Hvad skal vi bruge AI til?

Vores designdokument specificerer, at der skal være forskellige typer af NPC´er i spillet. Disse NPC´er skal kunne opføre sig på en fornuftig måde, både i forhold til deres interaktion med Mojo, men også i forhold til deres egen interaktion med spillet. Den opførsel refererer vi til som NPC´ens AI, dens kunstige intelligens.

Der er typisk to veje til at implementere en AI, man kan benytte sig af en tilstandsmaskine eller et Behaviour Tree.  
Hvis man vælger at benytte sig af en tilstandsmaskine, skal man identificere hvilke forskellige tilstande den givne AI skal kunne befinde sig i, samt de(n) faktorer som skal udløse et skift fra en tilstand til en anden.

En FSM er god når der er tale om en relativt simpel AI, med et begrænset antal tilstande. Grunden til det er, at det er relativt let at implementere. Hvis AI´en derimod indeholder mange tilstande, eller fra starten er uklart defineret så man skal tilbage og rette i den, kan det gå markant ud over overskueligheden.

Det samme gør sig gældende hvis man indlejrer tilstandsmaskiner i hinanden. Et eksempel kunne være en AI der har en tilstandsmaskine til at håndtere ”fredelige” gøremål, mens den implementerer en helt ny når den indtræder i kamp.

For at imødegå svaghederne ved at implementere sin AI vha. tilstandsmaskiner, kan man benytte sig af et Behaviour Tree. Et BT er en orienteret acyklisk graf, hvilket vil sige at hver kant har en retning, og disse kanter er knyttet sammen så der ikke opstår cyklusser i grafen.

I vores spil skal NPC´erne løbe, hoppe, skyde og dø, hvorfor det er helt igennem rimeligt at benytte sig af en tilstandsmaskine til implementationen af deres AI.

## Hvordan er vores NPC implementeret?

Vores NPC er en massefabrikeret robotkanin, der blot er designeret EvilBunny. Modellen er importeret og oprettet efter samme metode som beskrevet i kapitlet omhandlende Mojo. Ligeledes har denne figur tilknyttet en rigidbody, en fysik-collider, og en animator med *Apply Root Motion*-funktionalitet slået til. Disse komponenter beskrives ikke nærmere her.

Vi har oprettet en ny controller-komponent til EvilBunny, da den ikke har samme funktionalitet som Mojo. Som det fremgår af Figur 1, så er udgangstilstanden for EvilBunny løb, mens de to andre mulige animationer er Jump og Attack. På samme måde som med Mojo, bliver Jump-tilstanden aktiveret af en Trigger. Mens det er en regulær boolean der benyttes i forbindelse med PlayerSeen. Grunden til det, er at når spilleren er erkendt, så skal NPC´en forblive i Attack-tilstanden indtil den ikke længere kan ses.



Figur - EvilBunny controller-komponent

For at kunne se spilleren har EvilBunny fået tilknyttet en Box Collider med IsTrigger-funktionaliteten slået til. Hvis en anden collider, som også har IsTrigger slået til, interagerer med den i Figur 2 viste kasse, så kan man via script registrere det.



Figur - Box Collider repræsenterer NPC´ens synsfelt

Udover at skulle være i stand til at kunne se, er det også en forudsætning for NPC´en at kunne opfange kollisioner på figuren selv. Den funktionalitet skal bruges for, at han kan hoppe imellem de forskellige platforme og for at opfange hvis han rammes af skud, eller af Mojo selv.

For at opnå dette er der tilknyttet endnu en collider med IsTrigger slået til, denne gang omkranser collideren kun lige figuren. (Figur 3)



Figur - EvilBunny omkranset af capsule collider

### Enemy AI-script

For at udnytte den funktionalitet der er til rådighed via de tilknyttede komponenter oprettes et nyt script, kaldet EnemyAI. I dette script implementeres den tilstandsmaskine som driver EvilBunny´s kunstige intelligens.

Første trin er at definere hvilke tilstande NPC´en kan befinde sig i. Af designdokumentet fremgår det, at den skal kunne bevæge sig imod Mojo, og angribe ham hvis han kommer ind i dens synsfelt. Det er givet at den ikke starter på samme platform som Mojo, da objektet der spawner NPC´erne er det samme som spawner platformene, derfor skal NPC´en kunne forcere distancen imellem platforme. Derudover skal NPC´en kunne dø, hvis han rammes af Mojo´s kugler.

public enum State

{

Init,

Walk,

Jump,

Attack,

Die

}

Figur 4 - AI´ens mulige tilstande

Sammenlagt udmønter disse krav sig i de tilstande som er vist i Figur 4. Af figuren fremgår også tilstanden Init, som kun køres en enkelt gang som det første når objektet oprettes, og som har til formål at initialisere de variabler som anvendes så længe scriptet kører. Metoderne der benyttes til at fremfinde de forskellige komponenter er gennemgået i KAPITLET OM MOJO!!!

Figur 5 viser den tilstandsmaskinens primære cyklus. Start-metoden køres automatisk af Unity ved objektets oprettelse, men i hvor vi normalt benytter den til at initialisere scriptet, benyttes den her til at drive AI´en. For at kunne fungere som tilsigtet er det nødvendigt at while-løkken kører uendeligt, da det ikke er til at vide hvor mange gange den skal gennemgå cyklussen før NPC´en gøres inaktiv.

IEnumerator Start () {

\_state = State.Init;

while (true)

{

switch(\_state)

{

case State.Init:

Init();

break;

case State.Walk:

Walk();

break;

case State.Jump:

Jump();

break;

case State.Attack:

Attack();

break;

case State.Die:

Die();

break;

}

yield return 0;

}

}

Normalt ville spillet (og Unity) standse og behøve manuel genstart hvis man implementerede en uendelig løkke, hvilket er hvad ”While (true)” reelt er. Men ved at ændre returtypen til en IEnumerator, og returnere Yield 0 undgås dette problem.

For hver gang while-løkken køres evalueres det hvilken tilstand NPC´en er i, og den relevante metode kaldes. Som det ses under metodesignaturen, sættes *\_state*-variablen til Init ved første kald, derfra sættes *\_state* til *Walk,* hvilket resulterer i at Walk() kaldes i den efterfølgende while-cyklus.

Figur - Tilstandsmaskinens løkke

NPC´ens standardtilstand er at gå, hvilket også fremgår af Figur 1, derfor er det ikke nødvendigt at sætte ham i gang. Walk-metoden som vidst i Figur 6 indeholder et check af om NPC´en er død siden sidst. Hvis det ikke er tilfældet kontrolleres det om spilleren kan ses, om han er forsvundet fra synsfeltet, og til sidst om NPC´en har ramt et JumpPoint.

Figur - Walk-tilstand

private void Walk()

{

//Debug.Log("Enemy Walk");

if (!CheckForDeath())

{

if(\_sight.playerInSight)

{

//Debug.Log("set");

\_anim.SetBool("PlayerSeen", true);

\_state = State.Attack;

}

else if(!\_sight.playerInSight)

{

\_anim.SetBool("PlayerSeen", false);

}

if(\_feels.NpcJump)

{

\_feels.NpcJump = false;

\_state = State.Jump;

}

}

else

{

\_state = State.Die;

}

}

An efter resultaterne af de checks som foretages i Walk-metoden transiteres der til enten Attack, Jump, Die eller intet, i hvilket fald Walk-metoden bliver ved med at blive kaldt for hver cyklus.