Prosjektbeskrivelse
Steam Buccaneers: Search for the Seven Cogs

Dangerzone - Programmerere

Bacheloroppgave

Bård Harald Røssehaug | John Olav Haraldstad | Lars Solli Hagen

Innholdsfortegnelse

Introduksjon

Vår problemstilling er: Hvordan utvikle mekanikker som gir spilleren en følelse av å være en pirat i verdensrommet, samt lage et utfordrende univers.

Bakgrunn

Hvorfor har vi valgt å gjøre det vi gjør? (vi har det i en tidligere rapport, Justin.)

Formål

Hvilket formål har spillet vårt?

Vise frem hva vi har lært

Vi ønsker å vise at vi kan produsere et produkt som kan selges (vi har det i en tidligere rapport)

Vise at vi kan utvikle et produkt fra begynnelse til start

Samarbeid mellom to grupper som gjør samme prosjekt

Via pilotproduktet vårt ønsker vi å vise frem hva vi har lært og kan, samt utvikle et spill vi kan være stolte av ved å vise at vi kan produsere et produkt som kan selges. Vi ønsker å bevise at vi kan utvikle et produkt fra begynnelse til slutt

Problemstilling

AI

Utvikling av mekanikk består da av blant annet å utvikle en god AI. Hvordan utvikle en god kunstig intelligens for at dette skal se ut som det gjør i verdensrommet?

Hvordan skrive en AI som simulerte et rom uten gravitasjon?

Hvordan bevege romskipene så de ser ut som båter på vann?

Hvordan lage en utfordrende AI som ikke er for intelligent, men heller ikke for dum?

Leveldesign

Hvordan designe spilluniverset med nok innhold, men ikke for mye så man fremdeles har følelsen av å være i et stort rom?

Hvordan designe et level med riktig progresjon?

Hvordan skape en skalering av verden som føles og ser riktig ut? Hvordan gi skaleringen en følelse av realisme?

Hvordan gi en naturlig følelse av at et level endres?

Hvordan begrense det spillbare området?

Opplæringssekvens

Hvordan lære opp spilleren på en interessant måte?

Learning by doing - Dewey

Leaning just in time - James Paul Gee

Hvordan flette opplæring med spilling?

Hvordan bruke dialog til “leaning just in time”?

Gravitasjon

Hvordan lage gravitasjon slik den er i verdensrommet?

Hvordan kan denne opplevelsen gjøres interessant i et spill, det vil si å skalere ned gravitasjonen til en spillbar verden?

Spillerkontroll

Hvordan lage kontroller som simulerer å være et skip i verdensrommet?

Hvordan lage et kontrollskjema som føles intuitivt og naturlig å bruke?

Hvordan balansere hastigheten til spillerens skip i forhold til andre objekter?

Idéutvikling

Vi hadde hyppige møter hvor vi diskuterte alt vi ville ha med i spillet, hvordan det ville fungere og så videre. Alt dette noterte vi ned i forskjellige møtedokumenter navngitt med ulike datoer, slik at vi enkelt skulle kunne gå tilbake til tidligere iterasjoner av idéer, og eventuelt bygge videre på noen av de, eller for å huske hvilke idéer vi hadde gått vekk i fra.

Prosessen av å finne ut av hva vi absolutt ville og måtte ha med, og det som hadde vært kult å ta med, men ikke nødvendig for det vi ønsket å lage, tok flere måneder. Vi hadde flere møter hvor vi brainstormet idéer, og deretter kuttet ut det som virket overflødig og unødvendig foreløpig, og beholdt det vi absolutt skulle ha med. Gjennom hele idéprosessen brukte vi idéverktøyet kjent som SCAMPER (Manktelow et al., s.a.), for å definere idéene våre. Det er et nyttig verktøy, som ble brukt til å se om vi blant annet kunne videreføre idéene våre, kombinere noen av de med hverandre, og se om noen bare ble helt overflødig i forhold til de andre systemene vi ville ha.

Når vi nærmet oss det som vi anser som sluttproduktet av ideen, noterte vi alt dette ned i et spilldesigndokument. Dette var både for at vår veileder skulle få grep på hva vi holdt på med, men også i stor grad for oss selv, slik at vi alltid hadde et dokument å gå tilbake til skulle vi bli usikre på hva vi hadde planer om. Det er også mer nyttig i steden for å spørre noen andre i gruppen, og muligens få andre beskjeder, eller noen har tolket eller glemt avgjørelsene, og heller referere til et dokument vi skrev i plenum og tok avgjørelser (Karlsen, 2008, s. 233).

Gameplay

Vi skal i dette avsnittet forklare og argumentere for valg vi tok under utviklingen av gameplay. Dette innebærer da elementene AI, Gravitasjon, spillerkontroll, oppgraderinger og kampsystemet.

AI

Utvikling av mekanikk består da av blant annet å utvikle en god AI. Hvordan utvikle en god kunstig intelligens for at dette skal se ut som det gjør i verdensrommet?
Hvordan skrive en AI som simulerte et rom uten gravitasjon?
Hvordan bevege romskipene så de ser ut som båter på vann?
Hvordan lage en utfordrende AI som ikke er for intelligent, men heller ikke for dum?

En kunstig intelligens i dataspill, herretter kalt ai, er et dataprogram som simulerer en intelligent oppførsel fra ikke-spillbare karakterer. Disse kan komme i form av for eksempel dyr, fiender eller lagkamerater. Donald Kehoe sier "Spill ai trenger ikke være sannsende eller klar over egne evner og mangler" (Kehoe, 2009, egen oversettelse), han sier så at målet med en ai i spill er å simulere intelligent oppførsel for å gi spillerne en troverdig motstander som ikke er umulig å beseire.

I vårt spill trengs det en ai som kan ta egne valg basert på fastsatte regler, som byr på både utfordring og underholdning. I tillegg til at ai'en primært skal være en motstander for spilleren må den også unngå farer i spilluniverset. Dette innebærer da å kjempe mot gravitasjonen fra de ulike planetene i spillverdenen, unngå å kollidere med andre skip og ikke kollidere med butikkene i universet.
Det må også se ut som om ai'en er båter på havet, og må derfor ha regler for hvordan den kan manøvrere. Hovedutfordringen ble å få svingingen til å virke naturtro, og få ai'en i kamp til å posisjonere seg på en måte som gjorde at sidekanonene var rettet mot spilleren.

I starten ble det gjort forsøk på å finne ferdiglagde koder vi kunne hente fra "Unity Asset Store", Unity sin egen butikk hvor vanlige brukere kan laste opp sine egne produkter og selge dem for en ønsket pris. Dessverre funket ikke disse til vårt formål. De aller fleste var laget med tanke på fiender av typen menneske eller monstre, men ikke til kjøretøy.

Utviklingen av vår ai gikk som følge av dette gjennom en rekke ulike iterasjoner, hver av dem et steg nærmere det ønskede resultatet. Det ble nødvendig å finne inspirasjon fra andre kilder, og adaptere det inn i vår ai. Vi skal i dette avsnittet gå gjennom alle iterasjonene og argumentere for valgene som ble tatt.

Marinen er hoved fienden i spillet. Disse vil patruljere solsystemet og aktivt gå inn for å drepe spilleren dersom spilleren kommer for nær en av disse skipene. Etter hvert som spilleren beveger seg mot den endelige bossen i spillet vil marinen få oppgradert skipene sine i form av helse og våpenkraft for å kunne by på motstand og underholdning gjennom hele reisen.

Cargo skipet er ment som en enkel fiende som skal gi spilleren tilgang til en større mengde penger enn det marinen kan gi. Den er designet for å være svak og enkel å drepe, men den vil alltid forsøke å flykte fra spilleren og være en plage frem til spilleren har tatt den nok igjen til å enkelt klare å skyte den ned.

Bossen er finalen i spillet. Den har fire ganger så mange kanoner enn det spilleren har, og vil være en vanskelig motstander for spilleren. Den vil opprettes på et fast punkt i spillbrettet, og spillet avsluttes når spilleren klarer å beseire bossen.

Hvordan navigerer ai seg?

Det finnes i hovedsak tre ulike navigasjonsteknikker en ai kan benytte seg av for å navigere seg rundt i spillverdenen, og disse er; "Ikke se fremover, ta et steg av gangen", "kalkulere hele stien på en gang" og variasjoner av A*. A*brukes i grid baserte spill, hvor karakterer beveger seg fra rute til rute.
Vår ai benytter seg av "Ikke se fremover" metoden. Alle skip vet hvor målet er, og vil kjøre mot det, men de vet ikke hvilke hindringer som vil oppstå ettersom skip og måner kontinuerlig endrer posisjon i verden. Med denne kunnskapen i bakhodet ble det mulig å finne ut hvilken type pathfinding som best passer for vårt spill.

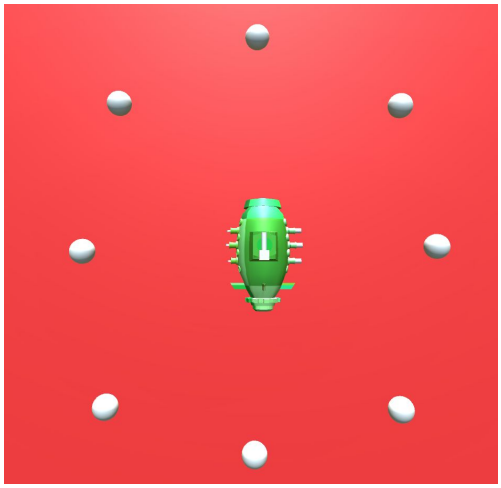
Navigasjon

Første iterasjon

I starten ble det brukt Unity sitt eget NavMesh Agent system til å kalkulere hvordan vår ai skulle komme seg fra A til B (pathfinding). Flere forum på nettet diskuterte dette systemet og gav det skryt for å være enkelt å bruke og lett å lære. Vi fant også en rekke videoer som kunne lære oss hvordan å bruke systemet effektivt, så vi valgte å gi det en sjanse.

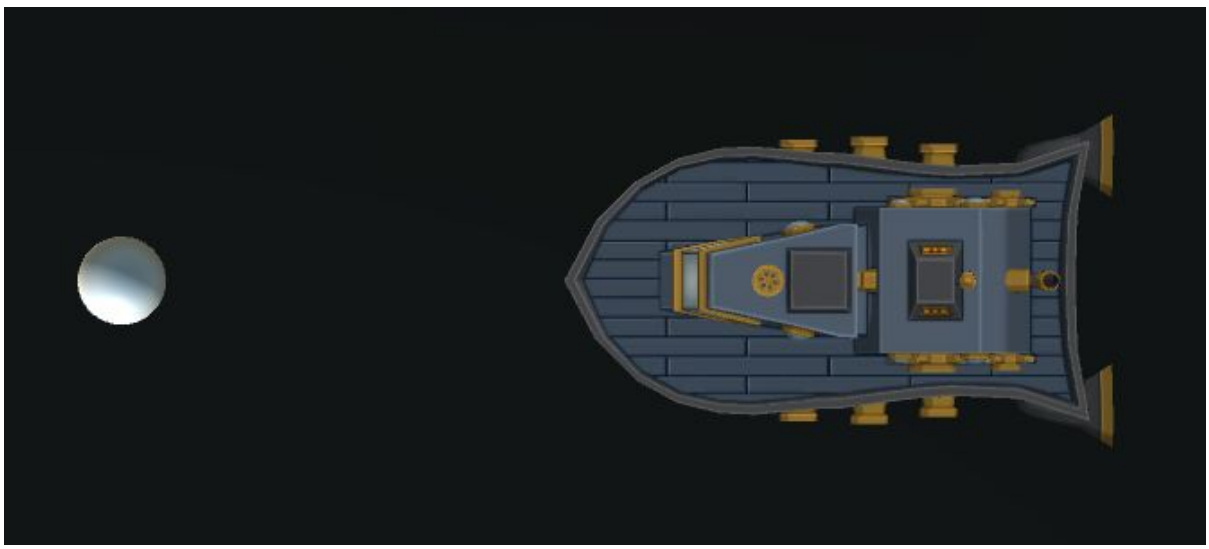
Unity har et innebygget system for å dekke bakken med et lag som NavMesh agenten kan se, og hvor som helst på denne bakken kan agenten bevege seg. Dette systemet heter "NavMesh". Systemet dekker automatisk bakken med en NavMesh. og er meget effektivt for spill som skal benytte seg av en åpen verden med flere ai'er som går rundt samtidig.
NavMesh Agent er en ferdiglaget egenskap som legges til et hvilket som helst object i Unity (komponent), og agenten vil gå mot punktet, eller objektet, du sier er dens mål. Vi satte agenten på vår ai for at skipet skulle gå mot spilleren.

I første omgang gikk agenten direkte mot spilleren, men det ble raskt oppdaget at dette gav et annerledes resultat enn vi ønsket, da dette førte til at ai'en kjørte rett inn i spilleren ved alle mulige anledninger. Det er nemlig ingen måte å si til agenten at den skal kjøre opp på siden av målet, den kan bare stoppe når den har oppnådd en X distanse mellom seg selv og målet, eller kjøre helt inntil. En løsning på dette var å danne en ring av Unity's innebygde primitive sphærer rundt spilleren, hvor agenten ville gå mot disse heller enn spilleren. Disse vil henvises til som "spillerballer".



Spillerballene er de hvite sirklene som omringer spillerens skip. Disse roterte sammen med spilleren. Agenten ville kjøre mot den ballen som var nærmest ai'ens skip, noe som gav en simulering av at ai'en kjørte etter spilleren mens den svingte og kjørte fremover. Dersom spilleren sto i ro ville agenten kjøre til den ballen som var nærmest marineskipet, stoppe opp, for så å rotere til sidekanonene pekte mot spilleren.

Å bruke NavMesh Agent virket som en god løsning, men dessverre oppførte ikke ai'en seg som en båt. Unity sin agent er laget for å fungere best på menneskeliknende enheter eller dyr, objekter som gjerne kan snu seg uten å måtte bevege seg i en bue. Vi ønsket at ai'en skulle rotere som et skip i havet, ved å måtte kjøre fremover mens den svinger, eller ved å rotere mens den står i ro. Løsningen ble å lage et eget objekt for agenten, og et eget objekt for ai-skipet. Det er dette som definerer andre iterasjon av ai'en.



Andre iterasjon

Agenten er den hvite spheren, og marinen (ai-skipet) er det store skipet til høyre. Marinen vil rotere mot agenten dersom den ikke har denne direkte forran seg, og kjører fremover så lenge de ikke kolliderer. Nå kunne vi legge inn regler på hvor raskt skipet kunne rotere seg, som både gav oppførselen vi ønsket og balanserte ai'en mot spilleren ettersom den ikke lenger hadde overlegne navigasjons muligheter.

Spillerballene roterte i første omgang ikke sammen med spilleren, og ettersom agenten bytte til hvilken som var nærmest marinen ville den endre ball kontinuerlig så lenge spilleren roterte. Å gjøre slik at ballene ikke roterte med spilleren fjernet dette problemet, men det fikset ikke at dersom spilleren roterte til å kjøre motsatt vei i forhold til det marinen gjorde, ville marinen måtte ta en stor sving mens spilleren kunne skyte uten motstand. Grunnen til dette er at agenten stod på en nesten statisk ball, og marinen kjørte mot denne samtidig som spilleren roterte. Hverken agent eller marine tok hensyn til at spilleren nå var rotert til å kjøre motsatt vei av det dem stod, noe som gav spilleren en urettferdig fordel.

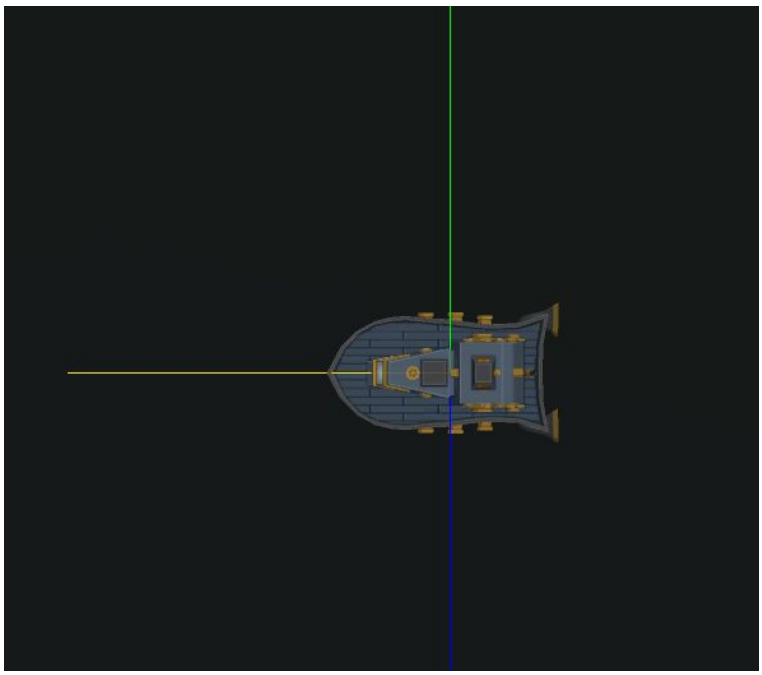
Ved å plassere en sphere forran marinen og gjøre at agenten kjørte til den spillerballen som var nærmest denne, heller enn den som var nærmest skipet, fikset vi litt på problemet. Spheren var såpass langt ute at avstanden mellom den og marinen var større enn avstanden spillerballene hadde mellom hverandre. Dette førte til at innen marinen var kommet til en spillerball var det en ny som var nærmest spheren, og da måtte agenten finne en ny som marinen kunne kjøre mot. Nå kunne marinen rotere rundt spilleren selv om spillerballene nesten var statiske mens spilleren tok en sving, men vi kunne ikke kontrollere hvilken vei marinen kjørte, den kjørte bare mot den nærmeste sphæren.

Endelig iterasjon

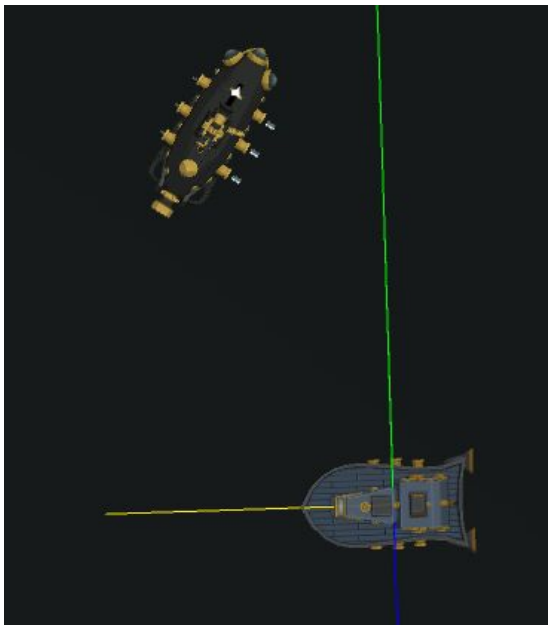
Det ble gjort to forsøk på å skrive navigasjon av ai fra bunnen av, og resultatene ble meget ulike. Den første versjonen, et forsøk på en kopi av Unity sin NavMesh Agent, feilet fullstendig. Før den ble slettet bestod den av over 200 tester som sjekket om påstander stemte eller ikke (if-tester), hvor testene var av typen "hvor er spiller i forhold til ai" og "hva er rotasjonen til spiller i forhold til ai". Her ble den relative x og z posisjonen spilleren hadde i forhold til ai kalkulert, slik at koden kunne si "dersom spillerens relative z posisjon er større enn 0.5 og dens relative x posisjon er større eller lik 5, da skal ai sjekke forskjellen i rotasjonen mellom seg selv og spiller.". Når den så hadde regnet ut

forskjellen skulle den ta ulike valg basert på hvor store forskjellene var, og hvilken retning deres rotasjoner var i forhold til hverandre.
Vi regnet ut hvor mange tester det ville bli dersom vi skulle ha en reaksjon for hver 45grad i en rotasjon, og hver relative x og z posisjon fra -0.5 til 0.5, med en økning på 0.1 ved hver test, og vi kom frem til at det ville bli mellom 12000 til 16000 tester. Med andre ord ikke en god løsning. Disse testene skulle tatt sted ved hver oppdatering i koden, noe som ved all sannsynlighet ville forårsaket lav bildeoppdatering og gitt spilleren en dårlig opplevelse.

Den andre versjonen er kraftig inspirert av Sid Meier's Pirates. AI'en de bruker i dette spillet er veldig enkel, men den fungerer veldig bra. Reglene den følger er å alltid måtte kjøre fremover, og roterer slik at den har kanonene vendt mot spilleren.



Vi adapterte dette inn i vårt spill og sørget for at vår ai gjøre det samme, med noen ekstra endringer. Vi ønsket at ai'en forsøker å holde seg på en minimumsavstand fra spilleren, men samtidig ikke være for langt unna. Dette ble enkelt løst ved å regne ut avstanden mellom de to objektene. Vi kastet så en rett linje ut fra høyre og venstre side av ai'en (RayCast), som fungerte som sensorene for dens kanoner. Ved å regne ut den relative x posisjonen til spilleren kan ai enkelt som om den har spilleren på sin høyre eller venstre side. Negativ verdi betyr at spilleren er på venstre side, positiv verdi betyr at spilleren er på høyre side. Derretter sjekker ai den relative z posisjonen. Negativ betyr at den har spilleren bak seg, positiv betyr at den har den foran seg. Når den har gjort disse fire testene kan vi si til ai at den enten skal rotere mot venstre eller høyre.



Bilde: Her vil marinen svinge mot venstre for å vende kanonene mote spilleren.

Når en av sensorene på siden av marinen, altså den blå eller grønne sensoren, treffer spilleren vil kanonene på samme side avfyre kanonkuler mot spilleren. Det vil så gå et sekund før kanonene får lov til å avfyre nye kanonkuler, men den vil fremdeles forsøke å ha spilleren innenfor kanonenes synsvinkel. Avfyrte kanonkuler vil slettes når de når en distanse på 300 enheter (meter i spillet) mellom seg selv og spilleren. Dette er for å forhindre at spilleren skal se kanonkulene forsvinne mens de er synlige på skjermen.

På bildet over kan vi se at det også går en RayCast fremover, og denne har fargen gul. I tillegg til sensorene som sjekker om kanoenene kan skyte eller ikke er det enda et sett med RayCasts, og disse ser etter planeter, butikker og andre ai'er i verden. Ved å bruke disse kan all ai oppdage elementer i universet de må unngå, og manøvrerer seg unna fare basert på hvilken sensor som har oppdaget hindringen. Sensorene som går ut til siden er ikke synlige på bildet, da de er plassert på nøyaktig samme sted som sensorene som ser etter spilleren.
Sensorene som ser etter hindringer, som planeter, andre skip og butikker er korte for å forhindre at marinen forsøker å unngå hindringer som er såpass langt unna at de er å anse som ufarlige.

AI'en i spillet vil unngå planeter dersom de er innenfor en av skipets sensorer, men registrere ikke på noen måte gravitasjonen en planet eller måne har. Dette er et designvalg for å la spilleren kunne bruke gravitasjonen til egen fordel uten å måtte bekymre seg om at ai'en er for smart og stadig unngår å kjøre innenfor et gravitasjonsfelt, eller forsøker å få spilleren mellom seg selv og planetene.

Med dette hadde vi oppnådd målet om å ha ai i spillet som beveger seg som et skip, som påvirkes av gravitasjon og som klarer å navigere seg på en troverdig måte i spilluniverset. Hastigheten på enhver ai kan balanseres i forhold til spilleren ved å endre noen få variabler, som gjør det enklere å balansere og modifisere etter ønske og behov.

Marine

Første iterasjon

Marinen får automatisk tildelt helse og nivå av våpen basert på avstanden spilleren har til spillets startpunkt. Startpunktet er determinert med en helt vanlig kube som er plassert i nivået, og "SpawnAI" scriptet bruker denne for å regne ut hvor stor avstanden er mellom spiller og startposisjonen. Denne har blitt tildelt navnet "GameOrigin". Denne kuben er ikke plassert i origo.

Kanonenes nivå blir satt i "setCannonLevel" funksjonen i scriptet "SpawnAI". Variabelen "temp" holder på verdien tilsvarende 2% av avstanden mellom objektene, så er avstanden 100 vil temp være 2. Ved å bruke 2% har marinen en balansert endring på antall oppgraderte våpen gjennom hele spillet. Originalt var tallet 10%. Det virket som et greit tall, men etter bare 1000 meter ut i spillet var det allerede 100% sjanse for at alle våpen på marinen var oppgradert. Ved å endre dette tallet til 2% må avstanden mellom spilleren og startpunktet være 5000 meter for å oppnå samme prosent, og størrelsen på vårt univers er 6000 meter. Det er nå mye jevnere spredt med våpen oppgraderinger til marinen utover i spillet, heller enn at alle marinene er fult oppgraderte før spilleren har rukket å komme seg en femtedel ut i spillet.

En sekvens i koden gjentas så seks ganger (for-loop), tilsvarende alle de seks kanonene på skipet. Hver gang opprettes et tilfeldig tall mellom 0 til 100 som brukes i testen for å se om en kanon skal bli nivå 2. Dersom dette tallet er mindre enn "temp", som tilsvarer 2% av avstanden mellom spiller og origin, vil kanonen bli nivå 2. Det opprettes så et nytt tall mellom 0 til 100, og det sjekkes om dette er mindre enn 50% av verdien til tem, med andre ord dobbelt så liten sjanse for å få lavere tall. Er tallet lavere enn "temp" blir kanonen nivå 3.
Etter denne loopen sjekkes avstanden mellom spiller og origin. Her skjer det bestemte tester som sjekker om marinen har fått oppgradert nok kanoner til et minimums nivå i forhold til avstanden mellom spiller og origin. For eksempel skal det være minst 2 nivå 2 kanoner når avstanden er over 200meter. Stemmer ikke dette settes en tilfeldig kanon til nivå 2, slik at kravet er nådd.

Denne løsningen lar oss dynamisk oppgradere marinen etter hvert som spilleren beveger seg utover i spillet. Det ville vært mulig å sette bestemte milepæler i koden, for eksempel at ved hver 200meter skal en ny kanon oppgraderes en gang, men det er ikke morsomt. Det gjør også spillet mye mer monotont når alle fiendene er like, og det ikke er noen variasjon i vanskelighetsgradene. Vi anser det også som spennende at det er en liten sjanse for å få en sterk fiende tidlig i spillet, men også en svak fiende senere i spillet.

Helsen til marinen tilsvarer 1% av avstanden mellom spiller og origin. Dersom avstanden er 100m vil da helsen til marinen være 1. Det er satt inn en if-test rett under som sjekker om helsen til marinen er under 20, og dersom dette stemmer settes helsen til 20. Det gjør marinen til en enkel, men ikke for enkel, fiende de første minuttene i spillet slik at spilleren kan bli kjent og komfortabel med kontrollene.

Vi måtte forsikre oss om at marinen alltid ville klare å ta igjen spilleren, da det ellers kunne resultere i en uendelig jakt om ikke. Når marinen går til kamp mot spilleren får den en økt hastighet frem til den har tatt igjen spilleren. Vi har satt at distansen mellom marinen og spilleren skal være minst 60 meter. Når marinen har nådd denne distansen er den nærme nok til å kunne skyte og treffe, samt spilleren føler seg presset til kamp. Når avstanden har blitt 60meter eller mindre vil hastigheten til marinen bli lik spillerens maks hastighet.

Andre iterasjon

Den femte versjonen av marines ai inkluderte en rekke spennende endringer. Nå opprettes det opp til 10 mariner som kjører rundt på kartet samtidig. Når en marine opprettes vil den samtidig få et punkt i verden den skal kjøre mot. Hvilket punkt er ikke så nøye, ei heller om den kjører mot eller bort fra spilleren, da det viktigste var å befolke universet på en måte som lar spilleren se andre skip når kameraet er zoomet ut. Marinene slettes også når de kommer for langt unna spilleren, så dersom en marine ble opprettet bak spilleren, med et mål om å kjøre videre bak spilleren, vil det resultere i at den slettes relativt raskt og en ny vil opprettes.
Denne løsningen lot oss gi spilleren en følelse av å ha et univers med relativt mye liv i seg, uten å måtte fylle hele kartet med mariner som kjører rundt fra starten av.

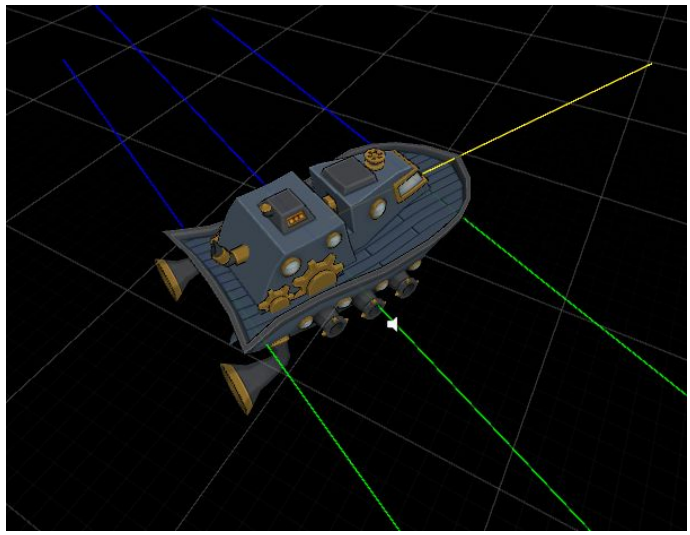
Når en av marinene kommer nær nok spilleren vil alle andre enn den marinen slettes fra nivået. Marinen som lever vil så gå inn i en kamp-modus og begynne sin jakt på å drepe spilleren. Når marinen dør vil “SpawnAI” scriptet begynne å befolke universet igjen, og hele syklusen gjentas. Den originale ideen var å ha flere mariner i kamp mot spilleren samtidig, men det ble for vanskelig å sørge

for at marinene unngikk hverandre i tillegg til å angripe spilleren på en effektiv måte. Ofte begynte de å kjøre inn i hverandre, eller havnet i en evig pressekamp da veiene til to eller flere sv dem krysset.

For å forhindre dødtid ble det opprettet en timer som tikker oppover med 1 hvert sekund. Når denne når en bestemt verdi, i vårt tilfelle 20 sekunder, vil marinen nærmest spilleren gå i kampmodus og angripe spilleren. De andre marinene slettes, slik som ellers når en marine går til angrep, og timeren vil restartes. Timeren restartes hver gang en marine går til kamp eller dersom bossen lever for å forhindre å starte en unødvendig kamp.

Endelig versjon

Det absolutt aller meste fra femte iterasjon er å finne under denne versjonen av marinen. Den største endringer en to ekstra linjer (RayCast) som sjekker om den treffer spilleren for å gi flere muligheter for marinen å skyte på spilleren. Vi oppdaget flere situasjoner hvor marinen egentlig kunne skutt og muligens truffet spilleren, men den ene rette strålen (RayCast) ikke så spilleren skjøt den ikke.



De to strekene i midten og den gule streken i fronten tilhører samme test. Disse kastet direkte fra marinens skip, og sjekke retter hindringer marinen må unngå. De kortere linjene i akter (bak) og baugen (front) på skipet er de nye sensorene som ser etter spilleren. Kommer spilleren inn i en av disse vil kanonene som ser mot spilleren bli avfyrt.

Når en kanon avfyres vil også et lydklipp spilles av. Det er lagt til en Audio Source (lydkilde) komponent på den midtre sidekanonen på hver side av skipet. Dette er for å komme rundt systemet Unity har for avspilling av lyd; dersom en lydkilde får beskjed om å spille av et lydklipp mens det allerede spiller av et lydklipp, vil det avslutte det pågående klippet og starte på det nye. Dette vil høres veldig rart ut i et spill. Derfor vil den midtre kanonen på venstre side spille av en lyd når kanonene på den siden skyter, og den midtre på den høyre siden når den siden skyter. At det er de midtre kanonene er helt tilfeldig.

Partikkeleffektene er plassert på marinen sin ferdiglagde kopi av spillobjektet (Prefab), og simuleringene deaktiveres når marinen opprettes. Når helsen er på 66% av sin originale helse vil røyk simuleringene starte. Når den har 33% gjenstående helse vil flammene også oppstå. Disse er underobjekter av marinen, så når marinen slettes vil også simuleringene slettes.

Det er til enhver tid bare 5 mariner ute på kartet samtidig. Det gir en fin balanse av liv på minikartet, spilleren fiender å oppdage og fjerner risikoen for at mariner begynner å kollidere med hverandre dersom de skulle opprettes relativt nær hverandre.

Cargo

Første iterasjon

Enkelt forklart er cargo skipet en invertert marine. Den opprettes, får et tilfeldig punkt den skal kjøre mot, og passer seg for pirater heller enn å jage dem. Forskjellen vises så snart spilleren kommer for nær, for cargo skipet vil flykte og kjøre i motsatt retning av spilleren heller enn å forsøke å gå til kamp. Dette gjøres ved å aktivere funksjonen “flee()” i scriptet “AImove”.

Slik som når marinen kjører mot spilleren sjekker denne koden de relative x og z koordinatene den har i forhold til spilleren. Den vil alltid forsøke å ha spilleren på sin negative z koordinat, det betyr at den har spilleren bak seg, og den vil forsøke å ha den relative x differansen så nær 0 som mulig. Er ikke disse kravene oppnådd vil den rotere enten mot høyre eller venstre til de er det.
Dette betyr at uansett hvordan spilleren kjører vil cargo skipet alltid kjøre motsatt vei, så lenge den ikke kommer nær en hindring som krever at den tar nødvendige unnamanøvrer.

Andre iterasjon

Gjennom spilltestene ble det klart for oss at ingen likte at cargo skipet flyktet med en gang det så spilleren. Nå må spilleren skyte på skipet for at det skal flykte, eller spilleren må allerede være i en pågående kamp når et cargo skip oppdager spilleren.

BOSS

Første iterasjon

Bossen vil ikke få kanonene sine oppgradert på samme måte som marine og cargo skipene gjør. Dette er den endelige kampen i spillet, så vi kan ikke risikere at spilleren møter på noe annet enn det bossen var designet for å være. Den har ekstra mange kanoner på hver side, og alle er fult oppgraderte. Hver side har en ventetid på 1 sekund etter å ha vært avfyrt før de kan avfyres igjen, så det er nært umulig å beseire bossen

slik den er nå. Den ble opprettet samtidig som marinen når den var under sin fjerde iterasjon, noe som betyr at kanonene bare har en sensor på hver side av skipet, fra sentrum av skipet, som ser etter spilleren.

Bossen i spillet legger ut bomber bak seg hvert 5. sekund, og disse sprenger dersom et skip eller en kanonkule treffer dem. Når de sprenger opprettes en datastruktur (array) av kollisjons detekteringer som sjekker om det er noen andre kollisjons detekteringer innenfor en angitt radius, og disse vil legges til i arrayen. Det kjøres så en loop gjennom arrayen som ser etter spillobjekter med komponten rigid kropp (RigidBody). Disse får påført seg en kraft, og distansen til bomben determinerer kraften. Lenger avstand tilsvarer lavere kraft.
Sammen med dette vil alle "AIMove" script pauses i 1 sekund. Dette er for å gi spilleren en følelse av å miste kontroll etter å ha blitt skadet av en bombe, men det skjer også med alle ai'er (marine og boss) som påvirkes av bomben. Dette ble gjort både fordi det er logisk at også bossen skal påvirkes, samt det ble ubalansert at bossen kunne kjøre videre mens spilleren var et hjelpeløst mål.

Andre iterasjon

Det ble tidlig oppdaget at bossen hadde store problemer med å oppdage spilleren pga sin egen størrelse. Spilleren kunne uten problemer sno seg unna sensoren til bossen, og skyte uten større problemer. Vi fikset dette ved å legge til sensorer som gikk til venstre og høyre både på de bakerste og de fremste kanonene. Det er nå såpass trangt mellom sensorene at spilleren ikke kan unngå kanonene. Er kanonene på en side vendt mot spilleren, og spilleren er forran dem, vil bossen med meget stor sannsynlighet registrere dette og avfyre mot spilleren.
Vi økte ventetiden mellom hver gang en side kan avfyres med et sekund, så nå går det to sekunder før en side kan avfyres på nytt. Dette er et tilfeldig tall vi ønsker å prøve ut for den neste spilltesten.

Som med marinen er partikkeleffektene plassert på bossen sin ferdiglagde kopi av spillobjektet (Prefab), og simuleringene deaktiveres når bossenopprettes. Når helsen er på 66% av sin originale helse vil røyk simuleringene starte. Når den har 33% gjenstående helse vil flammene også oppstå. Disse er underobjekter av bossen, så når bossen slettes vil også simuleringene slettes.

Alle levende skip vil flykte når bossen opprettes. Dem som er i kamp mot spilleren vil også flykte. Dette er både for å gi en dramatisk effekt av at noe farlig holder på å ta sted, men samtidig rydder det spillbrettet vårt for uønskede skip. Vi vil at spilleren skal kunne fokusere helt på bossen, uten at andre skip kommer i veien og også vil være med på kampen.

Opprettelse av AI

Marine

Det ble lagt inn at marinen vil opprettes et godt stykke unna spilleren. Dette var for å forhindre at spilleren skulle se marinen plutselig dukke opp i spillet. Hvor marinen opprettes er en automatisk utregning i scriptet "SpawnAI", hvor den tar to tilfeldige tall mellom spillerens posisjon og spillerens posisjon + 3000. Er spilleren i origo blir det da et tilfeldig tall mellom 0 og 3000. Dette gjøres 2 ganger; en for x og en for z posisjonen.
Det skal så sjekkes om verdiene skal være positive eller negative. Dette gjøres ved å først opprette et tilfeldige tall mellom 0 til 10, og er tallene høyere enn 5 skal x være positiv, er det lavere blir x negativ. Så gjøres det samme for z verdien.
Vi legger så sammen den nye x verdien med spillerens x verdi, og den nye z verdien med spillerens z verdi. Disse verdiene er nå spillerens nye posisjon. På denne måten sørger vi for at marinen alltid opprettes relativt til spilleren posisjon, og et godt stykke unna spilleren. Y kalkuleres ikke da alt tar sted i 0 i y-planet.

Før marinen opprettes sjekkes det noe allerede befinner seg på opprettelsepunktet. Koden vil opprette en datastruktur (array) av kollisjons detekteringer på opprettelsepunktet, og ser om det er noen andre objekter innenfor angitt radius, 10 meter, som også har en kollisjons detektering komponent koplet til sitt objekt. Er det ingen objekter innenfor radiusen vil ai'en kunne opprettes på det punktet. Er det derimot et annet objekt der vil koden opprette et nytt potensielt punkt å opprette ai'en på, og kjøre samme test om og om igjen frem til den finner et ledig punkt.

Cargo

Cargo skip vil opprettes 200meter forran spilleren relativt til spillerens rotasjon. Det betyr at uansett hvilken rotasjon spilleren har, vil cargo skipet opprettes 200meter forran spillerens front. Er det noe i

veien for cargo skipet der der skal opprettes vil det trekke et nytt tilfeldig opprettelsepunkt på samme måte som marine skipene gjør.

Boss

Bossen skal ikke opprettes på tilfeldige steder slik som cargo og marine skipene, men på et bestemt sted i spillverdenen. I "SpawnAI" sjekkes det om spilleren er nærmere enn 150 meter fra punktet hvor bossen skal opprettes, og dersom det stemmer skal boss opprettes. Om ikke skal det opprettes en normal marine. I "AIMaster" scriptet er det så en boolsk verdi som heter "isBoss", og denne er bare sann dersom det er bossen som opprettes. Denne brukes for å forhindre at bossen skal slette alle andre skip fra scene, da vi ønsket at spilleren skal kunne se at skipene flykter når han/hun nærmer seg bossens opprettelse punkt.

Oppgraderinger

I starten av produksjonen planla vi at vi skulle ha med oppgraderinger og at dette ville være en del av progresjonen til spilleren. Progresjonen ville da fungere ved at spiller vant noen kamper, oppgraderte skipet og kunne da fortsette lengre ut i spillverdenen og slå større fiender.

Vi startet med å planlegge en enkel tre-nivå inndeling av spillverdenen som skulle tilpasses etter tre nivåer av våpen spilleren kunne kjøpe. Ved å gi spilleren tre våpen på hver side av sitt skip, har spilleren en del å oppgradere. Videre så vi at vi trengte noen flere variabler på spillet sitt skip. Å bare variere våpen skade kunne bli litt kjedelig, så vi la til skip fart og skip beskyttelse. (Ikke ferdig)

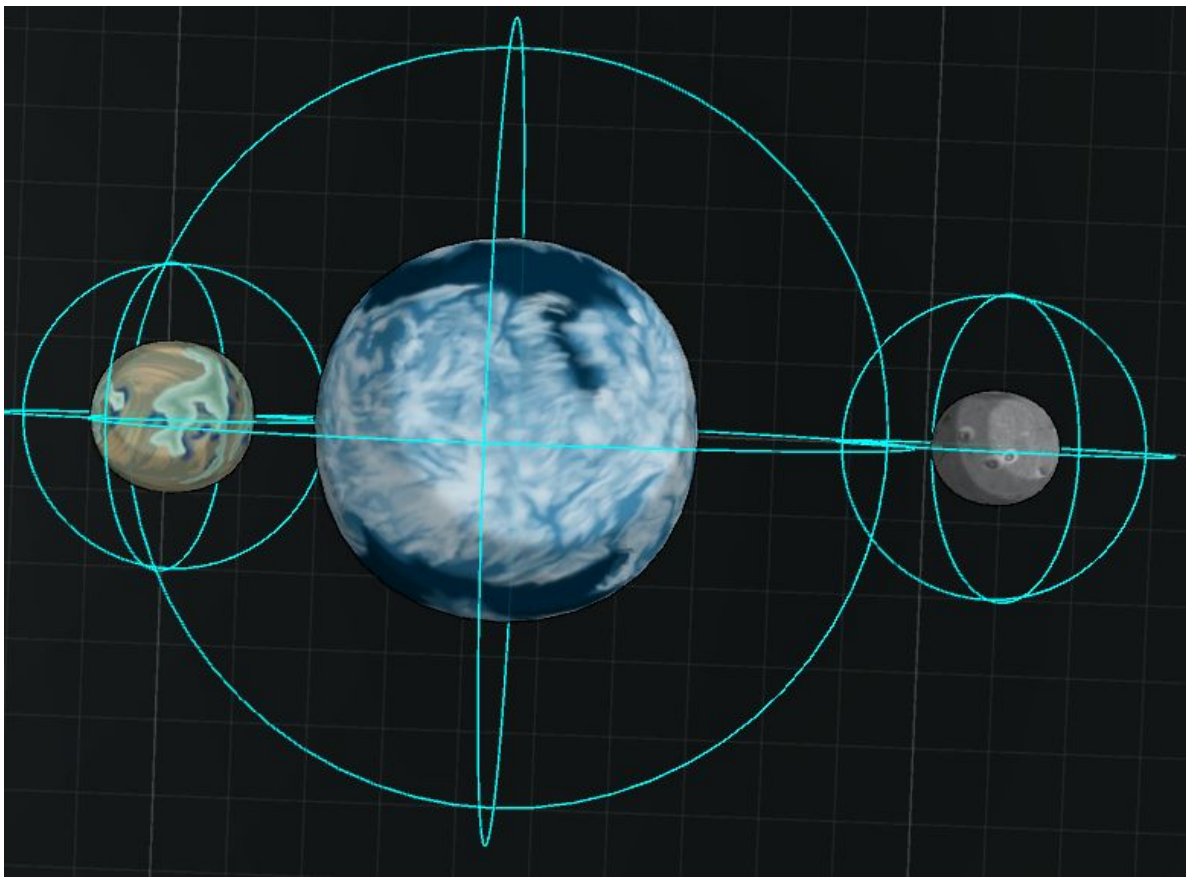
Gravitasjon

Gravitasjon er en essensiell del å ha med når spillet ditt skal være i rommet. I vårt tilfelle så ville vi bruke dette til å gjøre det om til en spillmekanikk. Gravitasjon påvirker alt som har en masse og har tiltrekningskraft etter hvor stor massen er. Ved å ha flere forskjellige masse på planetene og månene i spillet har dette spennende effekter på de andre spillobjektene. Eksempler på andre spillobjekter er spiller skip, kanonkuler som blir skutt av spiller og fiender og fiende skip.

Vårt mål for gravitasjonen var at vi skulle lage det tilnærmet slik den er i virkeligheten, men skalert ned i spillbar form. For å få det ned i mindre form måtte vi prøve å teste hvor stort feltet skulle være og styrken. Dette var vår største utfordring med gravitasjonen.

Vi fant ut at vi trengte å bruke to former av gravitasjon. En som trekker spillobjektene som er nevnt over, mot planeten og en som fikk måner til å roterer rundt planeter. Å kunne simulere en fysisk

korrekt bane til en måne er ikke nødvendig i et spill og krever for mye. Isteden for roterer vi bare månen rundt planet.



Bilde: Planet med to måner. Indigo streker rundt illustrer lengden gracitasjon påvirker andre objekter.

Når vi planla og ha tyngdekraft på planetene regnet vi med at dette ville bli en ganske vanskelig ting å programmere og å få til å fungere helt riktig. Derfor ble det planlagt at det ville bli brukt 2 uker for å ferdigstille denne koden. Dette viste seg å være galt. Etter litt søking på nettet fant vi et script som laget gravitasjon på tingen vi festet scriptet til i Unity. Ikke bare åpnet dette opp med to uker med ekstra tid, men gjorde at utviklingen av AI og spillerkontroller kunne bedre tilpasses til gravitasjonen.

Spillerkontroll

Noe av det viktigste, om ikke det viktigste i et spill, er det å få selve karakteren du styrer til å føles bra ut, se bra ut, i hvert fall i et 3-persons synspunkt, og fungere feilfritt.

Spillerkontrollen tok utgangspunktet i den tidlige prototypen for et kontrollordning utviklet høsten 2015, et simpelt script hvor spilleren bare ble flyttet fremover ut i fra dens lokale x-retning, og roterte rundt sin egen z-akse. Prototypen hadde også en kanon på taket, som alltid hadde sin rotasjon vendt

mot musepekeren, og stasjonære kanoner på begge sider som kun skyter rett frem fra skipets høyre eller venstre side.

I hovedsak så fokuserer spillet vårt på kamp og bevegelse, det å unngå både kuler og planeter, så kontrollene må gi god respons og reaksjonstid. Men vi ville fortsatt ikke gjøre dem for responsive slik at det faktisk føles som om du er ute i verdensrommet hvor det ikke er noe luftmotstand som senker farten din. Måten vi har fått spilleren til å gå fremover er ved å legge til force, altså kraft. På denne måten vil spilleren akselerere opp til maks hastighet, isteden for å alltid ha en konstant fart så fort man trykker på tasten for å bevege seg fremover. Dette har vi valgt fordi det er er slik fysikk fungerer, det føles mer ekte, og det kan på mange måter øke vanskelighetsgrad, både når man er i kamp med andre skip, og hvis man havner for nært andre planeter.

Som nevnt tidligere så har spillerentre kanoner på hver på høyre og venstre side av skipet, disse aktiveres ved å enten trykke på Q-tasten eller E-tasten. Spilleren har evig ammunisjon til disse kanonene, men som balansering har vi gjort disse mye svakere enn hovedkanonen på taket. Disse kanonene er i stor grad ment for å kunne brukes dersom spilleren er tom for ammunisjon til hovedvåpenet, eller som et tillegg til å supplementere skade på fiendene. Vi så derimot at under den første spilltesten, at spillerene kun brukte disse sidekanonene i steden for å bruke hovedvåpenet, som viste oss at hovedkanonen var overflødig i denne iterasjonen, og at spilleren heller ville bruke pengene sine på å oppgradere andre deler av skipet, og reparere det. Dette førte til at vi måtte rebalansere både sidekanonene og hovedkanonen, for å gjøre det åpenbart at hovedkanonen er det beste våpenet å bruke.

Hovedkanonen er en kanon som er plassert på taket på skipet, denne fungerer ved at kanonløpet er orientert mot musepekeren, og skyter når man klikker med venstre musetast. Denne er ment for å gi spilleren en ekstra fordel i kamp, hvor man da lettere kan sikte seg inn på et mål og skyte dem, med en kule som gjør mye mer skade en sidekanonenes kuler gjør. Denne krever ammunisjon for å kunne brukes, som blir kjøpt i butikker rundt om i solsystemet. Som nevnt i forrige avsnitt var det veldig få som brukte dette våpenet, selv om det er mye bedre enn kanonene på siden. I tillegg la vi merke til at spillerene irriterte seg over musepekeren som bare fløt på skjermen, når de aldri brukte denne kanonen. Dette fikk oss til å tenke at musepekeren burde være skjult, og man må gå inn i en kampmodus for å kunne se musepekeren, og kunne skyte med kanonen. På denne måten vil ikke musepekeren obstruere skjermen.

Spilleren svinger skipet ved å bruke A- og D-tastene. Når spilleren svinger roterer også skipet over på siden, avhengig av hvilken vei man svinger. Dette er gjort for å få det til så se mer levende ut, og mer realistisk, ved at man da setter kraft mot siden, og skipet "tipper" litt over, slik som seilskip gjør når de svinger. Det har også innvirkning på hvordan sidekanonene fungerer, ved at disse også roterer med skipet, slik at om man skyter midt i en sving, vil kanonene skyte diagonalt opp eller ned, og kan dermed bomme på målet sitt ved å skyte enten over eller under. Dette er gjort som et balansetiltak for å gjøre sidekanonene svakere, og gir nok en grunn til å bruke hovedkanonen, etter som denne vil skyte rett uansett hvilken rotasjon skipet har, som igjen er gjort bevisst for å gjøre dette våpenet til et bedre valg å bruke i enkelte situasjoner.

(Må finne noe kilder til alt dette over)

Hvis vi går videre til hvilken informasjon spilleren trenger, og hvordan vi formidler denne, kan vi liste det opp ut i fra prioriteringer (The Art of Game Design: A Book of Lenses, Second Edition, Schell, J. 2015, s. 267)

1. Helse
2. Umiddelbare omgivelser
3. Fjerne omgivelser
4. Penger
5. Ammunisjon
6. Kart/minikart
7. Kompasset

Vi deler denne listen inn i frekvensen man må være bevisst på denne informasjonen, vi starter først med informasjonen vi trenger å vite til enhver tid. Her er umiddelbare omgivelser viktig, spesielt i vårt spill, hvor omgivelsene kan være en stor fare for spilleren. Er det fiender i nærheten? Er spilleren for nærme en planet? Blir man trukket mot den? Er man inne i gravitasjonsfeltet og vil skyte seg selv? Hvilken bane har dine og fiendens kuler? Alt dette er viktig informasjon for spilleren for å passe på at han eller hun overlever. Dette er hovedsaklig den informasjonen man må tilegne seg nesten konstant. Videre kan vi se på informasjon man trenger å se underveis, men ikke konstant. Her kommer helse, fjerne omgivelser og ammunisjon inn. I vårt spill har vi ikke en typisk representasjon av hvor mye helse spilleren eller motstanderene har. Her må man heller være oppmerksom på sitt eget skip, og se hvordan selve modellen ser ut. De fjerne omgivelsene må man ofte tenke over, slik at man kan gjøre valg i tid før man plutselig krasjer inn i en planet, eller hvordan man skal håndtere en fiende som kommer fra utenfor skjermen, og ammunisjon må man ha kontroll over, slik at man alltid er klar til

kamp. Denne informasjonen er ikke konstant viktig å holde øye på, hvis man for eksempel bare kjører rundt uten farer. (her må vi legge til kompasset også/minikart)

Og sist, informasjon som man bare behøver å vite innimellom, som er hvor mye penger spilleren har. Denne informasjonen er ikke like viktig mens man seiler rundt i rommet som den andre informasjonen nevnt ovenfor, ettersom penger ikke er en ressurs man nødvendigvis får umiddelbar bruk for når man seiler rundt. De viktigste tidspunktene å vite dette på er hvis man har øyne på en oppgradering man vil kjøpe og vil samle opp penger til man har råd, eller bare passe på at man har nok penger til ammunisjon og eventuelle reparasjoner. Likevel har vi valgt å ha denne ressursen konstant vist på skjermen, slik at spilleren lett også kan se hvor mye penger man får etter en kamp.

Kamp

Her snakker vi om kamp i spillet

Kanonkulene

Både spiller og all ai deler det samme scriptet "AIProjectile" for å sjekke om kanonkulene treffer noe. Det er også dette scriptet som gir kulene en fysisk kraft når de opprettes, som er med på å bestemme hastigheten de har i spillet og gi en simulering av fysiske og ekte kanonkuler som blir avfyrt. Scriptet sletter kulene dersom de kommer langt nok unna spillerens skip, og tester om kanonkulene treffer spilleren, en ai, en planet eller en butikk. Treffes en av disse skal kulene slettes, korrekt mengde skade skal påføres og en eksplosjon partikkelsimulering skal spilles av der kula traff. Det spilles også av et lydklipp.

Kanonkulene - AI

Mens spilleren øker skaden på kanonkulene sine ved å kjøpe oppgraderinger i butikken, må ai'en regne ut og bruke funksjoner som skaper en viss tilfeldighet i håp om å få oppgraderte kanoner.

Som forklart i "AI Marine, første iterasjon" blir en kanonkules nivå determinert av kanonen den avfyres fra, og kanonens nivå blir kalkulert når skipet opprettes. "SpawnAI" kjører "setCannonLevel" funksjonen som tar distansen spiller har til GameOrigin objektet i meter, og tar så 2% av dette tallet og gjør det om til sjansen ai'en har for å få en oppgradert kanon. Måten funksjonen sier om en kanon

skal være oppgradert eller ikke er ved å sjekke om 2% av distansen er større enn et tilfeldig tall mellom 1 og 100. Er tallet større blir kanonen level 2. Så trekkes et nytt tilfeldig tall mellom 1 og 100. Er halvparten av 2% av distansen større enn det tilfeldige tallet blir kanonen level 3. Det betyr at dersom distansen spilleren har til GameOrigin er 1000meter blir 2% av distansen 200meter, og halvparten av 2% blir 100meter.
Denne dataen lagres i en datastruktur av typen heltall (array av type int). Arrayen vil hver gang en test er gjennomført lagre et tall, 1, 2 eller 3, hvor tallene symboliserer kanonens level. Disse tallene kan scriptet "AIsideCanons" bruke for å determinere hvilke kanonkuler som skal avfyres.

Når en ai er opprettet vil "AIsideCanons" starte med å sjekke hvilken type skip scriptet befinner seg på. Er det en marine vil den sette hver av kanonene til å bli samme nivå som arrayen opprettet i "SpawnAI". Er det et cargo skip vil det gjøre det samme, forskjellen er at arrayen bare vil kjøres to ganger istedenfor 6 for å ta hensyn til at skipet bare har 2 kanoner. Er det en boss settes alle kanonene automatisk til level 3.
Når kanonene oppgraderes er det ved å kopiere dataen fra "SpawnAI" inn i arrayen "cannonLevel". Dette er en array som består av alle opprettelse punktene for kanonene, og scriptet vil bruke dataen lagret i denne for å bestemme hvilken kanonkule som skal opprettes. Det er viktig å lagre dataen med en gang ai'en opprettes, for når "SpawnAI" lager en ny ai vil dataen overskrives for å tilpasses den nye ai'en.

Når ai skyter vil den opprette en kanonkule av korrekt nivå, og derretter tar "AIProjectile" scriptet over og sørger for at kulen gjør som den skal.

Level Design

I vårt spill, så er universet et ekstremt viktig tema, hvis vi ser på problemstillingen vår, om hvordan vi skal utvilke mekanikker som gir følelse av å være i verdensrommet, og i tillegg lage et utfordrende univers, så må mye av dette også reflekteres i nivået i spillet. De viktigste spørsmålene for oss ble da: "Hvordan skal vi designe et spillunivers med nok innhold, men likevel ikke for mye slik at man fredeles har følelsen av å være i et stort og tomt rom?, Hvordan skape en skalering av verden som føles og ser riktig ut? Hvordan gi skaleringen en følelse av realisme? Og hvordan gi en naturlig følelse av at et level endres?"

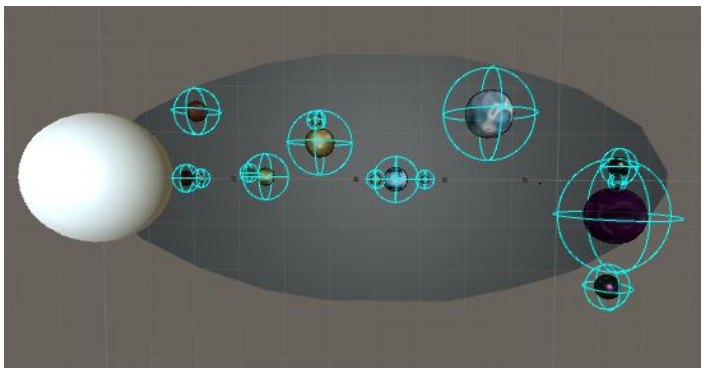
Nettopp dette er den største utfordringen med det å designe nivået, og noe vi har fått tilbakemelding på hver gang vi har hatt en spilltest, hovedsaklig at "det føles for tomt." Og dette er noe vi har jobbet mye med gjennom iterasjoner for å gjøre noe med.

I utgangspunktet var nivået i spillet basert på Solsystemet, hvor alle planetenes størrelser er tilnærmet de i Solsystemet, og avstanden mellom planetene også er i relasjon til størrelsene. Dette er selvfølgelig skalert ned en god del, hvis ikke hadde det vært et alt for stort nivå, og generelt upraktisk for den typen spill vi lager. Det eneste som i utgangspunktet ikke var i riktig skala i forhold til nivået, var stjernen, og månene til de forskjellige planetene.

(skrive her en plass om designprosess?)

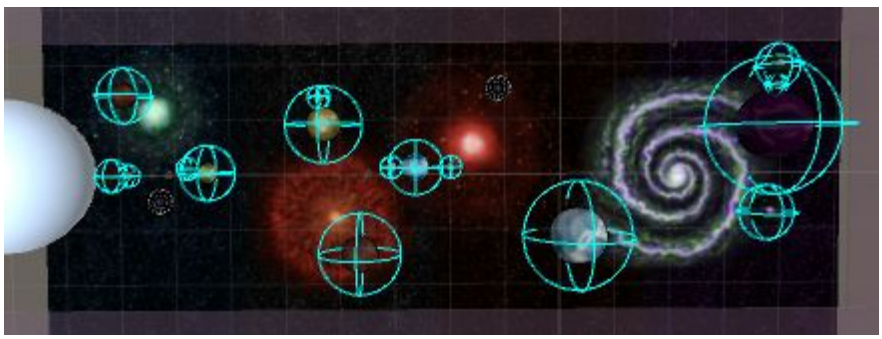
I første omgang, så ønsket vi et nivå som gikk rundt hele stjernen i et planetsystem, med det så mener vi at hver enkelt planet ville gå i bane rundt stjernen, og man skulle kunne reise 360° rundt stjernen. Etter mye drøfting kom vi frem til at det ville bli en alt for stor verden for mengden innhold vi klarte å produsere, og valgte heller å begrense verden til kun en liten del. (se figur nedenfor)
Her ville planetene rotere sakte rundt den hvite sfæren, mens de var over den grå ovale bakken, og så rotere med økt hastighet rundt stjernen når de var kjørt utenfor, og så sakte når de havnet over ovalen igjen. Et nivå som dette hadde tatt godt i bruk evolusjonsmetoden (De Jong, 2008, s 63) Denne baserer seg på at når man gjenbruker nivået eller er i et nivå over en forlenget tid, så vil spilleren returnere minst én gang til et område i nivået, men hver gang de returnerer, blir de presentert med andre omgivelser, eller endringer i gameplay. Evolusjonsmetoden er også en viktig metode for å få spilleren til å leve seg inn i nivået, og få spillverden til å virke levende, og derfor mer realistisk.

Dette fungerte godt for oss, og vi tenkte det var en god løsning, men etter den første spilltesten vi hadde så mente de fleste spillere på at spillverden var for tom, så vi gikk tilbake til tegnebordet og utforsket ande muligheter og løsninger.



Løsningen ble da å legge til flere planeter, og komme opp med noen nye planeter i tillegg. Vi laget skatteplaneter med to asteroidebelter rundt, hvor da spilleren må manøvrere seg mellom for å komme inn til planeten og plukke opp en skatt. I tillegg laget vi også døde skip som opprettes tilfeldig rundt spilleren på en intervall mellom fem og ti minutter. Disse skipene ligger døde og roterer rundt, med ekstra mye scrap svevende rund disse skipene. I tillegg la vi også til flere av de planetene vi allerede hadde rundt om kring, for å fylle opp alt tomrommet. Den ovale grensen på spillet ble også erstattet med en firkantet grense.

Som følger av disse endringene satt vi alle planetene til å stå stille, slik at de ikke ville kollidere med det ekstra innholdet vi la inn i spillet. Dette førte derimot til at nivået ikke virket like levende, og mer statisk. Likevel går fortsatt månene i bane rundt planetene.



Som man kan se på bildet ovenfor, så har vi fire distinkte bakgrunnselementer som ellers skiller seg ut fra bakgrunnen; galakser og stjernetåker. I et spill som vårt, hvor man er ute i verdensrommet, hvor det er mye tomrom og lengre distanser mellom objekter av interesse, er det viktig å ha landemerker (De Jong, 2008, s. 64,) som signaliserer for spilleren hvor han eller hun er, og hvor enkelete ting i nivået er plassert. Vi har selvfølgelig et minikart som hjelper spilleren med å orientere seg, men det er minst like viktig å gi spilleren en idé over hvor man er uten å kikke på minikartet hele tiden. Det er heller ikke alltid slik at minikartet ser et stort nok område til at man for eksempel vet hvor en butikk er. Et av de store og viktige landemerkene vi har, er den store grønne og lilla galaksen som ligger helt til høyre, denne forteller deg hvor bossen i spillet vil være, altså i sentrum. Så når man møter på denne galaksen, vet man nøyaktig hvor man skal kjøre. Disse galaksene er også ment å interessere spilleren i å oppdage, så kanskje kjører spilleren inn mot sentrum, og finner noe spesielt. Også bakgrunnen signaliserer spilleren hvor han eller hun er i nivået, ved å ha forskjellige farger ut i fra hvor langt ute i nivået han eller hun er. Det er fire av disse, og hvert område er like stort, og fargene går fra blått til hvitt, til rødt og så lilla. Disse vil også hjelpe spilleren til å vite hvor langt ut i nivået man er, og derfor også hvilken vanskelighetsgrad det er på fiendene i området.

Etter vi valgte å stanse opp planetenes baner rundt stjernen, så fungerer disse også til en grad som landemerker.

Implementering av GUI (Graphical User Interface)

GUI ble designet på en minimalistisk måte hvor alt av informasjon ble vist på minst mulig plass. Dette var et av målene våre ved GUIen var at spillinformasjon skulle være enkelt å finne ut og fint å se på. Vi brukte dette målet også i implementeringen der vi brukte et mindretall script for å holde kontroll på informasjon og å vise det på riktig GUI elementer.
GUI implementeringen gikk i tre steg. Først ble det designet av designgruppen, deretter ble det lagt inn i prosjektet og til sist ble informasjonen kodet inn for å vises.
De første tingene som ble laget ferdig, og også ble lagt inn først, var butikk-gui uten animasjoner og spill-gui. Disse ble lagt til uten problemer.
Neste steg var karakterportrettene. For å få frem illusjonen av et fantastisk univers introduserte vi tre karakterer. Shopkeeper, som hjelper spiller til å lære spillet i opplæringssekvensen, Marine, som er fiendene og Boss som er siste fiende å slå.
Tanken ved karakteren shopkeeper var at den skulle vise frem spillet og være familier person som spiller møtet på gjennom hele spillet. Ved å ha den karakteren lære opp spiller og selge ting til spiller, lager vi en trygg plass for spiller for å ikke føle seg så alene i universet.

Først skulle ikke marinen ha et karakterportrett, altså et ansikt bak romskipet. Vi fant ut at dette kunne hjelpe med å føles ut som et univers ved å ha et romvesen som viser seg når man skal sloss mot et skip. Andre ting som animasjon kunne vise spilleren var om fienden hadde lite liv eller eller om den hadde det veldig mye.

Med boss så ble animasjonen lagt til med dialog som spilleren hadde med bossen. Disse ble spilt av etter som det passet med dialogen. Bossen hadde også animasjon under kamp og viste om han mistet liv eller ikke.

Opplæringssekvensen

Opplæringssekvens
Hvordan lære opp spilleren på en interessant måte?

Learning by doing - Dewey

Leaning just in time - James Paul Gee

Hvordan flette opplæring med spilling?

Hvordan bruke dialog til "leaning just in time"?

Planleggingen av opplæringssekvensen var gjort av vår nivå-designer Lars, vår animatør Synnøve og kodeansvarlig John Olav. Grunnen til at det var spesielt disse tre som planla opplæringssekvense var fordi hver enkelt hadde en del som skulle inn. Lars måtte se til at opplæringssekvensen ville være bra gameplay-messig og at det ville være interessant for spilleren å spille. Synnøve har med alt av dialog å gjøre og måtte se til at karakterene fikk en dialog som passet til karakteren sin personlighet og samtidig gav den informasjonen som spiller trengte for å lære spillet. John Olav var med for å se til at tingene Lars planla kunne gjøres i koden, for så å implementere.

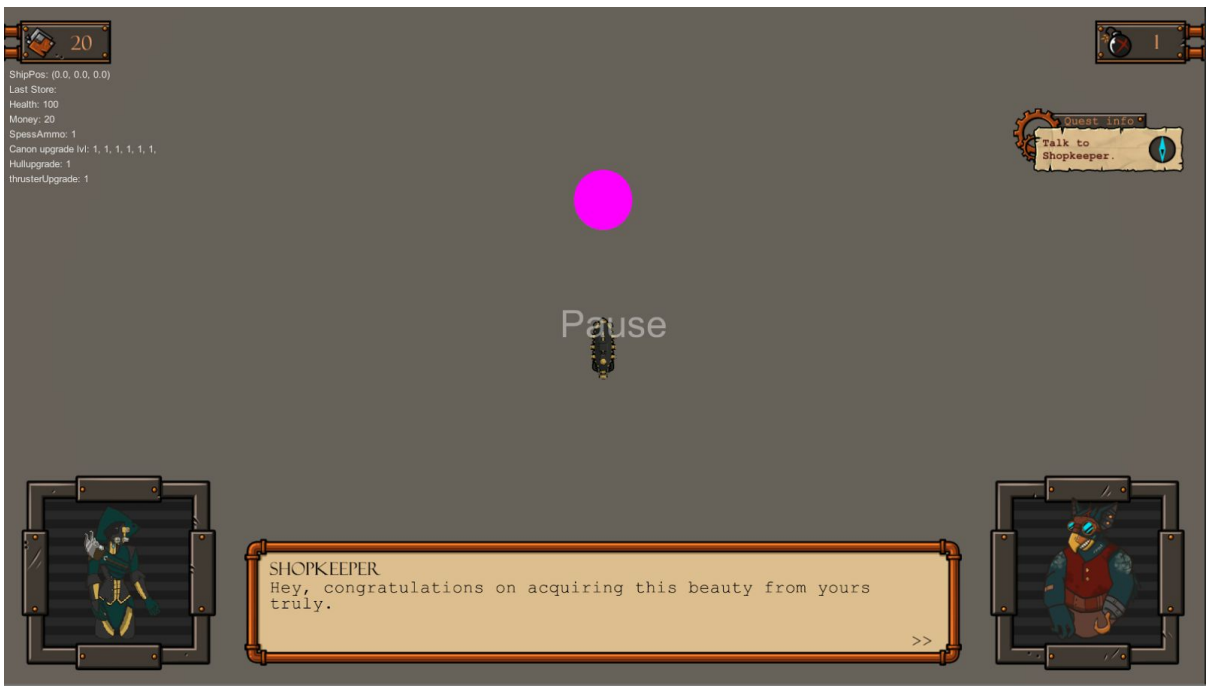
Grunnen til at vi ville ha et opplæringssekvensen var at mange strevde med å finne ut skytekontrollene på spilltesten. Vi så også på dette som en mulighet å få blandet inn historie og karakterene for å gjøre opplæringen mer naturlig. Dette ville også gjøre opplæring mer interesant. Vi tok inspirasjon fra *Harry Potter and the Philosopher's Stone* spillet der spilleren blir puttet i en situasjon og, samtidig som de lærer om karakterene, lærer kontrollene. Kontrollerene blir fortalt for så at spilleren for kontroll og kan prøve seg frem. Spilleren lærer også bare det som er nødvendig for å kunne slå et skip. Noe som er gjort etter James Paul Gee sin filosofi, learning just in time. Andre elementer som gravitasjon blir noe spilleren oppdager og lærer om ved å prøve seg frem.

Planleggingen gikk fint og en oppbygging ble skrevet.

Start

- *Plott: Protagonist kjøper romskip. Får instruksjoner fra butikkeier.*
    - *Dialog-system. Snakk*
- *Spiller lærer kontrollerne*
    - *Første steg: styre skipet.*
        - *Dialog forteller spiller hvordan kontrollere skip.*
        - *Neste steg utløses av spiller trykker inn W, A, og D. Skyting skrudd av.*
    - *Andre steg: Skyting*
        - *Dialog forteller spiller hvordan man skyter.*
        - *Neste steg utløses etter spiller skyter 10 kuler tilsammen.*
    - *Siste steg: Fiende*
        - *Fiend dukker opp. Resten av spiller pauses.*

    - *Dialog forteller spiller hva han skal gjøre og hva som skjer.*
    - *Start spill igjen og spiller bruker kontrollerne til å drepe fienden.*
    - *Neste steg utløses når fiende er drept og premie er plukket opp.*
- *Spiller lærer butikk*
    - *Første steg*
        - *Dialog: Forklare hva butikk gjør og hvordan du går inn.*
        - *Spiller kjører inn i butikk og forandrer scene.*
    - *Siste nivå*
        - *Dialog: Møt butikksjef som forklarer hvordan å bruke butikken.*
        - *Spiller fikser skip for skade.*
        - *Spiller kan forlate butikken og gå ut av opplæringssekvensen.*



Bilde: Første dialog med karaktervinduer og annet GUI.

Vi jobbet videre med dette og noen små forandringer var gjort. Som for eksempel at opplæringssekvensen fortsatte når spilleren ønsket, istedenfor at den fortsatte etter at spiller hadde trykket på knappene som styrte. Dette gjorde vi fordi vi så at alle trenger ulik tid på å venne seg til kontrollerne. Da er det greit å kunne utføre opplæringssekvensen i sitt eget tempo.

Etter å ha fått dialogen til karakterene måtte dette legges inn i koden på en god måte. Her brukte vi en array som holdt alt av dialog, så hver gang spilleren progresserer, så vil koden fortsette igjennom koden. Ikke en veldig dynamisk eller avansert metode, men det fungerte fint for den

lille opplæringssekvensen vi har. Alle andre deler av opplæringssekvensen som vi trengte hadde vi laget fra før. For eksempel oppretting av Marine, pause funksjon og fremleggelse av skatt. Opplæringssekvensen var mest å utføre allerede laget funksjoner i riktig rekkefølge enn å skrive nye funksjoner.

Det var også en del estetiske utfordringer under utviklingen. Å fange oppmerksomheten til spilleren til de riktige stedene er essentiellt for at spilleren skal få en god opplevelse og forstå hva som faktisk foregår. I sammerbeid med designerne jobbet vi da med å ha knapper blinke, skifte farger og flytte plass for å gjøre både dialog og butikk-scene mer forståelig. Blant annet måtte vi bytte farge på dialogene til karakterene etter hvem som snakket. Dialogen hadde en farge som matchet karakteren som snakket.

Bilde: Dialog og navn sin tekst farge matcher butikkeier sin vest.

Lagre/Laste inn

Vi trengte en måte for spilleren å lagre data på. Vi fant ut at å bruke "Language Specific Serialization" (Språkspesifikk serialisering) til å lagre data, fungerte godt i vårt tilfelle. Dataen blir da lagret i binærkode slik at det ikke kan leses eller redigeres av en normal spiller. Vi lærte dette via et av Unity sine tutorials. ("Referanse her").

Arbeidsmåte

For å sørge for at arbeidet ble utført på en organisert og strukturert måte ble det utnyttet en rekke nyttige verktøy. I dette avsnittet skal vi snakke om disse, hvorfor vi valgte dem og hvordan de ble brukt.

GitHub

Gjennom hele utviklingen av spillet har vi tatt i bruk GitHub. GitHub er en utvidelse av Git, som er et kildekode håndteringprogram for digitale utviklere. Det lar medlemmer

jobbe på hver sine oppgaver, for så å automatisk slå det sammen til et eneste stort og fungerende prosjekt. Alle medlemmene får så dette sammenslåtte prosjektet, slik at de kan jobbe videre med sine oppgaver.
GitHub lot oss på en enkel og effektiv måte samle arbeidene våres i et felles prosjekt, samt se på koden til de andre medlemmene for å finne inspirasjon eller hjelp. GitHub ble også brukt sammen med designergruppen, siden dette lot oss få alle elementene ferdig implementert i Unity.

Det var under de ukentlige møtene vi slo sammen de ulike pekerne, herretter kalt "branches", av lagret data til medlemmene inn i hoved pekeren som heter "Master". Vi valgte å gjøre det i fellesskap, ettersom det var relativt stor fare for konflikter. Dersom flere medlemmer har gjort en eller flere endringer i samme element (eksempelvis et object i nivået eller samme sted i et script) vil det oppstå en konflikt som må fikses før man kan lagre endringene. Ved de aller fleste anledningene kunne vi enkelt finne ut årsaken til en konflikt, men det skjedde også at det oppstod fatale feil som førte til at vi kunne sitte i flere timer mens vi forsøkte å reparere feilen. Vi har ved flere anledninger måtte lage nye branches og slette de gamle, da feilen ikke så ut til å kunne repareres. Når vi slettet en branch kopierte vi først hele prosjektmappen den innehold, og limte så dette inn i den nye branchen. Vi vet fremdeles ikke hvorfor dette fikset problemet, men vi har ikke undersøkt det nærmere.

37.5 timer i uken

Pilotprosjektet blir behandlet som en fast jobb, og det forventes at hvert medlem jobber ca 38 timer i uken, som nevnt i "Iterasjoner av Gantt" over. De fleste medlemmene jobber 8 timer om dagen, 5 dager i uken.

Unntak kan selvfølgelig forekomme. Noen ganger jobber man kanskje med noe som var forventet å ta veldig lang tid å fullføre, men plutselig kommer man på en kode som løste problemet på en mye enklere og mer effektiv måte enn planlagt. Har man da jobbet 6 timer den dagen går man gjerne hjem heller enn å begynne på noe nytt med de resterende 2 timene.

Et eksempel kan være ai. Det siste som gjenstår er å sørge for at ai opprettes i nærheten av spilleren, og at den så går direkte mot spilleren. Når denne koden er ferdig, er ai antatt å være ferdig. Det er torsdag, og Bård skal være ferdig i løpet av fredagen. Etter 6 timers arbeid fullførte han oppgaven, og har 2 timer til overs. Det er da akseptabelt at Bård drar hjem istedenfor å starte på mandagens oppgave, dersom han ønsker dette. Når det så blir fredag skal Bård starte på mandagens oppgave.

Et medlem kan jobbe overtid for å spare opp fritid. Dersom Lars jobbet mandag til lørdag har han en dag til gode, og kan ta seg fri når han føler for dette, så lenge det ikke skaper konflikt for resten av gruppen.
Dette lot medlemmene spare opp tid til å ta seg en liten ferie, som gjør godt både for kropp og sjel.

Samlet

Hver uke var det en kamp om å sikre seg et grupperom for hver dag fra mandag til fredag. De aller fleste medlemmene ønsket å møte opp på skolen for å jobbe i felleskap, av flere årsaker. Den største var at man lettere klarte å holde seg konsentrert om man satt sammen med andre som jobbet. Det er også morsommere å sitte sammen med dem man lager spill med, enn å sitte hjemme alene. Ved å sitte sammen er det også lettere å samarbeide, som er til stor fordel for oss programmerere. Noen ganger hender det at et medlem støter på et problem en annen allerede har vært borti og løst, og da kan vedkommende hjelpe personen som nå trenger hjelp.

Ukentlige møter

Hver fredag var det obligatorisk oppmøte for alle medlemmer i gruppen. Under fredagens møter gikk vi gjennom hva hvert enkelt medlem har gjort siden sist, hvordan de ligger an i forhold til planen (Gantt) og hvilke eventuelle endringer i Gantt som må foretas. Vi merget også sammen prosjektet i Git på fredagene.
Møtene var viktige for gruppen. De sørget for at medlemmene gjorde det som var forventet av dem til angitt tid, var en mulighet for alle å møtes og være sosiale, planlegge fremover og gi en følelse av kontroll.

Iterasjoner av planlegging (Gantt)

Våre planer for hva vi skal gjøre, og når, har undergått endringer kontinuerlig gjennom arbeidsprosessen dette semesteret. I denne delen skrives det om de ulike versjonene vi har hatt, og hvorfor vi har foretatt de ulike endringene.

Vår gantt er fargetkodet, som betyr at hver farge symboliserer et medlem. I radene nedover kolonne A ser vi oppgavene som skal utføres. I kolonnne ved siden av er hver oppgave farget, som forklarer hvem som har hovedansvar for hver enkelt oppgave.
For hver dag som går skal hvert medlem skrive ned antall timer de jobbet den dagen. Dette er for at alle medlemmene i gruppen skal kunne se hvordan de andre ligger an, og om de gjør arbeidet som forventes av dem. Det er selvfølgelig mulig at noen jukser på timene de har jobbet, men det er likevel en bedre ordning enn ingen notering av timer i det hele tatt.
Raden i toppen av gantt viser datoer, hvor for eksempel 08.01 betyr 8. januar. Dette er for at vi enkelt skal kunne se hvilken dato de ulike oppgavene skal utføres. Har en dato rosa ruter under seg er det obligatorisk oppmøte, da vi skal slå sammen prosjektet vårt i Git og se gjennom gantt for eventuelle endringer som må utføres. Grå ruter er ment for å skrive på rapporten, og svarte er dato for innlevering til veiledere.

Første iterasjon

I den første versjonen av gantt (se vedlegg 1.1) hadde vi kartlagt alt som måtte utføres av arbeid, hvem som skulle gjøre hva og når. Det er også inkludert at medlemmene skal skrive inn X dersom de ikke har jobbet med dagens oppgave, F dersom de har jobbet med og fullført hele oppgaven og A+rad dersom de har jobbet med noe annet enn oppgaven de er satt opp til. Dette var ment å skulle brukes dersom en oppgave ble ferdig før tiden, for eksempel gravitasjon. Ble gravitasjon ferdig noen dager før tiden, skulle John bruke A+rad, for eksempel A30, for å henvise til oppgaven han jobbet med istedenfor.

Det er et synlig hopp i John Olav sine arbeidsoppgaver. Dette skyldes dårlig fordeling av arbeidsoppgaver. Vi valgte i første omgang å ikke fylle ut dette tomrommet, da vi valgte å tro at han kunne fungere som en støttespiller de to ukene, altså veksle mellom å assistere Bård og Lars i deres oppgaver.
Vi ønsket ikke å gi ham flere arbeidsoppgaver, ettersom dette ville føre til at han måtte ta noen av de andre programmerernes oppgaver, og da ville de stå med tomrom i arbeidsplanen. Vi antok

også at vi ville møte på problemer underveis som førte til at tomrommet ikke ville bli noe problem, og mer eller mindre fikse seg selv.

Foreleserne gav klare signaler om at de forventet at alle studenter jobbet hver dag i uken, og vi valgte derfor å ikke notere ned helgene i gantt for første iterasjon. De forventet samtidig at alle studenter jobbet 37.5 timer i uken, noe som betydde 6 timer med arbeid hver dag. Vi kom frem til at et medlem kan jobbe så mye eller lite de vil, så lenge de blir ferdig med oppgaven i tide. Dette betydde at dersom noen ble ferdig 2 dager før tiden kunne de ta seg 2 dager fri.

Andre iterasjon

Det tok ikke lang tid før vi følte behovet for å implementere helgene inn i gantt. I andre iterasjon av gantt (se vedlegg 1.2) har det nå kommet grønne kolonner, som da betyr helg, altså lørdag og søndag. Vi forventet fra da av at medlemmene jobbet 8 timer om dagen, eller 37.5 timer i uken. Ved å innføre helgene og gjøre pilotproduksjonen om til en jobb føler man seg mindre presset til å måtte jobbe i helgene, noe som gjør at medlemmene kan hvile seg ut i to dager for så å prestere bedre de neste 5. Jobber derimot et medlem mindre enn forventet og ikke klarer å fullføre oppgaven i tide, er det forventet at medlemmet må jobbe inn tapt tid i helgen. Denne løsningen har vist seg å være veldig god for oss, og vi har holdt på dette systemet ut semesteret.

Medlemmer kan fremdeles jobbe for eksempel 10 timer om dagen, fire dager i uken, for så å ta en ekstra fridag. Vi forventer bare at de skal ha jobbet minimumskravet og fullfører oppgavene sine.

Det er også innført blå ruter. Blå ruter betyr obligatorisk fremføring forran klassen. Det er bare to av disse, men det er likevel greit å få dem skrevet ned slik at vi ser dem i god tid før presentasjonen.

Det ble heller ikke gjort noen forsøk på å reparere gantt for andre iterasjon, men det hadde allerede oppstått problemer. John fullførte gravitasjonen og lagre/laste mye tidligere enn antatt, så det ble nødvendig å gi ham nye oppgaver å jobbe med. Løsningen ble å sette ham på implementering av GUI (grafisk brukergrensesnitt), noe som har fungert veldig bra. Han fikk arbeid å utføre, og Bård, som vi kan se i vedlegg 1.3, fikk 2 uker ekstra på å forbedre ai'en.

Tredje iterasjon

Til tredje iterasjon av gantt (se vedlegg 1.3) ryddet vi opp i arbeidsfordelingen, arbeidstider og endret til å vise hvordan arbeidet faktisk har foregått. Hovedsakelig betyr dette at vi har fått ryddet

opp i rotet med John sine oppgaver. Som vi kan se har vi nå redusert gravitasjon til bare 1 dag, ettersom det var så raskt det gikk. Det er også lagt til hvilke dager han jobbet med hva frem til 27. januar, datoen vi endret til tredje versjonen av gantt.
Nå viser det bedre hva han skal gjøre, og når, samt vi har fått laget en jevn arbeidsfordeling frem til alle oppgaver er utført. Lars har fått noen ekstra dager på nivådesign, da han så dette som en nødvendighet, og det er lagt til hvor lenge ekstra Bård skal jobbe med ai.

Som vi kan se er det noen trange uker hvor det bare blir en eller to dager til å arbeide på spillet, men vi har valgt å ikke endre på dette da det fremdeles vil fungere som en vanlig arbeidsuke. Det blir bare 2 dager til å jobbe med spillet i slutten av januar, men til gjengjeld er det 3 dager som går bort til forberedelse for innlevering. Før den første fremføringen er det nødvendig å sette sammen det vi har og forberede seg til presentasjonen, så derfor blir uken seende ut som den gjør.
Hadde vi endret på ukene for å alltid få 5 sammenhengende dager med arbeid på prosjektet ville det overstyrt helgene og dagene vi uoffisielt har fri, noe som igjen ville ført til trettere medlemmer og lavere effektivitet.

Fjerde iterasjon

Som vi kan se i Vedlegg 1.4 ble det lagt til noen få nye punkter og ryddet opp i timefordelingene. Noen medlemmer lå foran planene, noen litt bak, så vi oppdaterte gantt til å vise den nye mengden arbeid som måtte utføres for å fullfør en oppgave.
Oppgavene som ble lagt til var “Marine AI - unngå shop, boss, planeter” og “Marine AI - spawne på patruljepunkt, patruljere, oppdage player”. Grunnet endringer i planene for hvordan vi ville ha ai’en i spillet til å oppføre seg ble det nødvendig å innføre disse to endringene til gantt.
Oppgavene ble skrevet inn like etter bossen til spillet ble fullført ettersom det ikke var mer i forhold til planene for Bård å gjøre.

Spilltestene

Under spilltestene møttes vi på skolen for å møte de andre gruppene i et samarbeid om å teste hverandres spill. Her deltok medlemmene fra de ulike gruppene, og gav hverandre konstruktiv tilbakemelding på spill, ide, og konsept. Vår gruppe noterte ned det meste som ble sagt, og utførte nødvendige balanseringer og bugfixing etterpå. Designvalg ble så utført i fellesskap med gruppen basert på tilbakemeldingene vi fikk under spilltestene.
Se “Vedlegg 2” mappen for referat fra de ulike spilltestene.

Medlemmenes tanker

Diskusjon

Konklusjon

Referanseliste

Jessen, S. A. (2008). Prosjektledelse trinn for trinn: En håndbok i ledelse av små og mellomstore prosjekter (SMPer) (2. utg.). Oslo: Universitetsforlaget.

Karlsen, J. T. (2013). *Prosjektledelse: Fra initiering til gevinstrealisering* (3. utg.). Oslo: Universitetsforlaget.

Schell, J. (2015). The Art of Game Design - A Book of Lenses (2. utg.) Boca Raton: CRC Press

De Jong, S. (2008). The Hows and Whys of Level Design (2. utg) Sjoerd De Jong

Kehoe, D. (2009). Designing Artificial Intelligence for Games (Part 1). Lokalisert på https://software.intel.com/en-us/articles/designing-artificial-intelligence-for-games-part-1