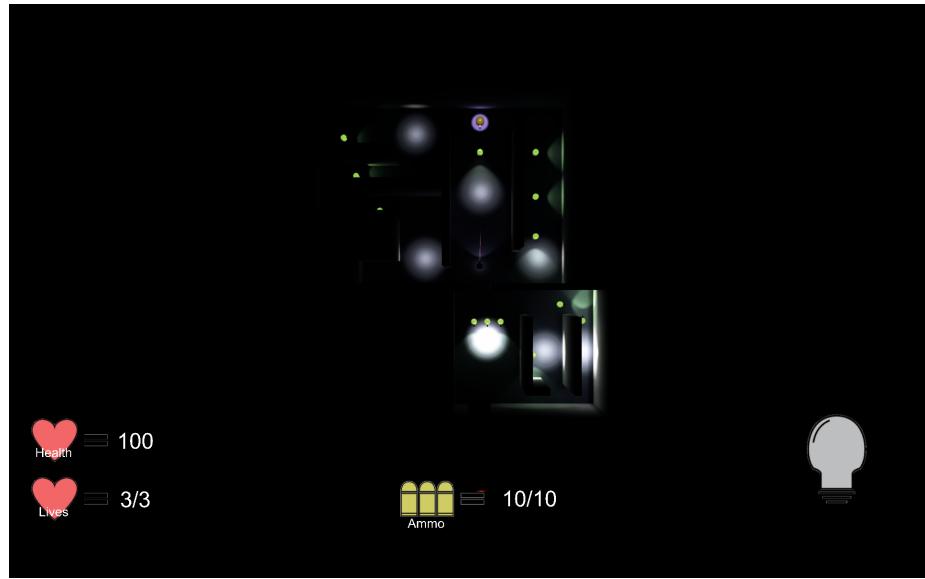
spillet, for at slippe fra frustrationer der var observeret under diverse spilteste gennem spillets levetid.

Præmært hjælper det spilleren til at få information om hvor meget liv, og ammunition de har tilbage. Det var frustrerende for testpersoner at kunne spille et level igennem uden at vide hvor mange skud de havde tilbage i geværet, samt hvor meget liv de havde tilbage, så de ikke vidste om de skulle passe på. Derudover har spillere løbende haft besvær med at vide hvornår de faktisk er i lys eller ej, pga måden det fungere på. Derfor er der implementeret et ikon der viser om man er belyst eller ej. Mere information om dette kan findes i segmentet 7. Der er også et element der viser hvor mange liv spilleren har tilbage.

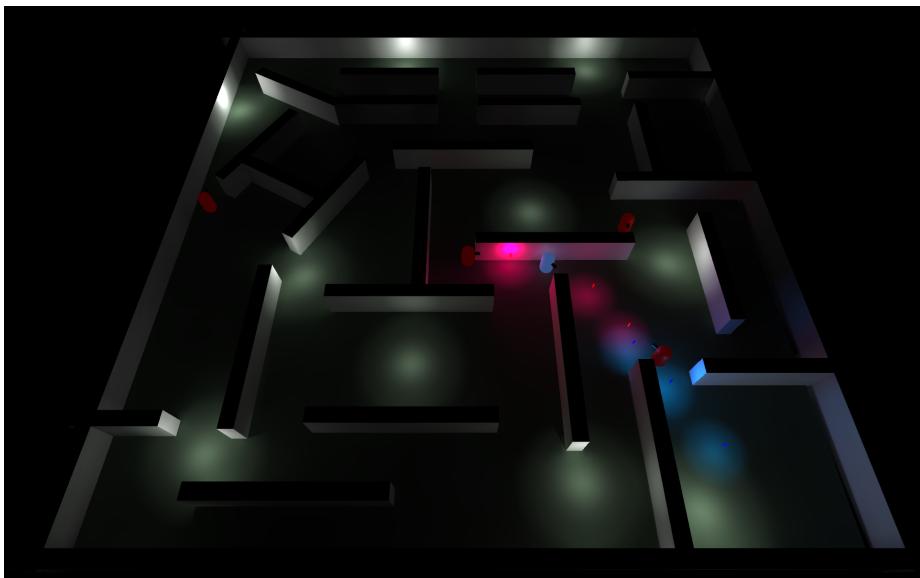
Det er et super simpelt UI, der nemt og hurtigt kommunikere hvad det har brug for. Se figur 9, 10 og 11 for eksempel på hvordan det ser ud.



Figur 10: Billede af spillet i gang.



Figur 11: Billede af hvordan det ser ud når man benytter sig af map zoom funktionen.



Figur 12: Billede af en tidlig version af spillet, bemærk kamera ikke følger spiller (blå pille.) Derudover ses spillet med en skrå vinkel, og alle vægge er lave.

7 Implementering

BEMÆRK: Det anbefales at 7.1 delen læses inden resten af sektionen. Der fremkomme en masse forklaringer på ting her, som ikke vil blive gentaget senere. Derudover skal det også noteres, at dette afsnit også fungere som dokumentation for spillet, hvilket betyder de fleste ting er beskrevet i meget detalje. Derfor kan visse ting skimtes skulle det blive for teknisk. Generelt er de fleste ting beskrevet enten i teksten eller gennem fodnoter. Dog kan det ikke garanteres at alle ting er beskrevet lige nøje ift. læsers forståelse for Unity og emnet.

7.1 Fundament

For bedst at kunne forklare implementeringen af diverse elementer af spillet, ville der her blive beskrevet den fundamentale måde spillet er opsat, og fungere på.

7.1.1 Unity

Unity er navnet på den game engine (oversat til dansk, spilmotor) som er blevet brugt til at producere spillet til dette projekt. Det er den game engine som vi har mest erfaring med, og er hvad vi har brugt igennem vores tid på gymnasiet. Når der senere bliver refereret til Unity, er dette hvad der menes.

Komponenter & Scripts Byggestenene for alle objekter i Unity, er dets komponenter og scripts. Uden komponenter på et objekt, er det fuldstændig tomt, og indeholder udelukkende transformations data (position, rotation og størrelse,) Når der nævnes komponenter er dette hvad der refereres til. Eksempler på komponenter kan være colliders, rigidbodies, text og scripts. Scripts er også et komponent i Unity. Et script er i dette tilfælde komponenter som er skrevet af udvikleren, for at øge funktionaliteten af objekter i Unity, som ikke allerede er understøttet.

7.1.2 Den Typiske Scene

De fleste scener² i spillet er opbygget af mange af de samme elementer, og der en struktur der skal følges når et nyt level bliver produceret. Her er en gennemgang af alle elementer i en scene (Se figur 13) Det skal noteres, at de fleste af disse ting ville blive beskrevet mere fyldestgørende løbende. Dette er for at danne et overblik og forståelse:

- *Game Manager*

Bemærk: Denne fremkommer ikke på figur 13, af den grund at den bliver hevet over fra tidligere scener. Dette element har sin egen sektion: 7.2, da det er centralt for spillet.

²Scene(r) er det samme som et level. Det er terminologien der bliver brugt til at beskrive et individuelt level el. niveau i Unity.



Figur 13: Et eksempel på hvordan en typisk opsætning af en scene ser ud i projektet. Der er dog nogle variationer ved forskellige scener.

Dette er rygraden for alle scener, og spillet generelt. Dette objekt står for mange funktioner der bliver brugt ved hvert level, og er generelt hvad der styrer alle elementer af spillet.

- **Level Handler**

Dette objekt er hvad der holder styr på level specifikke ting. For at bruge symbolisme for at eksemplificere; hvis game manageren er chefen af firmaet, er dette objekt manageren, der holder styr på alle arbejdere og ressourcer. Det sørger for at de specifikke ting i enkelte levels fungere som de skal, og spiller sammen med game manageren.

- **PlayerCamera**

Dette er kameraet spilleren ser igennem, og er ens over alle levels. Kameraet har to scripts på sig, der øger dets funktionalitet. Det ene, *CameraFollow.cs*, gør at kameraet følger spilleren når de bevæger sig. Det andet, *CameraMapZoom.cs*, tilføjer map zoom funktionaliteten, der tillad spilleren at se hele banen på en gang, ved at kameraet bevæger sig langt op over banen.

- **CamWorldPos**

Dette objekt fungere udelukkende som en placholder for hvor kameraet skal bevæge sig hen når brugeren benytter sig af map zoom funktionaliteten.

- **Player**

Denne er selvstændige og er selve spilleren. Mere info følger.

- **World**

Dette objekt holder på alle objekter der sammensætter den fysiske verden såsom vægge og gulv.

- **EndDoor**

Et andet vigtigt objekt for spillets mekanikker. Dette er den dør som bringer spilleren videre til næste level. Dørens tilstand bliver kontrolleret af *Level Handler* objektet.

- **Lights**

Et tomt objekt der holder på alle lys i scenen.

- **Patrol Waypoints**

Et tomt objekt der holder på alle AI waypoints i scenen. Disse waypoints fungere som de punkter der definere AI'ers patrulje ruter.

- **AIs**

Et tomt objekt der holder alle fjende AI'er i et level. Dette er en ud af to objekter der holder på AI'er. Forskellen mellem de to, er at denne holder på AI'er som **altid** er med i et level, hvorimod den anden holder på AI'er der kan variere baseret på information fra DDA systemet. Se næste punkt.

- **Variable Lights, Enemies**

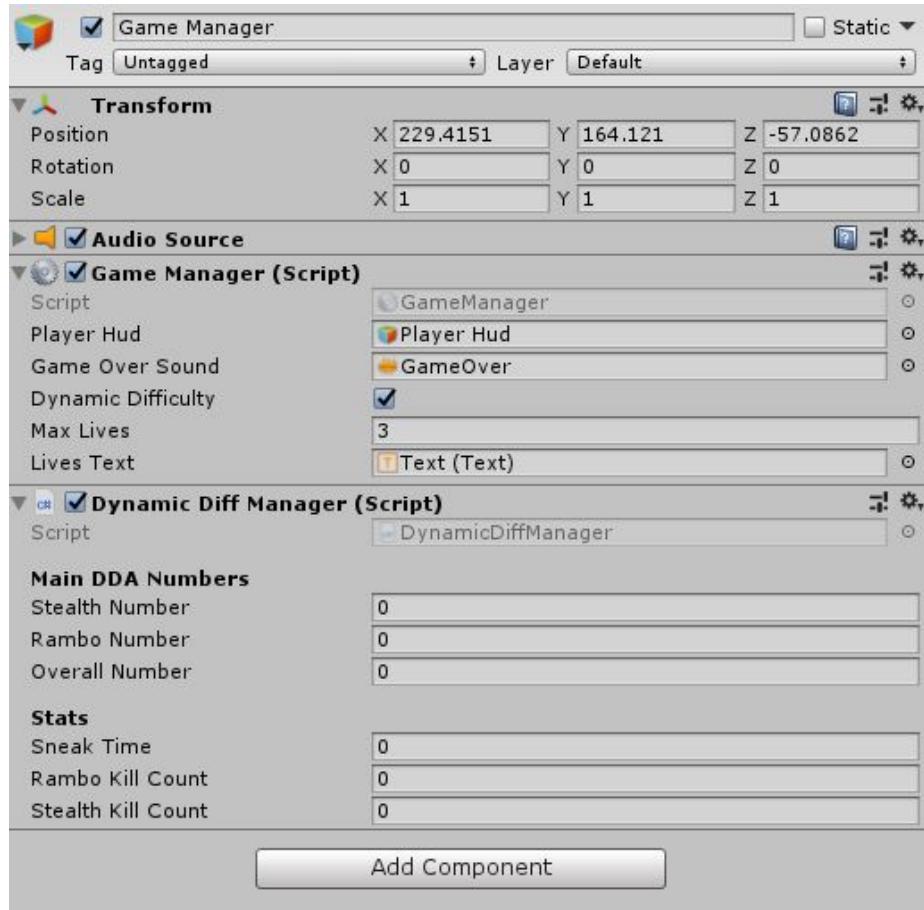
Begge disse objekter holder på henholdsvis, lys og AI'er. Disse er som udgangspunkt slæbt fra i et level. De bliver løbende aktiveret, baseret på information fra DDA systemet. Der er altså adskillige child-objekter på disse objekter³, som er slæbt fra som udgangspunkt. Dette er med til at ændre sværhedsgraden. Mere info følger.

7.2 Game Manager

Dette objekt fremkommer ikke over alle scener som kan ses på figur 13, ligesom level handleren f.eks. gør. Det er udelukkende tilstede under hovedmenu scenen. Dette skyldes at objektet er sat til at blive gemt hver gang der startes et nyt level. På denne måde følger objektet med hver gang spilleren starter et nyt level. Så selvom man ikke kan se objektet inden i hver scene i Unity, er det der når spillet kører.

³Et **Child-Objekt** er et objekt som er under et andet objekt. På figur 13 kan der ses at de flest objekter har en pil i venstre sidde. Dette betyder der er andre objekter under dem.

Objektet består af tre komponenter, to af dem er scripts: **Audio Source**, **Game Manager (Script)** og **Dynamic Diff Manager (Script.)** Audio Source er bare et komponent der tillader objektet at spille lyde. Se figur 14.



Figur 14: Et billede af alle komponenterne på Game Manager objektet.

GameManager.cs Dette script deler navn med objektet, og er kernen bag objektet. Det holder på størstedelen af objektets funktionalitet. Som nævnt, forbliver dette objekt over alle levels. Dette gør det ved at kalde *DontDestroyOnLoad()* funktionen i sin *Start()* funktion, hvilket er en Unity specifik funktion, fra *UnityEngine* namespacet.

De primære funktioner som dette script holder, uddover *Update()* og *Start()*⁴,

⁴**Update()** og **Start()** er funktioner som kan findes på mange scripts i Unity. Alt hvad der befinner sig i Start() bliver kaldt et frame efter scriptet er blevet aktiveret i scenen. Update() bliver kaldt hver frame scriptet eksistere, minus det første hvor Start() bliver kaldt.

er som følger:

- **OnLevelWasLoaded()**

Dette er en funktion fra Unity af, der bliver kaldt når en ny scene bliver indlæst. Den bruges i dette tilfælde til at sørge for at DDA systemet bliver sat i gang når scenen er indlæst. Dette skyldes måden DDA systemet fungere på.

- **LoadNewLevel()**

Denne funktion har til formål at indlæse et nyt level. Den indeholder en if-statement der checker om DDA systemet er aktiveret. Dette er fordi spillet både kan spilles med og uden DDA system, og det level som skal indlæses afhænger af dette. Hvis DDA er slæt til, indlæser funktionen et specifikt level baseret på DDA statistikker; hvis ikke, indlæser den et fuldstændigt tilfældigt level.

- **StartGame()**

Indlæser det første eller andet level i SceneManager⁵, baseret på om tutorial er aktiveret eller ej. Denne funktion bliver kaldt når man klikker på start knappen i hovedmenuen, og tager stilling til om tutorial knappen er slæt til eller fra.

- **RestartGame()**

Gør hvad man tror. Genstarter spillet.

- **GameOver()**

Bliver kaldt når spilleren dør. Navnet er en smule misvisende siden nyere revisioner af spillet, hvor et liv system blev implementeret, idet den ikke kun fungere som en funktion der slutter spillet. Når spilleren dør, spiller den en lyd der indikere man døde, også tjekker den om man har flere liv tilbage; hvis ja, indlæs nyt level og fjern et liv, hvis nej, kald *RestartGame()*. Originalt spillede den kun en lyd, ventede på den var færdig, også kaldte *RestartGame()*.

- **CloseGame()**

Lukker spillet. Bliver kun kaldt af *Exit* knappen i hovedmenuen.

Derudover indeholder det diverse variabler såsom *playerHud*, hvilket er det UI som spilleren ser i de diverse levels. Den indeholder også diverse booleans der tjekker om f.eks. tutorialen skal indlæses.

DynamicDiffManager.cs Hjertet af DDA systemet. Dette script ligger også på Game Manager objektet. Det indeholder fire primære funktioner der får DDA systemet til at kører:

⁵SceneManager er et system i Unity der holder styr på alle de scener, som er inkluderet i spillets build.

- **EngageDDA()**

Denne funktion har et formål, hvilket er at kalde de næste to funktioner der bliver beskrevet. Den eksistere for at spare plads når DDA systemet skal aktiveres. I stedet for at skulle kalde de individuelle funktion der får systemet til at virke, kan denne kaldes hvilket sørger for de andre bliver kaldt.

- **CalculateChangeValues()**

Udregner hvilke værdier der skal ændres efter et level, og fjerne de gamle værdier fra det level der lige er blevet spillet igennem. Helt basalt tjekker den om spilleren har flest eliminationer med rambo eller stealth spillestilen, og giver derefter et point til den spillestil der scorer højst. Derefter sletter den drabsdata, ved at kalde *ResetLvlStats()* funktionen, for at det nye level kan måles. Mere detalje kan findes i sektion 7.3.

- **MakeChanges()**

Tager værdier som lige er udregnet af den tidligere funktion, og benytter dem. Under sektion 7.1.2 blev objekterne *Variable Lights* og *Variable Enemies* beskrevet. Det er disse objekter denne funktion tager fat i. Den ser på den nuværende score spilleren har indenfor stealth og rambo, og aktivere lige så mange child-objekter under hhv. Variable Lights og Variable Enemies. For hvert point i rambo, bliver der aktiveret et objekt under Variable Enemies; for hvert point i stealth, bliver der aktiveret et objekt under Variable Lights. Dette betyder at jo højere score spilleren har i de individuelle spillestile, desto sværere bliver de.

- **ResetLvlStats()**

Dette er en simpel funktion som bliver kaldet af tidligere nævnte *CalculateChangeValues()* funktionen. Den tager to variabler og sætter dem til nul. Disse variabler er mængden af stealth og rambo eliminationer.

7.3 DDA Systemet

Denne sektion fungere som en opsummering af tidligere info, samt en mere konkret forklaringen af hvordan det hele fungere i et større sammenhæng.

DDA systemet i dette spil tager stilling til to primære ting for at vurde spillerens spillestil: hvor mange drab spilleren har fortaget sig, med enten stealth eller rambo spilletypen.⁶ Mere konkret, holder systemet styr på hvor mange fjender spilleren har dræbt med kniv og hvor mange med skydevåben. Disse bliver begge repræsenteret med simple variabler, *ramboKillCount* & *stealthKillCount* under *DynamicDiffManager* scriptet, som befinner sig på spillets game manager.

Efter hvert level, starter DDA systemet med at se på hvilken af de to variabler er størst, og giver derefter et point i enten variabelt *stealthNumber* el.

⁶For at genopfriske, så er spillets gameplay delt op i to hoveddele: Rambo- og Stealthstilen. Hvor fokus for spillet er at se om vi kan få spilleren til at ændre mellem disse spillestile, og ikke forholde sig til den ene eller anden.

ramboNumber, baseret på hvilken der er størst. Altså har spilleren dræbt flest fjender med stealth, får de et point i *stealthNumber*, og hvis de har dræbt flest med rambo, et point i *ramboNumber*. Disse værdier bliver gemt over hele spillet, til forskel fra dem der tæller antallet af drab. Med dette er det samlede antal fra *stealthNumber* og *ramboNumber* altid lig med det antal levels spilleren har været igennem, minus tutorial level, da det ikke medregnes i DDA systemet.

Når disse værdier er på plads, er det tid til at anvende dem. Dette sker ved at systemet tilføjer flere fjender eller lys til et level, baseret på hvor mange point spilleren har i de to spillestile. Helt konkret så er et point i *stealth* = et lys mere pr. level, og et point i *rambo* = en fjende mere pr. level. Så har spilleren en score som: **Rambo = 3, Stealth = 5**, ville den næste bane have tre ekstra fjender, og fem ekstra lys. Placeringen af disse fjender og lys er prædefiniret for hvert level, og der kan være et maks antal af otte ekstra af hver, på hvert level. Dette skyldes at der ikke er placeret flere pr. level. Idet at levels ikke er generet af et system, er det ikke muligt at overskride dette antal, da der simpelthen ikke findes flere lys eller fjender i et level.

Der var planlagt langt flere variabler til DDA systemet, dog blev de skåret fra senere i processen, da det blev vurderet det ville være for tidskrævende.

7.4 AI & Tilhørende

Det første der blev implementeret i spillet var AI'en, da spilleren ikke ville have noget at lave uden, og var derfor vigtigt at få implementeret så der kunne testes tidligst muligt. Det tilhørende der bliver nævnt i titlen, er alt hvad der bruges til at få AI til at virke, som waypoints og FOV cone (mere om dette senere.)

7.4.1 EnemyAI

Alle AI i spillet er baseret på *EnemyAI prefab*⁷. Dette prefab er altså kernen bag alle AI i spillet. Selve objektet indeholder tre child-objekter: **weapon**, **FOVCone** og **Spotlight**.

- **Weapon**

Weapon objektet er simpelt, og fungere som det våben Ai'en har. Det's eneste funktion er at være visuel, og at fungere som et sted hvorfra AI'ens skud kan instantiate⁸ fra.

- **FOVCone**

FOVCone objektet er en simpel model der fungere som AI'ens synsfelt.

⁷Prefab er betegnelsen på en sammensætning af objekter og komponenter, som kan genbruges og synkroniseres i Unity. F.eks. kan man lave en kasse med et script på der gør at den skifter farve når spilleren kigger på den. Her kan der være et variable der bestemmer hvilken farve kassen skal blive. Hvis man bare tager denne kasse, og kopier den rundt i hele scenen, også beslutter sig for det skal være en anden farve, skal man skifte denne farve på alle kasserne individuelt. Laver man derimod et prefab af kassen, kan man redigere alle instancer af kassen sammentidigt.

⁸Instantiate er når man skaber en kopi af et objekt. I dette tilfælde er det en kopi af objektet *AIBullet*, altså et skud.

Den fungere ved at den følger AI'en, og når spilleren er indenfor denne model, så tjekker AI'en om der er noget imellem spiller og AI ved hjælp af et raycast⁹. AI'en kan se spilleren hvis A) Spiller er indenfor FOVCone modellen (som fungere som en trigger¹⁰) og B) AI har en uforstyrret linje mellem sig selv og spiller (bruger raycast.)

- **Spotlight**

Et simpelt lys der nogenlunde viser spilleren hvor AI'en kigger hen.

Selve EnemyAI objektet indeholder tre vigtige komponenter: **Nav Mesh Agent**, **Ai Controller** og **Ai Manager**.

- **Nav Mesh Agent**

Dette er et komponent der kommer indbygget i Unity, og er kernen for AI'ens bevægelse. I Unity er der et indbygget navigations system for AI agenter, hvilket er baseret på et *Navigation Mesh*, som helt basalt er en måde for AI'er at vidde hvor de kan, og ikke kan gå henne i verdenen. Dette er altså ikke et hjemmelavet komponent.

- **AIController.cs**

selve hjernen for AI'en, og indeholder alt der definere den. Denne, og de tre andre komponenter er ens over alle AI typer i spillet, hvor den eneste forskel er værdien på de variabler dette script indeholder. På figur 15 kan der ses alle variabler som scriptet indeholder. Der er stor fleksibilitet, hvilket gjorde det nemt at skabe forskellige fjendtyper. Dette script har to funktioner der styrene AI tilstande, nemlig jag spiller, eller patruljer: *GoToNextWaypoint()* får AI'en til at tage hen til næste waypoint. (Læs mere om dette under sektion 7.4.2.), *ChaseAndShoot()* får AI'en til at jage spilleren, og skyde dem når de er tæt nok på, og har direkte syn til spilleren.

- **AIManager.cs**

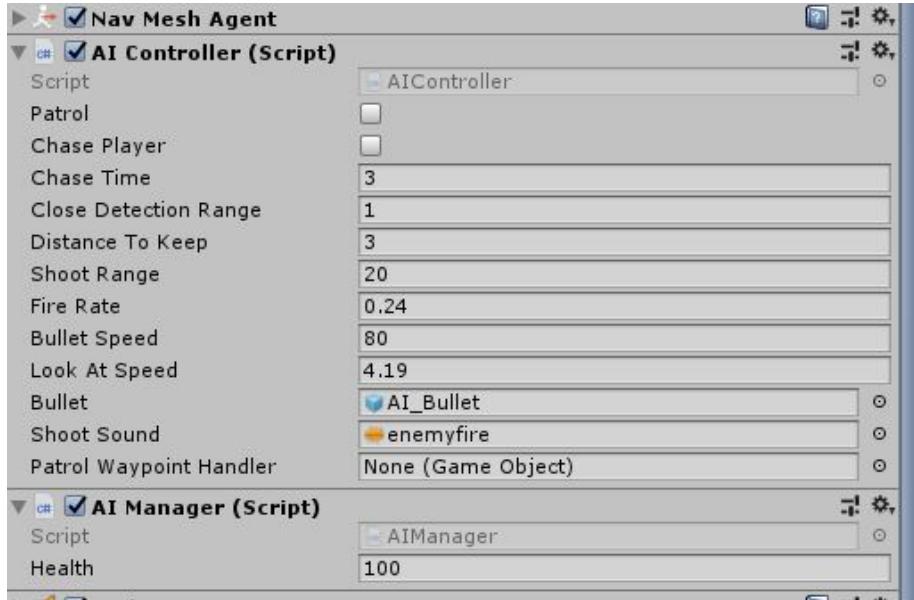
Styrer AI'ens liv, og tjekker om den skal dø eller ej.

7.4.2 Waypoints

Waypoints fungere som de punkter AI'erne skal følge når de patruljere. Når en AI er sat til at patruljere, skal der angives et objekt i variabelt *Patrol Waypoint Holder* under AI'ens controller. Under det objekt man giver, skal der være en række af selve waypoint objektet. Disse waypoint objekter indeholder et script kaldet *AIWaypoint.cs* som egentlig bare gør at waypoint objektets model bliver

⁹**Raycast** er en funktion i Unity, der ved hjælp af fysikenginen udsender en stråle(ray) som så kan kollidere med objekter der har en collider. Se eventuelt Unity's dokumentation for en bedre beskrivelse.

¹⁰En **Trigger** er en collider som fungere som en *aftrakker*, den kan retunere true eller false, baseret på om en anden collider har passeret den. Det kan findes en bedre beskrivelse under Unity's dokumentation.

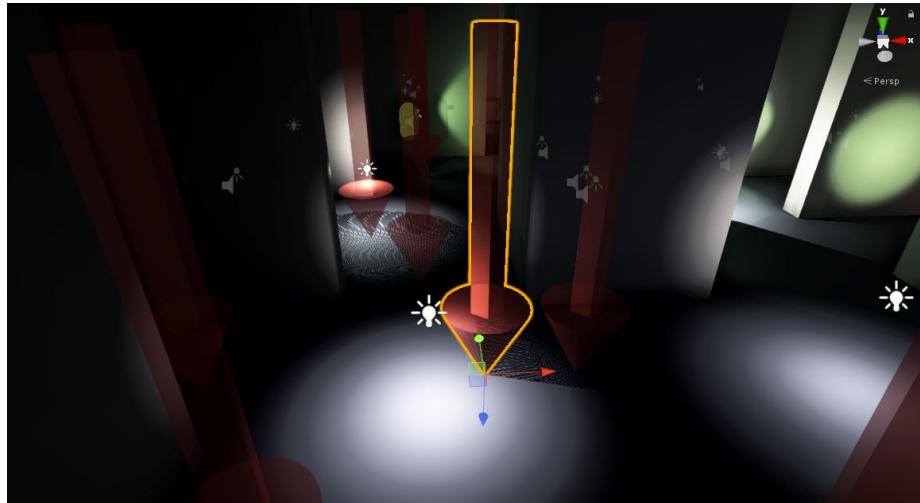


Figur 15: De tre komponenter der udgør kernen af AI'erne.

skjult, når spillet startes. Dette er fordi at når et level produceres i Unity, er Waypoint objekterne vist med en model af en stor pil, for at gøre det nemmere for udvikleren at skabe niveauet. Se figur 16.

7.4.3 Lys og AI'en

Det sidste der har en indflydelse på AI'en, ud over spilleren, er lysene i levels. Måde disse lys fungere på, er at de har et script på sig kaldet *LightDetection.cs*. Dette script tjekker om spilleren er inde for lysets rækkevidde. Dette gør det at lave en raycast imod spillerens retning, med startpunkt fra sit eget centrum. Dette raycast har samme længde, som lysets radius, altså raycasten rækker ligeså langt som lyset skinner. I starten lavede lyset dette tjek ved at skabe en *SphereCollider* omkring sig, med samme radius som lysets styrke, også tjekkede om spilleren var indenfor denne collider. Dette præsenterede dog problemet med at lyset ville registrere spilleren, selvom der var en væg mellem dem. Derfor blev raycasten introduceret, og det blev vurderet at raycasten kunne bruges alene, uden brug af collider. Dette system er dog ikke uden fejl, og kunne godt bruge forbedring. Lige nu så er spilleren enten synlig eller ej. Der er ikke noget imellem. Dette kombinere ikke godt med at AI ikke har en udskudt reaktionstid, hvilket betyder at hvis en spiller snitter kanten af et lys' radius, reagere AI'er med det samme på spillerens tilstedeværelse, hvilket fremkommer unfair over spilleren. Det nuværende system har også den begrænsning at det udelukkende fungerer på *Point Lights* og ikke andre typer lys, da selve lyset egentlig ikke er hvad der



Figur 16: De røde pile repræsentere waypoints.

tæller, men kun radius. Andre lystyper benytter sig ikke af radius.

7.5 Spiller Karakteren

Uden karakter objektet er der ikke noget spil. Spilleren har forskellige gameplay mekanikker, og består af følgende vigtige komponenter: **Player Manager**, **Player Controller** og **Player Footstep Controller**

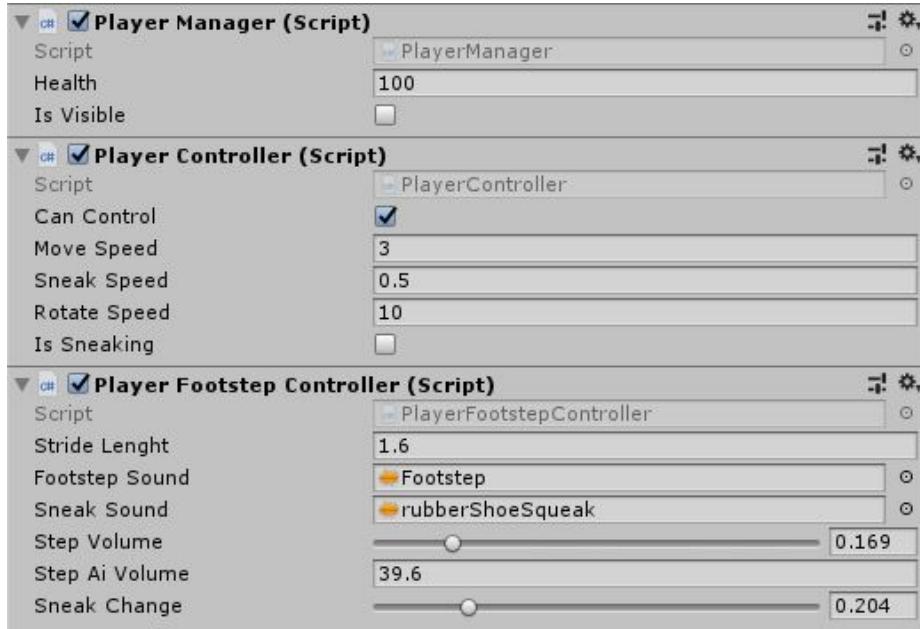
- **PlayerManager.cs**

Ligesom AIManager scriptet, har det til funktion at holde styr på spillerens liv, og at trigger en eventuel død. Derudover er dette komponent også hvad der holder styr på om spilleren er synlig, eller ej.

- **PlayerController.cs**

Dette er selve kernen bag karakteren, og er hvad der giver kontrol til spilleren. Som kan ses på figur 17, så er der diverse variabler der kan kontrolleres for at justere på måden karakteren fungere på. Det skal noteres, at dette script indeholder det eneste stykke kode som ikke er lavet under projektet af udviklerne, men er taget fra Unity3D wiki, hvilket er *MouseLook()* funktionen. Den gør det muligt for spilleren at kontrollere karakterens rotation ved at pege steder med musen. Det er dokumenteret i selve scriptet, hvor koden kommer fra, med link til ressourcen.

- **PlayerFootstepController.cs** Alle spillerens fodspor kommer fra dette script, som navnet hentyder. Der kan igen ses på figur 17, hvilke muligheder der er for justeringer på dette script. Dette script tog lidt tid at udvikle idet det indeholdte et problem. Først var ideen bare at afspille en lyd når spilleren bevægede sig. Dette er dog ikke så ligefremt som man ville tro,

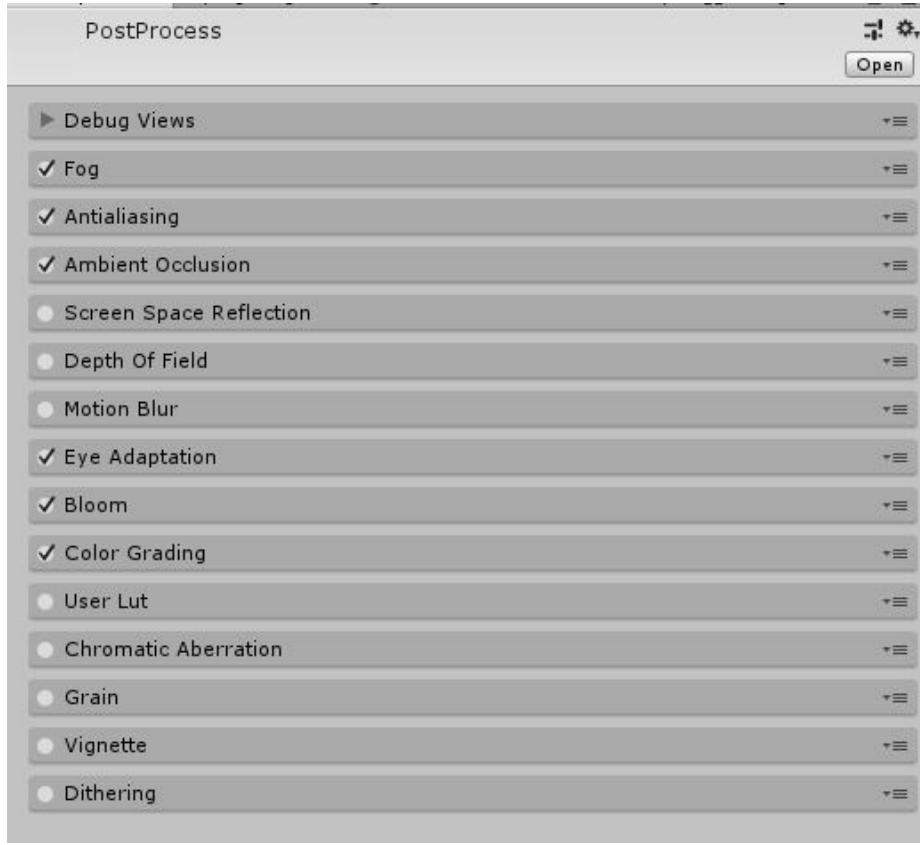


Figur 17: De vigtige komponenter på spiller karakteren.

da spilleren selv styrere hvor *langt* de bevæger sig, og hvor **meget**. Hvis man udelukkende spiller en lyd for hvergang der er bevægelse, lyder det underligt og unaturligt, og det passer ikke rigtigt til spillerens bevægelse. Dette blev løst ved at udregne distance mellem hvert *trin*. Måden det virker på, er at når spilleren bliver spawnet ind i verdenen, så registreres der et trin. Trinnet er defineret som en tredimensional vektorer, der holder styr på hvor det blev lavet i verdenen. Så mäter den på hvor lang væk spilleren har bevæget sig fra dette første skridt, og hvis spilleren er længere væk end en defineret længde, så laver den et nyt trin, og sådan fortsætter det uendeligt. Dette kan nemmest forstå ved at prøve at bevæge sig i spillet, små distancer af gangen og rundt i cirkler. Det gør det desuden muligt for spilleren at bevæge sig en smule uden at blive hørt af fjender, ligesom man i virkeligheden kan læne sig rundt om hjørner uden at tage et nyt skridt.

7.6 Grafik

Der er ikke meget at beskrive omkring implementeringen af grafikken, da den er ret simpel. Alle former og farver er noget som findes direkte inden i Unity, og der er ikke benyttet nogen form for importeret 3D modeller. Der er simpelt, effektivt og hurtigt. Lige som planlagt.



Figur 18: Vores Post Processing Profile, og de værdier vi benytter.

Det interessante at se på her, er brugen af Unity's egen asset pakke, ved navn *Post Processing Stack*. Denne pakke åbner op for en god del muligheder indenfor brugen af image effects på alle kamera i Unity.

7.6.1 Post Processing Stack

Vi benytter os af Unity's officielle asset pakke, *Post Processing Stack* til at skabe et bedre slut billede fra kameraet i spillet. Med denne pakke kan der implementeres post processing på et kamera, hvilket gør det muligt at skabe effekter som Depth of Field og Vignette.

Måden det virker på, er at man på det ønskede kamera sætter et komponent på med navn *Post Processing Behaviour* hvilket tager et enkelt input, i form af en *Post Processing Profile*. Denne profil er en som man laver i sin projekt mappe, og ser ud som på figur 18.

7.7 Level Design

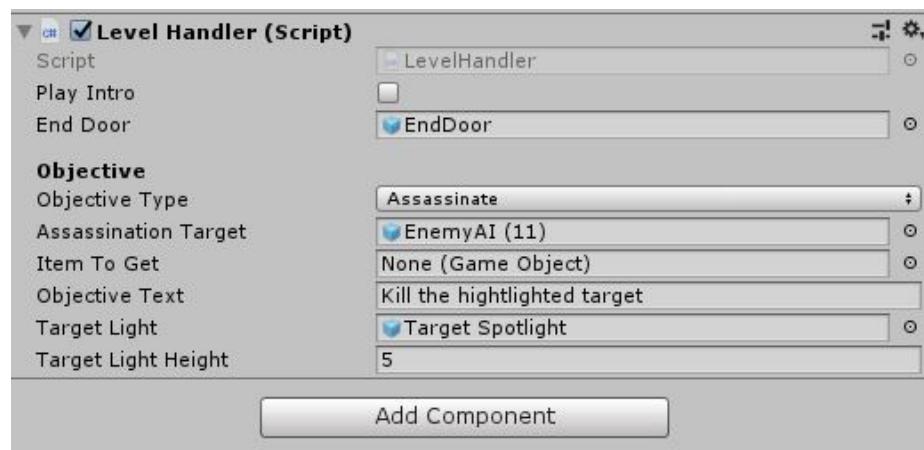
Alle levels er bygget ud fra (indsæt afsnit om waypoints, ai, levelhandler, lights, end door osv.). Da alle prefabs var oprettet, kunne man have alt hvad man havde brug for direkte på scenen i Unity. Det er ud fra tegningerne som 6, 7 og 8, at disse levels blev skabt. Tegningerne blev sat side om side med programmet, hvori der så først blev sat gulv og vægge på plads. Gulv og væggene bliver er skabt af 3d objekterne "Plane" og "Cube" i Unity. Man kan med disse planes og cubes bestemme størrelsen og placeringen, ved at ændre på "Scale", "Position" og "Rotation", som hver har en Z, Y og Z akse. Både cube og plane har en "Collider" komponent i sig, der forhindrer, i dette tilfælde, spilleren for at falde gennem jorden, og bevæge sig igennem vægge.

Efterfulgt af placeringen af vægge og gulv, er placeringen af lys. Lysenes placering er igen baseret ud fra tegningerne fra kapitlet 6.2.2. De implementerede lys, består af "Point lights" som er en af de lyskilder som Unity har til rådighed. Disse pointlights bliver brugt i dette tilfælde, da dette lys "Light" komponent består af en kugle. Denne kugle bliver så matchet med størrelsen af en "Sphere collider" som så også er sat på lyset som et komponent. Fordi at der varieres i lysets størrelse fra lys til lys, ændres der i lysets "Scale" under "Transform" komponentet i Unity, så at det kan passe med det ønskede størrelse.

Spilleren og fjenderne bliver så placeret på scenen. Da "Player" ikke skal ændre sig fra level til level, skal denne bare sættes ind i banen med minimal ændring. Denne ændring består af "Player"s rotation på Y aksen, da der kan varieres i hvilken retning som spilleren skal mod. Når den fjendtlige AI skal placeres skal man sørge for at dets script enten er sat til at være statisk, eller i bevægelse. Dette er sat op via en variabel som kaldes "Patrol" i scriptet "AI Controller". Hvis AI'en skal patruljere rundt, skal denne AI have "waypoints", i form af et "gameobject". Dette object indeholder så et bestemt antal waypoints bestående af et script "AI_Waypoint" som sørger for at AI'en følger den ønskede rute. Disse waypoints placerer man i rækkefølge fra top til bund, ud fra den rækkefølge man vil have at AI'en skal bevæge sig i.

7.7.1 Level Handler og Opgaver

Hvert level har, som nævnt under sektion 7.1.2, et *Level Handler* objekt, med et tilsvarende script, *LevelHandler.cs*. Hver gang der startes et nyt level, er det dette objekt der holder styr på hvad der sker undervejs, og om spilleren har opfyldt kriterierne for levelets opgave, for så at låse op for udgangen til næste level. På figur 19 kan der ses dette script i aktion, under et level. Det har en masse variabler som kan kontrolleres. Det skal dog nævnes at nogle af disse variabler ikke er i brug, da deres implementering blev stoppet halvejs da de blev skåret fra som et resultat af tidsbesparing. Dette var hovedsageligt et *intro* system til hvert level. Ideen var at når et level startede, ville der spille en lille introduktion af levelet, med en præsentation af ens opgave. Når et level bliver produceret, er det igennem dette objekt man definere opgaven som skal indgå i det level.



Figur 19: LevelHandler scriptet på Level Handler objektet.

8 Test

Dette projekt bringer en del udfordringer, og kalder på en ordentlig forumdørsøgelse, løbende research og test.

Vores primære udfordring er selve implementeringen af et DDA system. Der er mange dele der skal fungere sammen. Dette er grunden til vi vælger at afgrænse selve spillet som udgangspunkt, og det kommer ikke til at indholde utrolige mængder af gameplay. Det kommer altså til at være simple levels, dog skal der stadig være nok, til at de forskellige spillemåder er distinkte fra hinanden.

Løbene skal vi sørge for at teste spillet så meget som muligt, for at kunne se om vi nærmer os målet om at skabe en oplevelse der får spilleren til at prøve alt spillet har at byde på. Det bliver en udfordring at udføre disse test, da vi kommer til at skulle bruge en god håndfuld af mennesker, og det kan være vi skal bruge nye mennesker for hver test. Derudover skal vi helst udvikle forskellige versioner af spillet, for at opsætte forskellige test grupper op ad hinanden, og se hvad der virker bedst. Vi skal igennem disse test primært se på hvad den bedste tilgangsmåde er omkring spillerens viden af DDA systemet. Primært skal det testes, om det er bedst at spilleren ved det er der, eller ej.

Den største udfordring bliver at finde ud af hvad der virker bedst, og den bedste måde at implementere et DDA system på.

Vi har 3 forskellige test typer. Den første fungerer som kontrol og bug testing, hvor de to andre skal fungere som svar på vores problemformulering. Alle vores tests bliver testet af gamere, da det passer med den målgruppe som spillet er rettet mod.

Den første type test, er en standard bug test der sørger for at spillet virker som det skal, i forhold til ”bugs og glitches” (computerfejl). Denne type test kører vi i slutningen af vores første 3 sprints (markeret med lyseblå i tidsplanen), og dem skal der i følge planen ikke flere af efter de 3 sprints. Denne test kommer til at blive baseret på et allerede ønsket mål. Altså at konklusionen af denne test type allerede er kendt og på grund af dette, bliver denne test sat op ved at testpersonen skal spille spillet, og løbende fortælle om de problemer, bugs og glitches som personen oplever, i mens der bliver observeret og noteret.

Den anden type af test er den test vi i vores tidsplan kalder ”Problem 1 test” cog ”Problem 2 test”. Denne type test skal svare på vores problemformulering som er ”I hvilken grad kan vi påvirke spilleren til at ændre spillestil, og hvilken effekt har dette på spillerens engagement?”. I modsætning til den første test type, er der ikke allerede opsat et svar til testen. På grund af dette må man først kigge på det som vi gerne have svar på fra denne test.

Problem test 1 og 2 blev gjort på samme tid, selvom at det i følge tidsplanen skulle havde blevet gjort hvert for sig, med en uges mellemrum. Vi kom frem til den konklusion at vi sagtens kunne få svar på begge dele af vores problemformulering, på en gang. Testpersonerne skulle en af gangen op i et værelse, hvor de individuelt blev testet. Testpersonerne skulle hver igennem et tutorial level, hvor de derefter fik 3 forsøg på spillet.

8.1 Problem test 1

Vi målte hvilken spillestil hver person havde hvert level, ved at en af os stod bag testpersonen, og noterede hvilken af de 2 spillestile, som personen gik igennem. Dette blev gjort ved at kigge på metoden personen spillede på, og det næste level testpersonen endte i. Følgende tal er bestemt ud fra de antal levels som hver testperson har spillet en bestemt spillestil. Vi kom frem til disse tal: **MED DDA**: 4 Stealth 9 Rambo og 5 stealth 6 Rambo **UDEN DDA**: 7 Rambo 3 Stealth og 3 Stealth 8 Rambo. Vi fik i alt testede 4 personer Fra disse tal tog vi dem og dividere dem i hindanden for at få en procentdel, af hvor stor en forskel der var på testpersonernes ændring i spillestil, hvori at jo tættere på 100% der er, desto større en grad har spillet fået personen til at ændre spilletil. **MED DDA**: 44% og 83% **UDEN DDA**: 42% og 37,5%. Ved så at tage gennemsnittet af begge tal og trække dem fra hindanen, har vi fundet frem til hvor stor en grad vi har været i stand til at ændre på en spillers spillestil.

$$44 + 83 = 127,$$

$$127 / 2 = 63,5,$$

$$42 + 37,5 = 79,5,$$

$$79,5 / 2 = 39,75,$$

$$63,5 - 39,75 = 23,75$$

8.2 Problem test 2

Med problem test 2 brugte vi artiklen ”Five Approaches to Measuring Engagement: Comparisons by Video Game Characteristics”[3] til at vurdere testpersonernes engagement niveau. Vi valgte at bruge 1 ud af de 5 metoder til vores test, som var: ”Survey self-reported engagement”. Til ”Survey self-reported engagement”brugte vi spørgeskemaet ”The Game Experience/ Engagement Questionnaire”[2]. Dette spørgeskema er opsat i en struktur der deler hvert spørgsmål op i en genre, hvor man så kan tildele point til den genre baseret på testpersonens svar. Disse genre består af: **1.** Competence **2.** Sensory and Imaginative Immersion **3.** Flow **4.** Tension/annoyance **5.** Challenge **6.** Negative affect **7.** Positive affect. Fra dette spørgeskema valgte vi kun at inkludere ”Core Module”spørgsmålene, da spørgsmål som ”Social Presence Module” ikke giver mening i forhold til dette spil, samt kun inkludere genre fra ”Core Module” som gave mening i forhold til spillet. Fra disse tal tog vi de positive point (Competence, Flow, Challenge og Positive affect) og trak de negative point (Tension/annoyance, Negative affect) fra. Altså en bedømmelses skala fra **-10** til **94**. Her er de scorer som testene MED DDA fik, og UDEN DDA fik. Dette er dog sat op med 10 point, for at gøre skalaen mere relaterbar, så fra **0** til **104**.

MED DDA:

1. Competence: 23

2. Flow: 18

3. Tension/annoyance: 11

4. Challenge: 21

5. Negative affect: 24

6. Positive affect: 15

Endelig score: 52

UDEN DDA:

1. Competence: 26

2. Flow: 25

3. Tension/annoyance: 10

4. Challenge: 22

5. Negative affect: 12

6. Positive affect: 31

Endelig score: 92

9 Konklusion

Igennem en redegørelse, analyse, design, implementering og test, har vi kommet frem til den konklusion og svar på vores opstillet problemformulering, som lyder således: ”I hvilken grad kan vi påvirke spilleren til at ændre spillestil, og hvilken effekt har dette på spillerens engagement?” Gennem testen fandt vi ud der er en stigning på 23,75% når det kommer til spillerens tilbøjelighed til at sende spillestil fra rambo til stealth og omvendt, ved at spille med et Dynamic Difficulty Adjustment system i spillet. Dette tal er baseret på de antal gange i forhold til level, som spilleren har spillet en bestemt spillestil. Vi fandt dog også ud af at engagement niveauet faldt med 40 point, baseret på et point system opsat via et spørgeskema specielt designet til at måle en persons engagement/experience. Det vil sige at vi i følge vores test har fået spillerene til markant at ændre deres spillestile, men dette var ikke hvad de faktisk ønskede, da deres engagement niveau faldt med DDA systemet på. Denne konklusion er dog ikke et sikkert nok svar på formuleringen, hvilket vil blive uddybet yderligere i næste afsnit.

10 Refleksion

Det har været et langt og hårdt projekt, med op og nedture som der altid er. Den største udfordring under dette projekt, har været at få en prototype i gang. Den originale plan for projektet var at få en prototype produceret tidligst muligt, så vi kunne komme i gang med at teste. Vi forsøgte os virkelig på at komme i gang med dette, men det viste sig at blive en virkelig sen første prototype, hvilket har resulteret i en lav mængde af test. Desuden endte vores endelige test med at være utrolig sen og tidspresset, hvilket ledte til et så lavt antal testpersoner, at vi ikke kunne besvare vores problemformulering med sikkerhed. Inden vores endelige mundtlige præsentation af projektet, har vi tænkt os at udføre flere test for at finde en bedre besvarelse på vores problemformulering. Vi er selv meget engageret i projektet, og er virkelig skuffet over vores nuværende konklusion, vi håber på at have noget bedre til fremlæggelse.

Det har dog været et positivt projekt ift. Alt andet planlægning, og der har virkelig været godt styr på at bruge scrum og Trello, til at holde styr på opgaver, deadlines og andre ting.

Desværre er dette vores sidste projekt, men hvis vi havde flere, ville fokus være på at gennemtænke spillets design og kernemekanikker yderligere inden produktionen, for at sikre en tidlig prototype kan blive produceret for at undgå samme problem.

Litteratur

- [1] Sean Baron. Cognitive Flow: The Psychology of Great Game Design. https://www.gamasutra.com/view/feature/166972/cognitive_flow_the_psychology_of_.php?print=1, Marts 2012. [Online; åbnet 13-April-2018].
- [2] IJsselsstejin, W.A. de Kort, and Y.A.W. Poels K. The game experience questionnaire. pages 4–5, 2013.
- [3] Rosa Mikeal Martey, Kate Kenski, James Folkestad, Laurie Feldman, Elana Gordis, Adrienne Shaw, Jennifer Stromer-Galley, Ben Clegg, Hui Zhang, Nissim Kaufman, Ari N. Rabkin, Samira Shaikh, and Tomek Strzalkowski. Five approaches to measuring engagement: Comparisons by video game characteristics. 1–26, 2014.
- [4] Caetano Vieira Neto Segundo, Kennet Emerson Avelino Calixto, and René Pereira de Gusmão. Dynamic difficulty adjustment through parameter manipulation for space shooter game. pages 234–237, 2016.
- [5] Uffe Thorsen and Mikkel Nielsen. Digitalt Design og Udvikling Projektoplæg til eksamsprojekt sommer 2018.
- [6] Kevin Wong. Stealth Game Design. https://www.gamasutra.com/blogs/KevinWong/20140516/217856/Stealth_Game_Design.php, Maj 2014. [Online; åbnet 02-Maj-2018].

Bilag



Boards



DDU - Eksamensprojekt



Personal | Private

Backlog (FEATURES)

Todo SPRINT (Deadline: 13/04)

In Progress

Testing

Complete

Advanced Enemy AI

Simple Level(s)

Planning

Frokost

Advanced Player Controller

Simple Player Controller

Add a card...

Add a card...

Simple DDA system

Simple Enemy AI

Add a card...

Add a card...

Advanced DDA system

Dynamic Level System

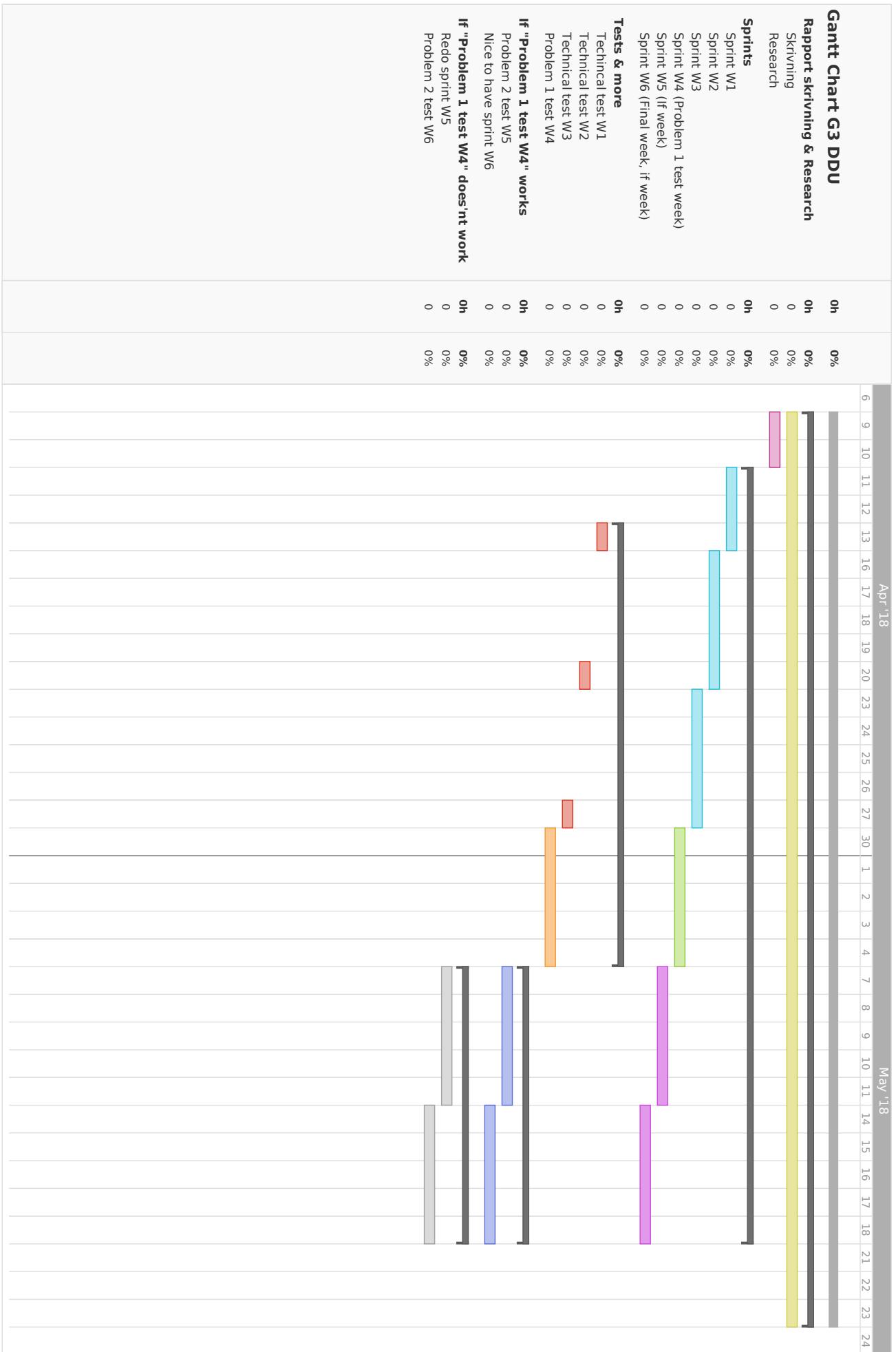
Add a card...

Add a card...

Add a card...

Add a card...





R: Rambo

S: Stealth

I: ideelle rute

