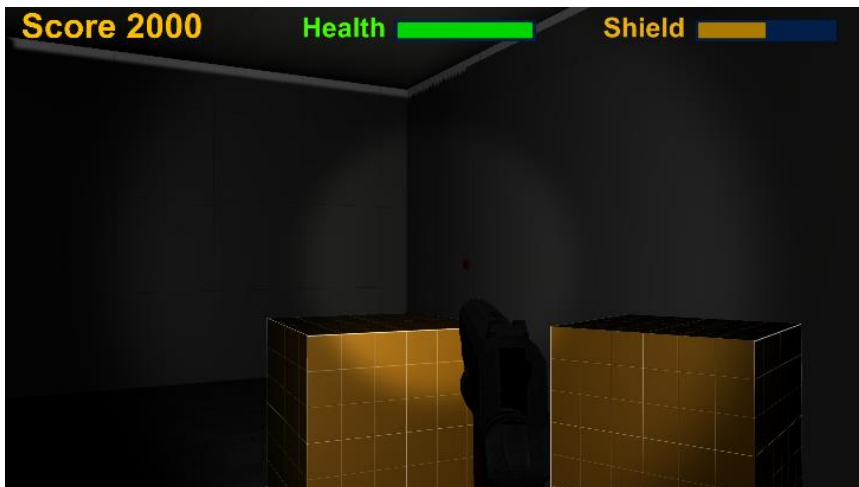


# Unity FPS

Af: Michael Hansen, Coding Pirates Furesø, 2022, version 1.00

Dokument og kode ligger her: <https://github.com/mhfalken/unity/>



Dette er en guide i hvordan man laver et simpelt 3D spil ala FPS, som vist på billedet. Det forudsætter noget kendskab til Unity, og at man selv kan skrive C# kode og finde rundt i GUI'en. Guiden udnytter at man kan finde gratis assets som er fuldt animeret og som har de fleste af de features som vi ønsker. Spillet er bevist lavet simpelt så det er lettere at lave, dvs. der er steder hvor man kan lave mere komplekse løsninger hvis man synes.

Dokumentet og koden er lavet i Unity version 2021.3. Den burde også virke i andre versioner, men der kan være små forskelle. Koden bruger det 'nye' input system.

1	Introduktion.....	3
1.1	First Person Demo .....	3
2	Bane og person.....	3
2.1	Ting man kan tage .....	4
3	Basis features.....	4
3.1	Pistol og skud.....	4
3.2	Lys og lygte .....	6
3.3	Knap og døre.....	6
4	Fjender og health system .....	7
4.1	Kanon som skyder (kan springes over).....	7
4.2	Player health system .....	8
4.3	Fjender.....	8
4.4	Fjende health system.....	10
4.5	Fjende patruljering .....	10
4.6	Hurtige fjender .....	11
5	Ideer.....	11

# 1 Introduktion

Denne guide er bevist lavet meget overordnet, da det er meningen at man skal lave det hele selv og dermed komme op på et højere Unity niveau. Guiden er derfor mere en vejledning i hvilke skridt man skal tage end hvordan man laver dem.

Guiden bruger en liste af gratis Assets fra Unity Asset store, som man selv skal hente og importere.

Gode shortcuts til at bevæge **Scene** viewet i 3D:

- Vælg et objekt og placer cursoren i **Scene** og tryk på **F** for at zoome ind på objektet. Godt hvis man skal finde et objekt i **Scenen**.
- **Alt** + venstre museklik, roterer omkring det valgte objekt
- **WASD** samtidig med at man trykker på højre museknap, det får billedet til at roterer på forskellige måder. Det kræver lidt træning at styre det.

## 1.1 First Person Demo

Opret et nyt 3D projekt i Unity.

Det første vi skal have lavet er vores Player og her bruger vi en færdiglavet first person spiller.

I *Unity Asset Store* (<https://assetstore.unity.com/>), find "*Starter Assets - First Person Character Controller*" og importer den.

Under importen skal man acceptere "Enable new input system".

Se videoen i *Assets Store* om hvordan man bruger den nye *Player*.

Load test scenen: *Assets/StarterAssets/FirstPersonController/Scenes/Playground* og forstå hvordan det hele er lavet:

- Se at alt virker
- Tryk på boksene og find dem i **Hierarchy**'et
- Forstå hvordan scenen er bygget op
- Kik i *StarterAssets* folderen og se hvad der er – vi skal bruge flere af delene.

## 2 Bane og person

Vi skal nu lave vores egen bane. Start med at skifte til banen i *Assets/Scenes/* så man får en 'tom' bane.

Opret et gulv (Cube). Det kan være en fordel at sætte **Position Y** =  $-scale\ Y/2$ , så er overfladen  $y=0$ .

Tilføj et **Material** til gulvet – bruge eventuelt nogle fra *StarterAssets/*

Tilføj nu en Player. Det gøres ved at bruge menuen: **Tools->Starter Assets->Reset First Person Controller**.

Se at playeren kan bevæge sig rund på gulvet.

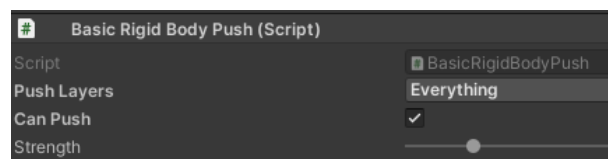
Tilføj nogle vægge og lav en lille bane. Det kan være en fordel at lave en *Prefab* af et stykke væg og så kopiere det.

Når man skal placere væggene kan det være en fordel at bruge Grid systemet. Grid system sættes op i **Scene** view med følgende knapper:



Man får grid'et til at 'snappe' ved at holde Ctrl tasten nede mens man flytter et objekt.

Tilføj også nogle kasser og få Playeren til at kunne skubbe til dem. Det sættes op i *PlayerCapsule*:



Får kasserne til at lave en skrabe lyd når de bliver skubbet hen over jorden.

Flyt kasser og brug dem til at hoppe op på. Tilføj dem til banen hvor det udnyttes.

## 2.1 Ting man kan tage

Vi skal nu lave nogle ting man kan tage for at kunne samle nogle point. Til det formål skal vi bruge nogle ting at tage og her kan man for eksempel bruge følgende fra Asset Store: *Simple Gems Ultimate Animated Customizable Pack*

Sæt en ting ind i **Scenen** og gør så man kan tage den (lav et script til det).

Tilføj en *Score* tekst og opdater den når man tager en ting.

Tilføj en lyd når man tager en ting. Lyde kan man fx finde her: <https://www.fesliyanstudios.com/>

Indsæt ting forskellige steder man kan finde og tage. Sæt dem lidt svære steder at komme hen til, så det begynder at ligne et spil.

Her kan man vælge at udbygge sin bane og lave et lille spil. De næste skridt bliver ikke lettere ...

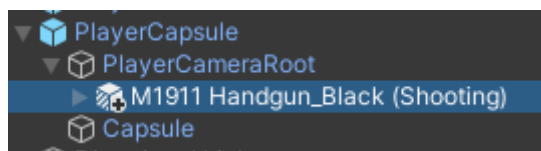
## 3 Basis features

De følgende features er ikke afhængig af hinanden og kan derfor laves i anden rækkefølge end vist. Dog vil nogle af forklaringerne så mangle ...

### 3.1 Pistol og skud

Vi skal nu have vores Player til at kunne skyde med en pistol. Importer følgende asset: *Modern Guns: Handgun*.

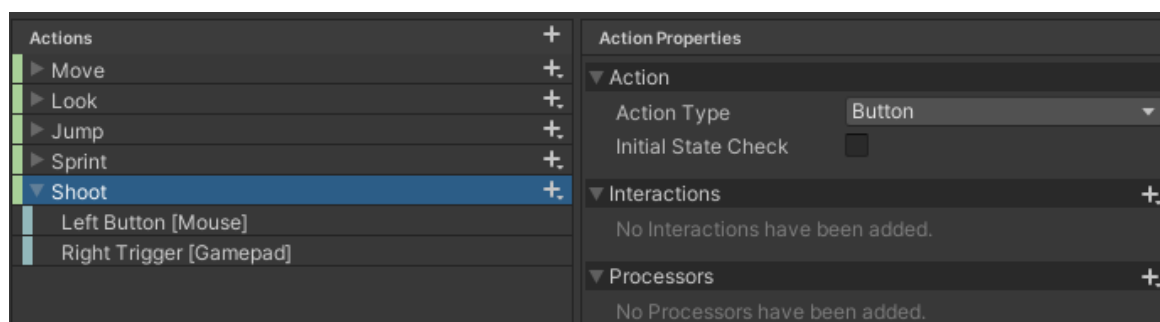
Træk en af pistolerne (*PreFab*) ind i Hierarchy'et og placer den her:



Den skal ligge her for at følge person og kamera rigtigt. Placer den i viewet så det ser godt ud. Hvis pistolen skal kunne skyde, skal man vælge den her hedder (*Shooting*) i navnet,

Når man tilføjer pistolen som kan skyde (*Shooting*) så får man en fejl, da den bruger det gamle input system. Det skal derfor laves om til at bruge det nye input system!

Det nye input system er defineret her: *StarterAssets/InputSystem/StarterAssets.inputactions*. Åben det og tilføj en *Shoot* action.



I scriptfilen som ligger lige ved siden af skal man tilføje følgende linjer:

*StarterAssetsInputs*:

```

public bool shoot;

public void OnShoot(InputValue value)
{
    ShootInput(value.isPressed);
}

public void ShootInput(bool newShootState)
{
    shoot = newShootState;
}

```

I filen *SimpleShoot* skal man tilføje følgende linjer (den ligger under *Nokobot/.../\_Demo Assets/*):

```

using StarterAssets;
using UnityEngine.InputSystem;

StarterAssetsInputs starterAssetsInputs;

starterAssetsInputs = FindObjectOfType<StarterAssetsInputs>();

if (starterAssetsInputs.shoot)
{
    gunAnimator.SetTrigger("Fire"); // Denne linje er der allerede!
    starterAssetsInputs.shoot = false;
}

```

Pistolen skal gerne kunne skyde nu. Prøv at skyde på en kasse og se hvad der sker!  
Tilføj en lyd når pistolen skyder.

Hvis man gerne vil kunne "finde" pistolen i spillet og så tage den, så skal den disables i **Hierarchy**'et og når man så tager en 'kopi' af pistolen så enables den. Det med at enable pistolen kan laves i det script som man bruger til at tage ting med. (Det er vigtigt at den kopi man lægger ind er den som IKKE kan skyde!)

Tilføj et script til kuglen, så når den rammer noget, så forsvinder det. Scriptet skal også sørge for at kuglen forsvinder efter noget tid, hvis den ikke rammer noget. (Det er selvfølgelig ikke alt den rammer som skal forsvinde, fx vægge)

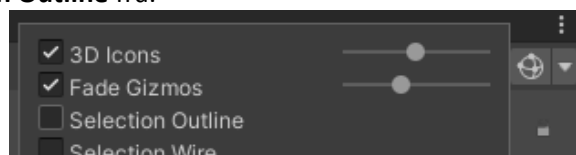
Kuglen flyver hurtigt og det er derfor ikke sikkert at den altid rammer noget! (Tænk over hvad problemet er?) Det kan udbedres ved at tilpasse nogle forskellige parametre under *Rigidbody* for kuglen (husk at kuglen skal rettes under *PreFabs*).



Man kan også med fordel gøre *collideren* noget længere, så det er mere sandsynligt at den rammer.

Tilføj en partikeleffekt når et skud rammer vægge og kasser.

Start med at oprette et **Effects->Particle System** og få det til at se godt ud. Det kræver lidt forsøg. Det kan være en fordel at slå **Selection Outline** fra:

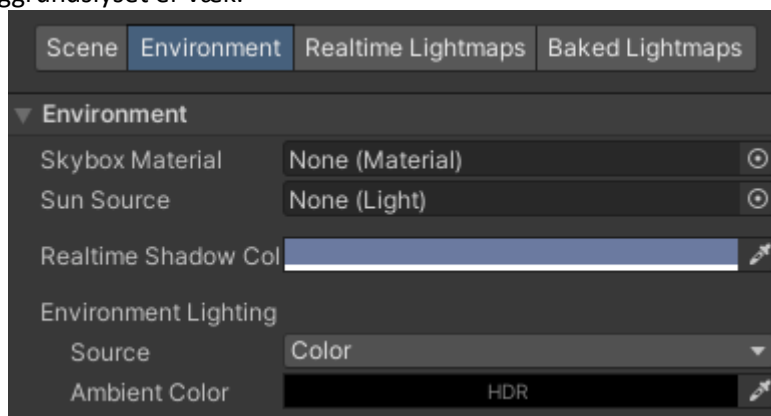


Lav en *PreFab* og **Instantiate** så *PreFab'en* når man rammer noget. Hust at fjerne den igen efter lidt tid.

## 3.2 Lys og lygte

Nu kommer vi ind i de mørke sider af Unity, for det med at lave rum uden lys er lidt magisk! Unity har per default lys alle steder – på den måde er det meget lettere at se hvad der foregår. Det har den konsekvens at selv om man laver et helt lukket rum, så er der stadig lys inde i det! Det er lidt irriterende hvis man gerne vil lave et mørkt rum, hvor man skal bruge en lygte for at finde vej.

Først skal vi have slukket baggrundslyset, og det gøres i menuen: **Window->Rendering->Lighting**. Det skal se sådan ud, for at al baggrundslyset er væk:



**Skybox Material:** None

**Sun Source:** None

**Source:** Color

**Ambient Color:** Black

Luk vinduet. Hvis scenen er for mørk nu, så skal det løses ved at ændre på *Directional Light* i **Hierarchy**'et eller tilføje nogle flere af dem.

Lav et 'lukket' rum (som vil kræve en lommelygte), der må gerne være en lille indgang, fx en tunnel. Stil Playeren i rummet, så man kan se om det er mørkt nok.

Lav et nyt materiale som skal bruges til mørke rum. Her skal **Emission** fjernes, da farven ellers lyser lidt. Lyset kan godt trænge igennem samlingerne i væggene, selvom de overlapper! Det kan 'løses' på flere måder:

- Flyt rundt på *Directional Light* kilden, ja det lyder mærkeligt!
- Lav dobbelt vægge!

Det kræver typisk lidt forsøg at få rummet helt mørk, men det kan godt lade sig gøre.

Importer en lommelygte fra Asset store: *Rusty Flashlight*.

Indsæt lommelygten (*PreFab*) på samme måde som pistolen, så det ser godt ud. Det skal helst se godt ud både med og uden pistol (hvis man ønsker at bruge begge).

Lommelygten bruger ligesom pistolen det gamle input system. Det skal fikset på samme måde som for pistolen, men denne gang skal man bruge en tast på tastaturet til at tænde og slukke lygten. Lygtens script hedder: *FlashlightToggle*.

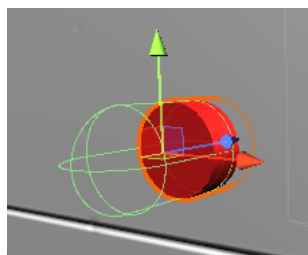
Tag lommelygten ligesom for pistolen. (Husk kun at bruge billedet af lommelygten og ikke *PreFab*'en!).

## 3.3 Knap og døre

Vi skal nu lave en knap, som vi kan interagere med, som derefter kan udføre en aktion som fx at åbne en dør.

Lav en knap af en cylinder og giv den en farve ved at lave et materiale.

Udvid *collideren* så den stikker så langt ud fra knappen som man ønsker for at kunne trykke på den. (Vi har ikke nogle arme, så man trykker på knappen ved at komme tæt nok på den).



Lav et script som får knappen til at virke, ved fx at disable et objekt (dør). Den kan også starte en animation, "tænde" for en bro mv.

Tilføj en lyd til knappen, så man kan høre når man trykker på den.

Husk at lave en *PreFab* når den virker.

## 4 Fjender og health system

### 4.1 Kanon som skyder (kan springes over)

Grunden til at vi starter med en kanon er, at det er lettere at lave end de fjender som vi laver senere.

Nu skal vi lave nogle 'fjender' som kan angribe os. Planen her er at lave en kanon som kan skyde på Player.

Den her kanon er fuldt animeret og kan selv dreje så den skyder i den rigtige retning. (Den skyder dog ikke med noget). Kanonen er lidt voldsom, men den ser kanon godt ud.

Importer kanonen fra Asset: *Free Sci Fi Gatling Gun - Tower Defense*

Det giver en error fordi der mangler et script. Åben *PreFab*'en og fjern referencen til det script i toppen som ikke findes!

Ret i scriptet *GatlingGun*, så den skyder på *Player* og ikke *Enemy*.

Sæt den ind i **Scenen** og bevæg Player i nærheden af kanonen og se at den skyder i den rigtige retning.

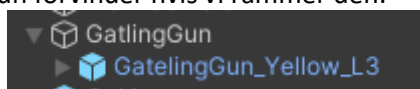
Bonus: Prøv at finde ud af hvordan effekten med patronhylstrene er lavet.

Tilføj en god lyd til kanonen når den skyder. Hint: Her skal man nok finde en god lyd og så tænde og slukke for den i stedet for at prøve at lave en separat lyd for hvert skud.

Hvis man skyder på kanonen så forsvinder den – hvilket i starten kan være ok. Problemet er at selv om man rammer meget ved siden af, så forsvinder den stadig. Det er fordi dens collider er meget stor, da den bruges til at aktivere kanonen. Vi skal derfor have lavet endnu en collider som skal bruges når vi skyder på kanonen. De skal have hvert sit tag, for at vores skud kan kende forskel på dem.

Giv kanonen et tag og søg for at vores skud ikke får den til at forsvinde.

Lave et nyt tomt objekt og læg så kanonen ind under det og giv det nye objekt en collider som passer til kanonens størrelse. Se at kanonen kun forsvinder hvis vi rammer den.



Lige nu så skyder kanonen ikke med noget og derfor rammer den heller ikke Player.

Sørg for at den reelt skyder kugler og kan ramme Player. Kik på koden fra *SimpleShoot* og find den linje som laver kuglerne og kopier den sammen med de linjer som linjen er afhængig af.

Lav en ny kugle og et nyt script ala *BulletCtrl* og ret det til, så det rammer Player. Se at det virker. (Hvis der kommer 'for mange' kugler pr sekund, så kan man lave et hold down system.)

Hvad der reelt skal ske når man bliver ramt er om til dig. Man kan dø med det samme eller man kan lave et helt health system (se 4.2).

Det er ikke realistisk at vores pistol kan skyde kanonen. Tilføj en olietønde ved siden af kanonen og brug den til at 'dræbe' kanonen, enten ved at vi skyder tønden, eller at kanonen 'kommer' til det selv. Selve tønde dræber mekanismen kan laves lidt ligesom med en kontakt. Når tønden 'dør' så disables den et objekt (kanonen). Det er vigtigt med en god effekt når det sker.

## 4.2 Player health system

Vi skal nu lave et simpelt health system, som kan udbygges til noget mere kompleks senere hvis man har lyst.

Lav først et tekstfelt på skærmen til at vise ens health.

Lav et script til Player health håndtering, som opdaterer tekst feltet.

Lav også en `public void Hit(float impact)` funktion i dette script som kan kaldes når Player rammes af 'noget' - skal kaldes fra *GatlingBullet* script på følgende måde.

```
collision.gameObject.GetComponent<PlayerHealth>().Hit(2);
```

Her hedder scriptet *PlayerHealth* og 2 er hit impact for Gatling kanonen.

Nu tæller health ned hver gang man bliver ramt af Gatling kanonen. Når health rammer 0, så er man død.

Her kan man fx vælge at skrive Game Over på skærmen og fjerne Player.

Tilsvarende kan man lave en funktion til at tæller health op når man tager bestemte ting. Funktion skal selvfølgelig sikre at health ikke kan komme over 100%, eller hvilket niveau man nu vælger.

Health kan også udvides med et skjold, som så indgår i `Hit()` funktionen, hvor 'formelen' bliver lidt mere kompleks.

Man kan i stedet for et health tal lave en health bar.

Tilsvarende kan man lave et health system til fjenderne, så de også skal rammes flere gange før de dør/forsvinder (se senere).

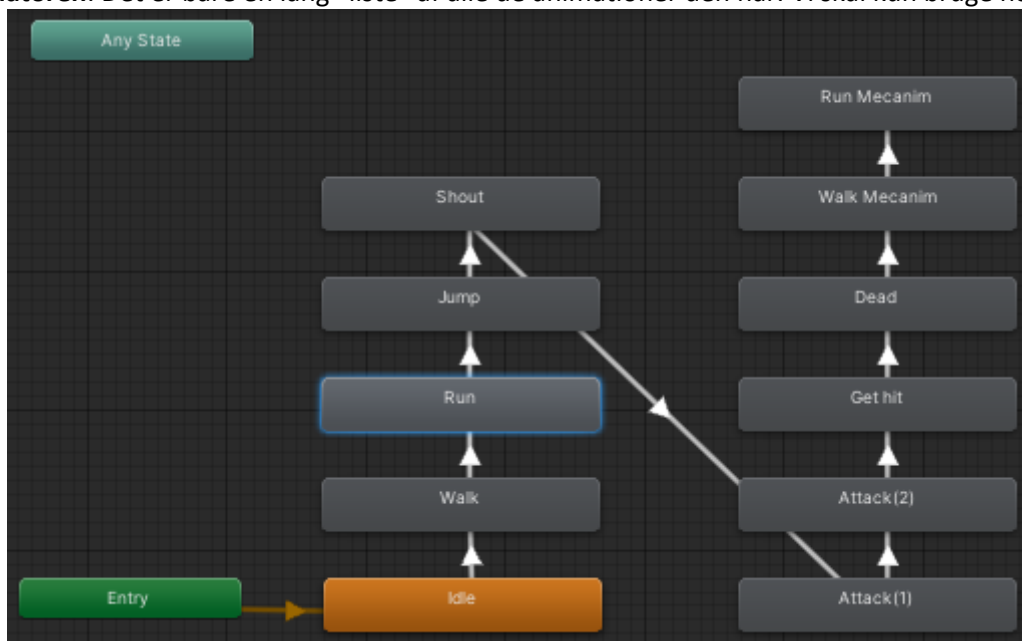
## 4.3 Fjender

Vi skal nu have nogle 'rigtige' fjender. Hent Asset: *Character Monsters X*.

Indsæt Prefab af Monster i **Scenen**.

Den er meget stor, så ret scale til 0.05.

Åben **Animatoren**: Det er bare en lang "liste" af alle de animationer den har. Vi skal kun bruge nogle af dem.



1. Start med at slette **Jump**, **Shout**, **Attack(2)**, **Walk\_Mec** og **Run\_Mec**.
2. For både **Idle**, **Walk** og **Run** set **Loop Time**. (Husk at trykke på **Apply**)
3. Tilføj en **bool** parameter som hedder *Walk*.
4. Lav en transition fra **Walk** til **Idle** og brug *Walk* parameteren på de to pile mellem **Walk** og **Idle**.


Lad bare resten af statene være (vi skal bruge dem senere)

### 4.3.1 NavMesh

For at fjenderne kan finde rundt i vores spil, skal vi bruge en Unity feature som hedder NavMesh. NavMesh laver et kort over alle de veje som vores fjender kan bevæge sig på.



Det er lettest at alle vores 'faste' strukturer som bygninger, kasser mv er under det samme objekt, så lav et tomt gameobjekt i **Hierarchy**'et og put alle disse objekter ind under dette.

Vælg dette objekt og i **Inspector**'en og klik på **Static**  og tryk **Include Children**. Vi har nu fortalt NavMesh at alle disse objekter ikke kan flytte sig og at den derfor skal navigere uden om dem.

Åben menuen: **Window->AI->Navigation**. Her kan man konfigurere NavMesh så det passer til ens scene. Til at starte med vælg **Bake** og tryk derefter på **Bake** nede i højre hjørne (bagefter har alle veje har fået en blå tint).

Nu har NavMesh overblik over hvor alle veje (paths) i gamet er. NavMesh skal opdateres hvis der tilføjes flere strukturer til spillet, da vejene så skal ændres.

Hvis man har en dør som kan åbnes og lukkes, eller en kasse som kan flyttes, så skal den tilføjes på en speciel måde, da den ikke er statisk. Vælg døren/kassen og tilføj komponenten *Nav Mesh Obstacle* og set **Carve**. (Det er vigtigt at døren/kassen ikke allerede er med i NavMesh – det vil sige at **Static** ikke er sat.)

### 4.3.2 Følg efter Player

Vi skal nu have Monsteret til at følge efter Player.

Vælg Monster og tilføj komponenten **Nav Mesh Agent**

Lav et script til at få monsteret til at følge efter Player. Brug følgende linjer til det:

```
using UnityEngine.AI;

GameObject player;
NavMeshAgent agent;

agent = GetComponent<NavMeshAgent>();
player = GameObject.Find("PlayerCapsule");

agent.SetDestination(player.transform.position);
```

I dette eksempel hedder Player *PlayerCapsule*.

For at få benene til at gå når Monsteret går skal man kalde animatoren.

```
ani.SetBool("Walk", true);
```

Monsteret går, så længe det er væk fra Player. Brug disse to metoder til se hvor langt vi er fra Player, og dermed om benene skal bevæge sig:

```
agent.remainingDistance // Afstand til Player
agent.stoppingDistance // Afstand defineret i Inspector'en
```

Man styrer den hastighed monsteret går i **Inspector:Nav Mesh Agent:Speed**. Få benene til at følge hastigheden så det ser naturligt ud.

### 4.3.3 Fjende skade

Vi skal nu kunne skyde Monsteret. Tilføj en collider som dækker Monsteret nogenlunde.

Test at vi nu kan skyde Monsteret.

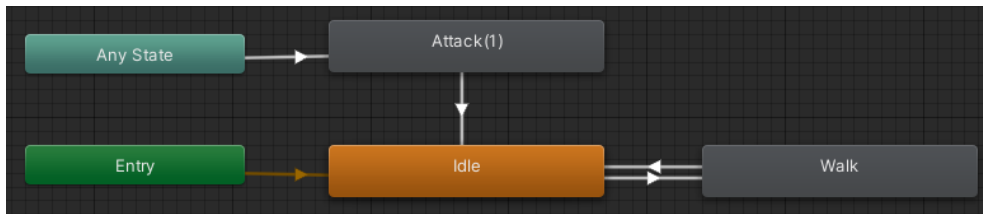
Monsteret skal også kunne skade os. Hvis det er tæt nok på os, så tager vi skade hver gang den angriber os. Tilføj nu noget kode, som gør skade på Player hver gang monsteret er for tæt på. Brug samme teknik som for *Gatling* kanonens kugler til at lave et `Hit()`.

Hvis man 'bare' kalder `Hit()` hele tiden når Monsteret er tæt på, så dør Player meget hurtigt.

Så når Monsteret er tæt på, så skal den lave et attack og kalde `Hit()` og så vente lidt, inden det laver et nyt attack.

For at lave et attack, så skal man lave endnu en **Transition** inde i **Animator**'en.

1. Lav en trigger parameter og kald den *Attack*.
2. Lav en **Transition** fra **Any State** til **Attack(1)** og tilføj *Attack* triggeren til denne pil.
3. Lav også en **Transition** fra **Attack(1)** til **Idle**.



Koden i `Update()` kan nu se sådan ud (lav selv det initialiseringskode der mangler):

```

timer += Time.deltaTime;
agent.SetDestination(player.transform.position);
if (agent.pathPending)
    return;
if (agent.remainingDistance > agent.stoppingDistance)
    ani.SetBool("Walk", true);
else
    ani.SetBool("Walk", false);

if ((agent.remainingDistance < 3) && (timer > 1.5f))
{
    ani.SetTrigger("Attack");
    player.gameObject.GetComponent<PlayerHealth>().Hit(20);
    timer = 0;
}
  
```

Lav en **Prefab** af Monsteret og indsæt den forskellige steder i spillet. Man kan også lave en Enemy spawner.

## 4.4 Fjende health system

Ind til nu har alle vores fjender kun kunne klare et hit før de dør. Vi skal nu have lavet et health system for vores fjender på samme måde som for vores Player.

Ændre koden så det nu kræver flere skud at dræbe et Monster. Det kan være en fordel at tilføje en hit divider, som kan sættes afhængig af fjenden.

```

health = health - impact/hitDiv;
  
```

Når Monsteret bliver ramt, skal det lave en **Get Hit** animation og når det dør en **Dead** animation og derefter 'forsvinde'. Opdater nu **Animator**'en og koden.

Lav fjender med forskellige *hitDiv* parameter, så nogle er svære at dræbe end andre.

## 4.5 Fjende patruljering

Lige nu forfølger fjenderne Player hele tiden. I større spil er det bedre at de patruljerer et område og så forfølger Player hvis man kommer for tæt på. Det kan laves på flere måder, fx som en liste af waypoints som bliver fulgt i rækkefølge eller en list af waypoints, hvor man så går hen til et tilfældigt waypoint fra listen. Den sidste metode er lettere at bruge i spillet, da det reelt kan klares med den samme liste til alle Monstrene.

Listen kan laves som et objekt, med en liste af waypoints under. På den måde kan man have flere forskellige waypointlister at vælge imellem. (Det er vigtigt at waypoints'ne rører ved "jorden", da NavMesh ellers ikke kan finde en rute.) Når Monstret er tæt nok på Player så skal den følge efter Player i stedet for at patruljere.

Lav nu en liste af waypoints og noget kode som gør at Monstrene følger waypoints'en og følger efter Player når de er tæt nok på. Det vil være smart hvis man kan vælge mellem *random* og *round robin* og den afstand som skal være til Player inden de begynder at følge Player. På den måde kan man lettere konfigurere sine fjender.

Man får brug for følgende funktioner:

```
waypoints.GetChild(index)
waypoints.childCount
Vector3.Distance(pos1, pos2)
```

Hints: Tag den kode som nu følger Player og put ind i en funktion: `Follow()`. Lav så en ny funktion som skal bruges til at lave selve Patruljeringen: `Patrol()`. `Patrol()` funktionen kræver at man laver en lille statemaskine. I `Update()` skal man så kalde den 'rigtige' funktion (*Follow/Patrol*) afhængig af hvor langt væk Player er. Det kan også være en god idé at lave en funktion til at finde næste waypoint, afhængig af *random* og *round robin*.

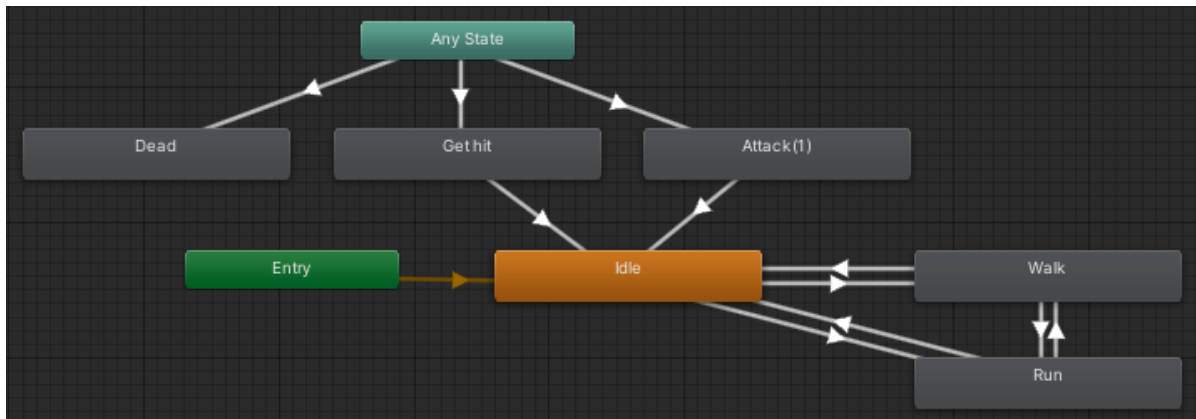
## 4.6 Hurtige fjender

Lav nu nogle fjender som går rundt og patruljerer, men løber når de får øje på Player.

- Man skal bruge **Run** animationen.
- Man skal ændre hastigheden de bevæger sig alt afhængig af hvad de laver.
- Det kan laves med en ekstra parameter som afgør om de kan løbe, så man har flere forskellige fjender.

Det vil være smart hvis man kan genkende disse hurtige fjender, fx ved at vælge en anden farve.

Hint: Animatoren ender med at se sådan ud:



## 5 Ideer

Udvid banen med flere rum, trapper, knapper mv. Brug eventuelt ProBuilder (plugin) og Terrain

Begrænset antal skud som pistolen har og læg nogle magasiner rundt om i spillet

Man har vundet når alle fjender er døde, eller et andet kriterie er opfyldt.

Hvis man begynder at have mange items at holde styr på, kan det være en fordel at lave et Game Master objekt – se *tower\_defense\_guiden* om hvordan man gør det.

Fjende spawner

Fjende healthbar

Døde fjender giver point

Flere våben (Man vælger mellem våbnene ved at enable og disable et våben i **Hierarchy**, via scriptet – som når man tager pistolen)