Prosjektbeskrivelse
Steam Buccaneers: Search for the Seven Cogs

Dangerzone - Programmerere

Bacheloroppgave

Bård Harald Røssehaug | John Olav Haraldstad | Lars Solli Hagen

Innholdsfortegnelse

Hvem skriver hva i rapporten?

- Introduksjon
- Idéutvikling? - Hvem enn som finner
- Gameplay
  - AI - Bård
    - Marine
    - Boss
  - Gravitasjon - John
  - Spillerkontroll - Lars
  - Oppgraderinger - John
  - Combat - Lars
- Leveldesign - Lars
- Implementering av GUI - John
- Opplæringssekvens - John
- Lagre/Laste inn - John
- Spilltester - Den som er relevant for hver unike test
- Arbeidsmåte - Bård
  - GitHub
  - 37.5 timer i uken
  - Samlet
  - Ukentlige møter
- Iterasjoner av planlegging (gantt) - Bård
- Medlemmenes (meninger/tanker?) - ALLE
- Diskusjon - John
- Konklusjon -
- Referanseliste

Introduksjon

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

Idéutvikling

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

Når vi utviklet den kunstige intelligensen til spillet, herretter kalt AI, var det noen få viktige oppgaver den skulle utføre; Kjøre rundt uten å være aggressiv, oppdage spiller, gå mot spiller og forsøke å drepe 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.

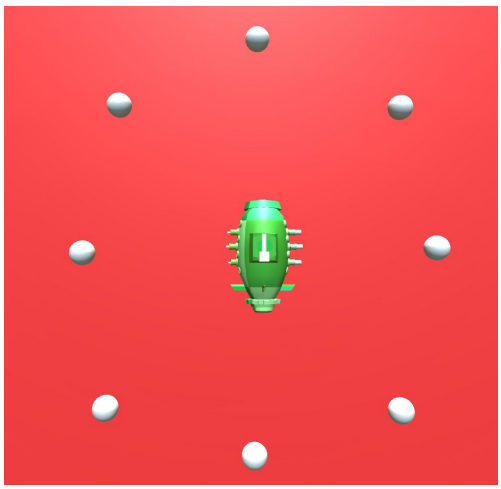
Cargo skipet er ment som en enkel fiende som skal gi spilleren tilgang til en større mengde penger enn det marinen kan gi. Den er designet for å være svak og enkel å drepe, men den vil alltid forsøke å flykte fra spilleren og være en plage frem til spilleren har tatt den nok igjen til å enkelt klare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

Bossen er finalen i spillet. Den har fire ganger så mange kanoner enn det spilleren har, og vil by på en real kamp når spilleren våger seg ut mot denne. Den vil opprettes på et fast punkt i spillbrettet, og spillet avsluttes når spilleren klarer å beseire bossen.

AI Marine - 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.

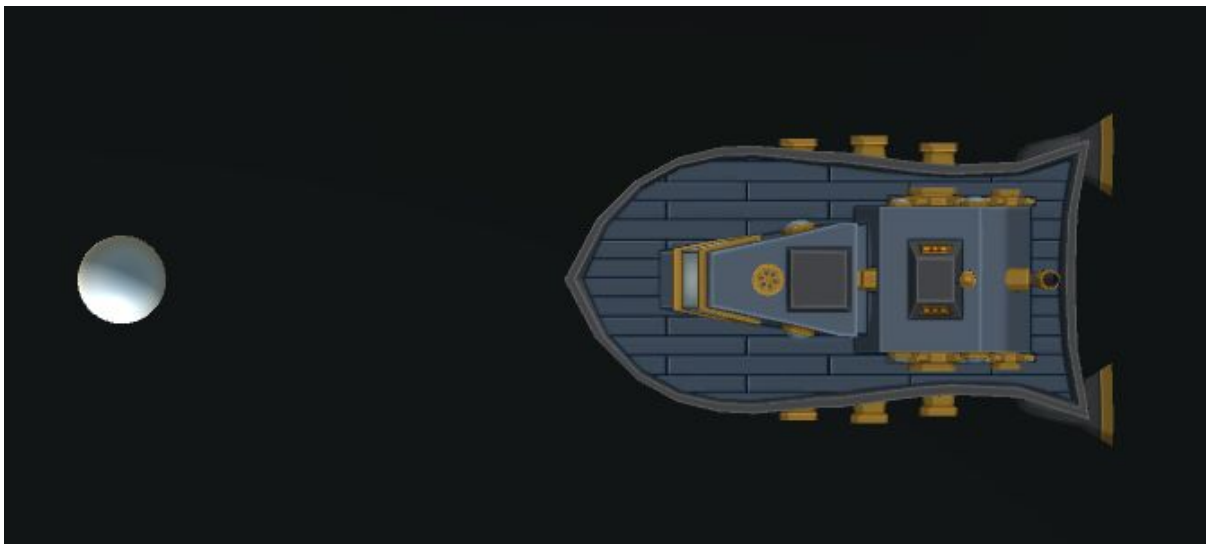
Unity har et innebygget system for å dekke bakken med et lag som NavMesh agenten kan se, og hvor som helst på denne bakken kan agenten bevege seg. Dette systemet heter “NavMesh”. Systemet dekker automatisk bakken med en NavMesh. og er meget effektivt for spill som skal benytte seg av en åpen verden med 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 da agenten på vår marine for at skipte 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 da 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.
Vi valgte å lage denn regelen for ai'en enkelt og greit fordi det virket som en logisk handling for den å gjøre. Den måtte enten kjøre rundt i ring rundt spilleren eller stå i ro, og vi valgte i denne omgang å gå med det vi følte var mest naturlig.

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.



AI Marine - 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.

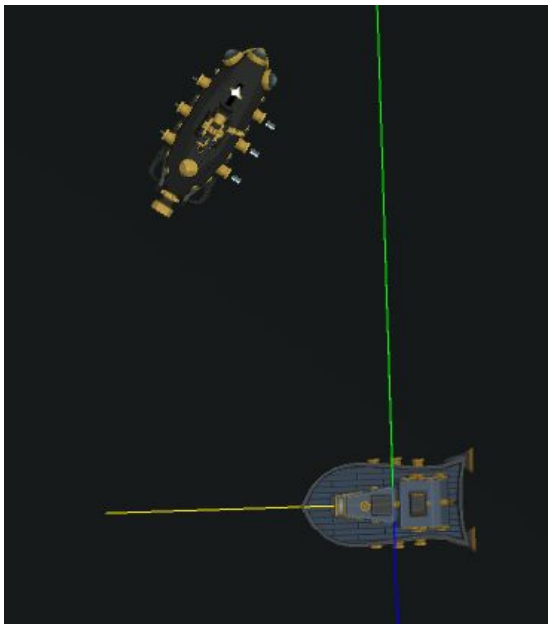
Ved å plassere en sphere forran marinen og gjøre at agenten kjørte til den spillerballen som var nærmest denne, heller enn den som var nærmest skipet, fikset vi litt på problemet. Spheren var såpass langt ute at avstanden mellom den og marinen var større enn avstanden spillerballene hadde mellom hverandre. 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.

AI Marine - Tredje iterasjon

Det ble gjort to forsøk på å skrive AI fra bunnen av, og resultatene ble meget ulike. Den første versjonen feilet fullstendig. Før den ble slettet bestod den av over 200 tester som sjekket om påstander stemte eller ikke (if-tester), hvor da 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 brukte i dette spillet var veldig enkel, men den fungerte 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 da 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.



Her vil marinen svinge mot høyre for å ta igen 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.

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.

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.

AI Marine - Fjerde iterasjon

Marinen fikk nå 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.

AI Marine - Femte 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

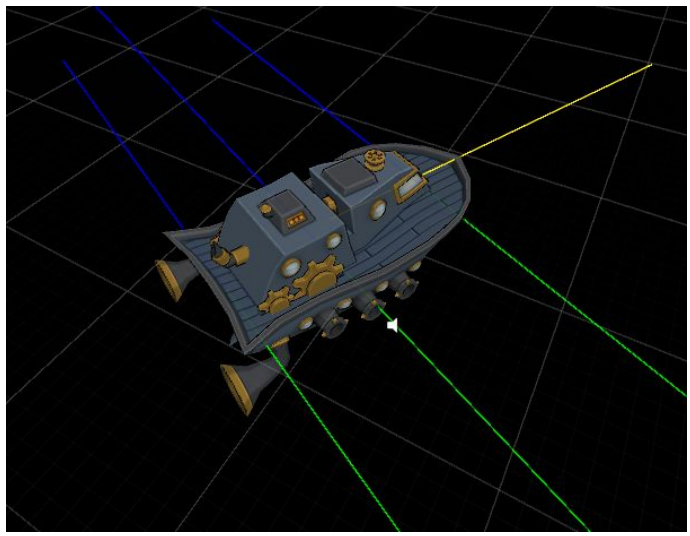
kameraet er zoomet ut. Marinene slettes også når de kommer for langt unna spilleren, så dersom en marine ble opprettet bak spilleren, med et mål om å kjøre videre bak spilleren, vil det resultere i at den slettes relativt raskt og en ny vil opprettes.
Denne løsningen lot oss gi spilleren en følelse av 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.

AI Marine - 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.
"SpawnAI" scriptet har fått en ny regel om at ingen skip kan opprettes derom et annet objekt er mindre enn 10 meter unna punktet hvor marine, cargo eller boss skipet skal opprettes. Koden vil opprette en datastruktur av kollisjons detekteringer, 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.

AI 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.

Cargo skipet bruker samme kode som marinen under 3. iterasjon for hvor den skal opprettes, og får på samme måte et punkt den skal kjøre mot. Den vil aldri snu for å skyte mot spilleren, men har kanoner med sensorer i tilfelle en situasjon hvor de kan avfyres skulle oppstå.

AI Cargo - Andre iterasjon

Eneste endringen som tok sted for andre iterasjon var at cargo skipet alltid opprettes forran spilleren relativ til spillerens rotasjon. Det betyr at uansett hvilken rotasjon spilleren har, vil cargo skipet opprettes 200meter forran spillerens front.
Etter å ha blitt opprettet vil den få et tilfeldig punkt å kjøre mot, og ellers fungere slik den gjør i første iterasjon.

AI BOSS - Første iterasjon

Bossen i spillet ble utviklet raskt og effektivt ved å bruke koden som allerede var utviklet til marinens ai. Siden marine og boss deler script måtte vi innføre noen få tester i kodene for å forsikre oss om at bestemte ting skjer dersom det er bossen som lever, men ellers undergikk de få endringer.
I “SpawnAI” sjekkes det om spilleren er nærmere enn 100 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.

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.

Dør bossen vil ikke dette registreres på noen annen måte enn ved at dens variabler resettes, så "SpawnAI" scriptet vil opprette en ny boss bare noen få sekunder etter spilleren drepte den, dersom spiller er nærme nok bossens opprettelsespunkt.

AI BOSS - 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.

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

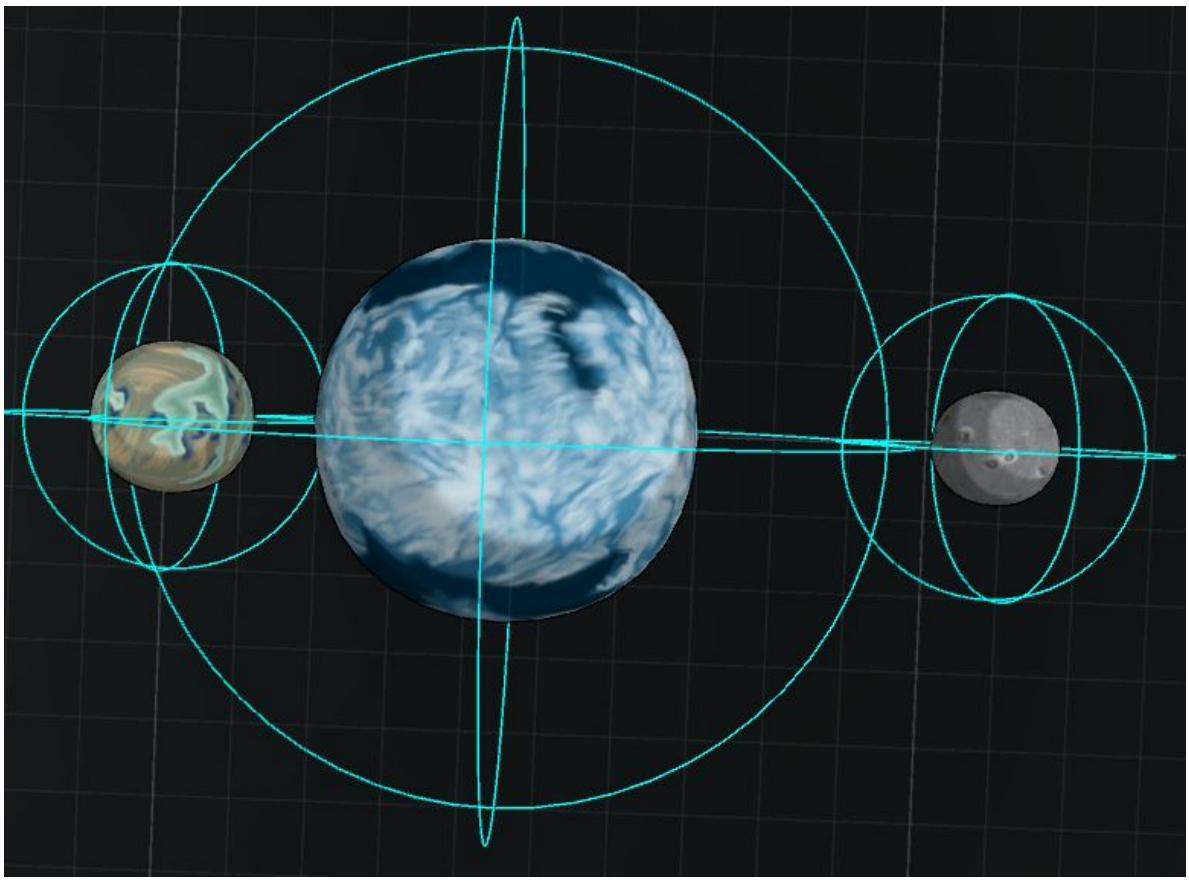
kamper, oppgraderte skipet og kunne da fortsette lengre ut i spillverdenen og slå større fiender.

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 tiltrekkingskraft 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.

Vi fant ut at vi trenkte 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 - Fjerde 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 da tallene symboliserer kanonens level. Disse tallene kommer 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.

notater:

- 1. iterasjon:

  Når en AI går til kamp slettes all annen AI.
- 2. iterasjon;

  Når en AI går til kamp kan andre mariner bli med i kampen

  3. iterasjon

  Resatt til 1. iterasjon

  AI kan ikke spawne så lenge kamp pågår

  4. iterasjon:

  Når en AI går til kamp vil alle andre AI'er som lever flykte

  AI kan ikke spawne så lenge kamp pågår
- kamera vibrerer når spiller treffes av en kule

Nivådesign

I boken "The hows and whys of level design" av Sjoerd "Hourences" De Jong, har han komponert en sjekkliste (De Jong, S. 2008, s. 26) over de 7 nøkkelelementene som et nivå vil bli bygget rundt. Vi valgte å bruke denne sjekklisten fordi det ville gi god oversikt over ha som trengtes å tenke på både under planleggingen og byggingen av nivået.

Tid

For selve planleggingen av nivået har vi totalt 7 dager bestående av 8 timers arbeidsdager, videre hadde vi 13 dager på å bygge og ferdigstille nivået i spillmotoren Unity.

Verktøy

Unity er spillmotoren som vil bli brukt til å lage designet, videre så vil en designer lage teksturer og modeller i Photoshop og Maya.

Begrensninger

sånn sett så har vi ingen klare begrensninger annet enn tid. Noe som kan være et lite problem er at kunstneren som skal lage tekstur til planetene aldri har gjort det før, og vi har ikke mye erfaring med å bygge et nivå før annet enn en som er basert på et virkelig kart av Galapagosøyene fra en tidligere skoleoppgave. vi har også designet brikker som var brukt i et brettspill laget tidligere i utdanningen.

Krav

Det vil være butikker rundt i systemet som må tas i betraktning når vi designer nivået, så nivået må designes slik at planetene rundt lett kan skape en viss følelse av gjenkjennelse så du vet hvor du er, og hvor de andre butikkene er. I tillegg så skal vi ha ulike soner hvor fiender med ulik styrke skal opprettes, og vi skal også ha en sone hvor det er plassert en boss.

Hensikt

Det er et enspiller spill med tilfeldig opprettelse av fiender og spesifikke butikker rundt om kring i solsystemet. Det er også plassering for en boss. Man skal også ha en viss følelse av oppdagelse i spillet.

Gameplay

Man styrer et romskip rundt, og har muligheten for tilfeldige møter, i tillegg til at man skal slåss mot en boss som sluttspillet, så det må være nok plass til å kunne bevege seg rundt. I tillegg så er også fysikk ganske viktig, i og med at vi har med at planeter har gravitasjonskraft, og trekker både spillere og AI, samt kulene deres mot seg om man kommer nære nok. "treghet" i rommet er også en viktig faktor, at man "sklir" når man kjører rundt, så det å ha nok plass til å skli rundt, men fortsatt ha god nok plass til å ha en responstid er viktig. I tillegg så vil det kanskje være store distanser mellom objekter, så man kjører fort. Vi har også lekt med ideen om å ha at planetene går i baner, veldig sakte, så nivået vil konstant endre seg.

Tema

Science fiction/verdensrom tema som er tegneserieaktig, så universet må på mange måter reflektere dette. Sevle stilen er også inspirert av steampunk.

"One designer may write everything down i a design document while another (...) plans it out in their head" (De Jong, S. 2008, s. 25)

Lars, som hadde hovedansvaret for å designe nivået, bruker vanligvis den sistnevnte metoden, der han heller planlegger det i hodet. Dette skaper store problemer ettersom det er vi som designer verden rundt gameplay og historie, og ikke de kunstneriske aspektene. Det gikk derfor opp for oss hvor viktig det var å så fort som mulig finne ulike bilder, eller formidlet ideene våre på papir, til de som faktisk tok seg av det kunsteriske, som å teksturere og modellere planetene, og bakgrunnene som ville være på plan i de ulike sone-inndelingene, slik at alle har en pekepinn på hvordan nivået skal se ut.

Så fort vi hadde fått visjonen på plass i hodet, fikk vi den ned på papir hvor vi laget flere forskjellige oppsett av hvordan verden skulle se ut, hvor ting skulle stå plassert, ulike soner og videre. Deretter gikk vi videre til å finne bilder av hvordan vi så for oss at de ulike planetene og bakgrunnene ser ut, og få stemning og følelse av hvordan det vil passe sammen, et såkalt moodboard. Dette viste seg å være mye til hjelp for både designeren som tok seg av teksturering av planetene, og designeren som skulle lage bakgrunner til nivået.

Noe av det store ved designet av dette nivået, er å få det til å virke som et dynamisk solsystem som faktisk lever. Vi ønsket at planetene skulle rotere rundt sola i systemet, og at spilleren også kunne reise rundt hele solen for å oppdage innholdet i spillet. I praksis var dette en stor utfordring i forhold til flyten i nivået, samt det ville bli for mye tomrom i nivået i denne omgang, uten nok innhold, og for store distanser å reise.

(se LevelDesign4 eller 5)

Løsningen ble heller i første omgang å ha nivået begrenset til et lite område, hvor planetene fortsatt vil rotere rundt solen, men ikke hele veien rundt. Her vil heller planetene havne i toppen av grensen, utenfor synsrekkevidde, og rotere ned mot den nedre grensen, når de er ute av synsrekkevidde her, vil de bli satt på toppen igjen, og begynne å rotere nedover. På denne måten får vi fortsatt følelsen av å være i et stort og levende solsystem, og det gir også litt av den følelsen av å oppdage, når kanskje den planeten som var der for 10 minutter siden, har flyttet seg ganske langt unna.

Etter første spilltesting fikk vi mye tilbakemelding rettet mot nivået i spillet. De fleste mente at nivået var for stort, at det var for mye tomrom i nivået, og at det ikke virket som om det var nok å gjøre. Selv om verdensrommet er stort, det er store distanser mellom hver gang man møter på noe som var litt av tanken, gikk vi tilbake til tegnebenken for å komme opp med løsninger for å fylle tomrommet. Noe av det første vi bestemte oss for da, var å ha med et minikart i spillet, som nevnt tidligere. Vi tok også beslutningen om at vi skulle flere planeter rundt om på nivået, selv om dette da ville gå på bekostningen av at planetene ikke lenger ville gå i bane rundt solen. Selv om det var en idé vi likte veldig godt, og kommer til å gjenoppta går vi videre med å produsere dette, så virket det nødvendig for nå. Det å sette planetene til å stå stille vil da også gjøre det lettere for spilleren å navigere seg, og gjøre seg kjent i verden.

Det vi også fikk implementert etter første spilltest, var bakgrunner i nivået, det er tre ulike hovedteksturer som repeterer seg over en stor flate i nivået, avhengig av hvilken sone man er i, dette

er også for å gi spilleren større kontroll på hvor han eller hun er, i tillegg så har vi spesielle sprites i bakgrunnen som for eksempel en galakse, eller stjernetåke, som ligger på toppen av flaten med hovedteksturen. dette vil også hjelpe på at universet føles litt tomt, ved å ha spesielle bakgrunner i nivået som man kan oppleve.

Vi fant også ut at, siden man er en pirat, må vi introdusere en form for skattejakt, hvor vi da kom opp med tre ulike ideer; Den første hvor det er planeter plassert rundt i nivået, med asteroidebelter rundt. Disse må man da kjøre gjennom, helst uten å bli truffet hvis det skal være verdt det, "lande" på planeten, og så komme seg ut av asteroidebeltet igjen. En annen idé var å ha en planet omdekket av stjernetåke, en effekt som vil komme foran kameraet og bli sterkere desto nærmere man kommer planeten. Denne tåken ville kunne kutte ut systemene på romskipet til spilleren midlertidig, som for eksempel at man ikke kunne svinge, eller skyte i et par sekunder, eller at fremoverdriften av skipet ikke stanser selv om man har sluppet fremoverknappen i noen sekunder.
Aller sist kom vi opp med ideen om ødelagte skip som står på sted i hvil, men roterer rundt. Dette er for å lettere kunne identifisere de som noe som ikke er fiendtlig, og som noe spesielt slik at spilleren vil utforske det første gangen han eller hun møter på det, for å lære seg at her er det skatter å finne. Disse skipene vil man også kunne se på minikartet, i form av et nødsignal, som igjen skal få spilleren til å dra mot det og utforske hva det er. Farene her vil da være selve marinen som også har kommet mot signalet, men i motsetning til spilleren for å hjelpe, og vil da angripe spilleren når han eller hun kommer nær.

Implementering av GUI (Graphical User Interface)

Vi utviklet også en måte å gå fra hovedspillscenen til butikkscenen og fortsatt holde på den viktigste spillerdataen som liv og penger. For å gjøre GUI enklest mulig å jobbe med, bruker vi mindre antall script for knappene. Dette gjorde vi fordi det ble alt for mange script, hvis vi skulle hatt et unikt for hver knapp. Måten vi gjorde dette på var at vi koblet scriptet til et tomt spillobjekt og brukte Unity sin innebygde knappefunksjon. Vi refererte til det ene scriptet på alle knappene, og hvilken funksjon som skal utføres.

Opplæringssekvensen

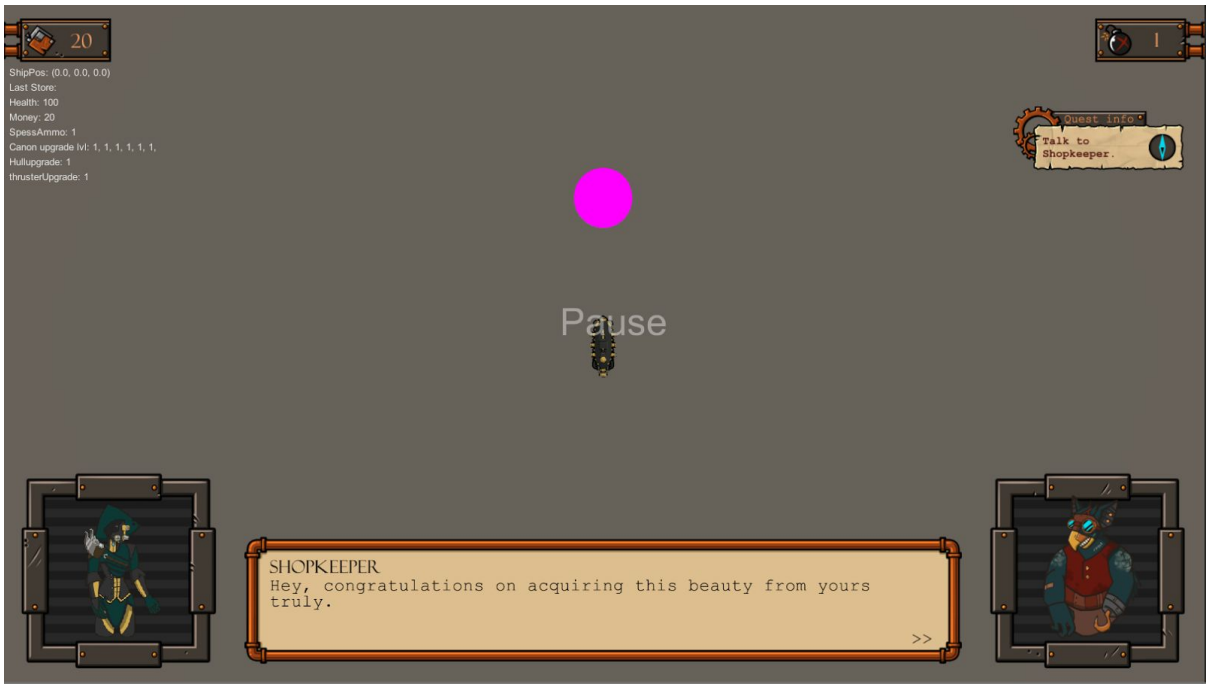
Planleggingen av opplæringssekvensen var gjort av vår nivå-designer Lars, vår animatør Synnøve og kodeansvarlig John Olav. Grunnen til at det var spesielt disse tre som planla opplæringssekvense var

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.

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, da dette lot oss få alle elementene ferdig implementert i Unity.

Det var under de ukentlige møtene vi slo sammen (merge) de ulike pekerne av lagret data (branches) til medlemmene inn i hoved pekeren (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 (commit). 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 da at hvert medlem jobber ca 38 timer i uken, sm 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. Han har jobbet 6 timer den torsdagen, og har så fullført hele oppgaven. 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 da betyr at hver farge symboliserer et medlem. I radene nedover kolonne A ser vi oppgavene som skal utføres. I kolonne ved siden av er hver oppgave farget, som da 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. 08.01 betyr da 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, da 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.

Vi valgte å ikke notere ned helgene i gantt i 1. iterasjon, da foreleserne gav klare signaler om at de forventet at alle studenter jobbet hver dag. De forventet samtidig 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 uka, 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 GUI, 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 da 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.

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