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.

## 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-karakteren. I lighed med Mojo, 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 udgangpunktet for EvilBunny Walk-animationen, 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 1 - 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.

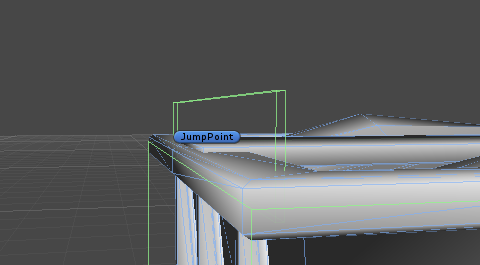


Figur 2 - Box Collider repræsenterer NPC´ens synsfelt

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. (**Fejl! Henvisningskilde ikke fundet.**) I scriptet refererers der til denne collider, som NPC´ens ”feels”. Det er med den han ”mærker” verden omkring sig.

For at AI´en kan hoppe på de rette steder, har vi til hver platform tilføjet en box collider i højre ende. (Figur 4) Denne collider er tagget med navnet: ”JumpPoint”, og da den også er en trigger, kan vi via script kalde en Jump-metode når NPC´en interagerer med et JumpPoint.



Figur - EvilBunny omkranset af hans ”Feels”-collider

Figur 4 - JumpPoint 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

}

Sammenlagt udmønter disse krav sig i de tilstande som er vist i Figur 5. 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 - AI´ens mulige tilstande

Figur 6 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;

}

}

Figur 6 - Tilstandsmaskinens løkke

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.

### Walk tilstand

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 7 indeholder et check af om NPC´en er død siden sidste kald. Hvis han er, så springes resten af metodens funktionalitet over, og tilstanden angives til Die.

Hvis det ikke er tilfældet kontrolleres det om spilleren kan ses, i hvilket fald animatoren skifter tilstand. Samtidigt angives det også at ved næste gennemløb NPC´en skifte til Attack-tilstand.

Det sidste check i metoden går på om NPC´en har ramt et JumpPoint. *\_feels.NpcJump* er en property på npcFeels.cs, som sættes til true hvis NPC´ens *feels*-collider, som vist i Figur 3, har ramt en collider der er tagget som ”JumpPoint”. Se hele metoden i Bilag XXX.

private void Walk()

{

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

if (!CheckForDeath())

{

if(\_sight.playerInSight)

{

//Debug.Log("set");

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

\_state = State.Attack;

}

if(!\_sight.playerInSight)

{

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

}

if(\_feels.NpcJump)

{

\_feels.NpcJump = false;

\_state = State.Jump;

}

}

else

{

\_state = State.Die;

}

}

Figur 7 - Walk-tilstand

### Attack-tilstand

Metoden der håndterer NPC´ens Attack-tilstand fremgår af Figur 8. Igen er der et check for at sikre at NPC´en ikke er død. Dernæst kontrolleres det igen om spilleren er i synsfeltet. Grunden til, at der det kontrolleres både i Walk() og Attack() er, at Attack() fortsætter med at blive kaldt så længe spilleren er hvor NPC´en kan se ham. Derfor skal det checkes for hvert gennemløb, at spilleren rent faktisk kan ses. Da der skal skiftes tilstand i samme øjeblik han forsvinder ud af synsvidde.

Hvis spilleren kan ses, standses NPC´ens bevægelse, og det kontrolleres om han er klar til at afgive skud. *\_isWaiting* er til for at sikre at NPC´en ikke afgiver et skud for hvert gennemløb af metoden. Det ville resultere i en endeløs strøm af kugler, hvilket ikke er hensigten. Coroutinen WaitForNextShot() som ses i Figur 9, sikrer at der går en prædefineret tidsperiode før der igen kan afgives et skud.

Kaldet til SpawnBullets-scriptet sørger for oprettelsen af en ny kugle. Dette script kan ses i dets helhed i bilag XXX

Hvis spilleren ikke længere er synlig for NPC´en, ændres både animator- og AI-tilstand tilbage til Walk-udgangspunktet

Figur - Attack-tilstand

private void Attack()

{

if (!CheckForDeath())

{

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

if(\_sight.playerInSight)

{

\_anim.applyRootMotion = false;

if (!\_isWaiting)

{

\_isWaiting = true;

\_spawnBullets.Spawn();

StartCoroutine(WaitForNextShot());

}

}

else

{

\_state = State.Walk;

\_anim.applyRootMotion = true;

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

}

}

else

{

\_state = State.Die;

}

}

IEnumerator WaitForNextShot()

{

yield return new WaitForSeconds(timeBetweenShots);

\_isWaiting = false;

}

Figur 9 - Venteperiode imellem skud

### Jump-tilstand

Som det fremgår af Figur 10, består Jump primært af et kald til en Coroutine kaldet WaitForDoubleJump(), som ses på Figur 11.

Figur - Jump-tilstand

private void Jump()

{

Debug.Log("Enemy Jump");

if (!CheckForDeath())

{

StartCoroutine(WaitForDoubleJump());

\_state = State.Walk;

}

else

{

\_state = State.Die;

}

}

I WaitForDoubleJump får NPC´ens rigidbody tilføjet kraft til dens Y-akse, hvilket resulterer i et hop. Dernæst ventes der i 1,2 sekunder, før den får endnu en påvirkning. Denne gang med en halv gange mere kraft end første hop. Dette resulterer i en fin dobbelthop effekt, der sender NPC´en fra en platform til en anden.

IEnumerator WaitForDoubleJump()

{

rigidbody.AddForce(Vector3.up \* jumpForce);

yield return new WaitForSeconds(1.2f);

rigidbody.AddForce(Vector3.up \* (jumpForce \* 1.5f));

}

Figur - NPC double jump

### Die-tilstand

NPC´ens Die-metode er til for at sikre oprydning efter objektet. I dette tilfælde er der kun tilbage, at sætte objektets tilstand til inaktivt. Efterfølgende det igen indgår i spawnpoolen.

Figur - Die-tilstand

private void Die()

{

this.gameObject.SetActive(false);

}

CheckForDeath-metoden er vist i Figur 13. Det kontrolleres om npcFeels-scriptet har registreret en kollision med et GameObject der er tagget med ”projectile”. (Se bilag XXX) Hvis det er tilfældet er NPC´en at betragte som død.

Grunden til at at NpcHit sættes tilbage til *false*, er at NPC-objekterne oprettes ved object pooling. De variabler der er påvirket af NPC-objektet, bibeholder deres nuværende tilstand når NPC´en sættes til inaktiv. For at undgå, at NPC´en starter med NpcHit sat til *true* ved næste aktivering, sætter vi den med det samme til *false* her.

private bool CheckForDeath()

{

bool dead = false;

if(\_feels.NpcHit == true)

{

dead = true;

\_feels.NpcHit = false;

}

return dead;

}

Figur 13 - Kontroller om npcFeels har registreret en kollision med et projektil