Implementing melee based skills for a melee combat game in Unity

Door Niels Poelder
Klas G&I4
3028718

Introductie over mij

- Ik ben Niels Poelder
- 24 jaar oud
- Studeer aan het HKU als Game Developer

Voor wie is deze tutorial?

- Deze tutorial is voor Game Developers.
- Game Designers zijn als zij enige kennis hebben van programmeren.

Welke voorkennis moet je hebben?

- Basis ervaring met Unity (Je weet hoe je game objects moet aanmaken, de editor gebruikt etc).
- Enigszins ervaring met C# Programmeren aangezien wij hier gebruik van gaan maken.
- Ervaring hebt met Unity’s nieuwe Input Systeem.

Wat zijn skills?

Skills zijn speciale acties die de speler kan uitvoeren. Elke game heeft zijn eigen interpretatie ervan wat een Skill is maar in de meeste gevallen zijn dit bijvoorbeeld krachtige speciale aanvallen en/of moves die de speler moet unlocken die niet te vaak kunnen worden gespamd.

Wat ga ik in je in deze tutorial leren?

In deze tutorial ga je leren hoe je je eigen modulaire skill systeem implementeert in Unity met behulp van Unity's nieuwe Input Systeem (Dit zorgt ervoor dat het extra optimaal werkt en kan je het ook meteen gereed maken voor controller support).

Verder implementeren wij ook melee schade aan een vijand m.b.v. Animation Events, en een Ragdoll death implementeert voor de AI. De tutorial Bestaat uit 4 delen.

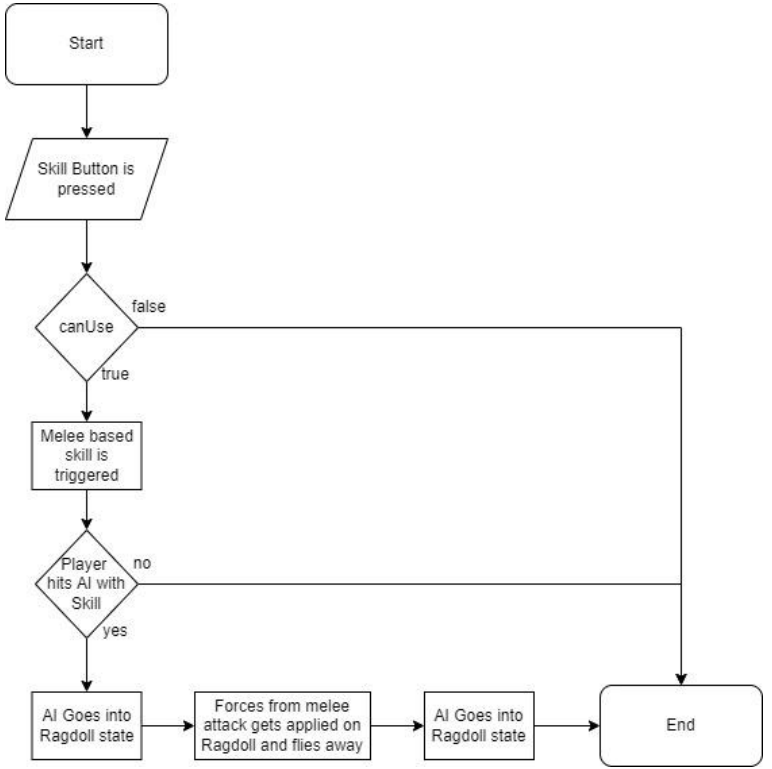
- Deel 1 programmeren wij het skill systeem.
- Deel 2 Implementeren wij deze skills met de juiste parameters zodat het goed werkt.
- Deel 3 programmeren en implementeren wij de Damage Checker van de character en Ragdoll physics voor de AI.
- Deel 4 voegen wij alles samen en testen wij de uitkomst.

Hoe werkt het systeem?

Het systeem werkt als volgt. De speler drukt op een zelf gedefinieerde knop, deze toets, er wordt gekeken of de melee based skill kan worden geactiveerd.

Als het kan dan voert het character de skill uit. D.m.v een collider op het wapen van de speler wordt er gekeken of de AI geraakt is.

Zo ja? Dan wordt de AI omgezet in een Ragdoll staat en worden er hier krachten op uitgevoerd.



Flowchart ter illustratie

De ontwerpkeuzes van dit systeem.

Ik heb gekozen om gebruik te maken van het nieuwe Unity InputSysteem omdat je daarmee de functies d.m.v. events kan aanroepen. Hierdoor wordt hij niet constant in de **Update()** loop uitgevoerd. Daaruit roep ik de skills dus aan d.m.v. de gekozen toets voor die specifieke skill. Dit zorgt er voor dat het systeem super optimaal werkt.

Wat betreft het kijken of de speler überhaupt schade toe doet heb ik er voor gekozen om een voorbeeld uit de industriestandaard te nemen waarbij er d.m.v. colliders wordt gekeken of er iets wordt geraakt. Daarmee heb ik mijn eigen systeem bedacht d.m.v. Animatie Key Events zodat ook in dit geval het maar wordt uitgevoerd wanneer het nodig is. Hierdoor is het uiterst optimaal.

Dan een van de moeilijkere dingen een Ragdoll implementeren met een animatie. Ik heb gekeken naar verschillende oplossingen en heb er uiteindelijk een systeem bedacht waarin je de Navmeshagent en Animatie van de AI kan combineren met de ragdoll d.m.v. het aan en uitzetten van deze specifieke componenten. Als je dus de NavmeshAgent en Animatie uitzet, en dan de Ragdoll aanzet, behoudt hij de positie en huidige animatie pose dus verandert hij niet in een T-Pose, dit is dus veel realistischer. Als deze dan uitstaan en de Ragdoll aanstaat, voer ik hier de krachten van het wapen van de speler op uit met het Damage Script.

Correlatie aan mijn endterm-project

Mijn project Rootlash gebruikt een skill systeem voor het character waarmee hij speciale vaardigheden kan uitvoeren zoals een Wortelstomp of genezen.

Een voorbeeld van Skills

Voordat wij beginnen heb ik voor jullie hier een voorbeeld van wat we dus uiteindelijk gaan maken.

Dit is de “RootAttack” skill in mijn end-term project Rootlash waarbij de wortels uit de grond schieten met VFX.

Videolink voor als de GIF niet werkt

Deel 1: Het skill systeem maken

Dan komen we nu aan bij het programmeren van het Skill Systeem.

We gaan hem in OOP Programmeren waardoor het mooi modulair wordt. Hierdoor kan je super snel nieuwe Skills maken en toevoegen in het spel.

We beginnen dus met het maken van de baseclass en daaruit erven wij de subclass skills.

Deel 1: Het skill systeem maken

We beginnen dus met het maken van de baseclass. Om dit te ontwerpen vragen wij ons af welke generieke eigenschappen er bij een skill horen?

- Een naam.
- Een skill heeft een cooldown time.
- Je kijkt of je de skill kan gebruiken.

Nu we deze hebben vastgesteld kunnen we gaan programmeren.

Deel 1: Het skill systeem maken

We maken dus de class aan genaamd “SkillBase” en wat belangrijk is, is dat we hem **Abstract** maken.

Dan voegen we onze waardes toe zoals de naam, cooldowntime, en kijken of hij gebruikt kan worden d.m.v. “Can use”.

Verder is het belangrijk om alvast ook Unity’s nieuwe Input systeem te importeren d.m.v. dit in de code te zetten “**using UnityEngine.InputSystem;**”

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.InputSystem;

    public abstract class SkillBase : MonoBehaviour
    {
        public string name;
        public float cooldownTime = 10f;
        public bool canUse = true;
    }
```

Deel 1: Het skill systeem maken

Dan gaan we nu kijken naar welke functies we nodig hebben.

- We hebben de Activatie knop functie nodig voor de skill.
- Een functie waarin de Skill zelf wordt uitgevoerd.
- En dan nog een functie die de Cooldown start.

Deel 1: Het skill systeem maken

We gaan deze functies stap voor stap implementeren. Eerst simpel maken we de functie aan waarin we onze code gaan maken voor elke skill. Deze code kan verschillen maar de functie zal zelf altijd dezelfde naam behouden. Dus maken we een “Skill()” functie aan. We maken hem **Abstract** zodat we deze kunnen gebruiken in de Child classes.

```
public abstract void Skill();
```

Deel 1: Het skill systeem maken

Dan hebben we nu de functie nodig waarmee wij de skill gaan activeren. Hiervoor moeten we zorgen dat we een callback actie maken naar het nieuwe Input systeem van Unity. We gaan de functie noemen “TriggerSkill”.

Deze functie gaat dan de Skill aanroepen, en de Cooldown die we hierna gaan implementeren.

We maken er een virtual void van zodat we hem kunnen aanpassen in de child classes indien nodig, maar in de meeste gevallen hoeft dit niet.

```
public virtual void
TriggerSkill(InputAction.CallbackContext context)
{

}
```

Deel 1: Het skill systeem maken

In deze functie voegen wij nog een If statement toe waarbij we d.m.v. de CallbackContext checken of de functie wordt aangeroepen. En of de speler daadwerkelijk de Skill al kan gebruiken d.m.v. de “canUse” boolean.

Nu is de functie af en gaan we de cooldown Implementeren.

```
public virtual void
TriggerSkill(InputAction.CallbackContext context)
{
    if(context.performed && canUse)
    {
        Skill();
        StartCooldown();
    }
}
```

Deel 1: Het skill systeem maken

Dan maken we nu de StartCooldown() functie aan. We zorgen ervoor dat het een protected functie is zodat hij alleen binnen de class en subclasses kan worden aangesproken.

```
protected void StartCooldown()
{

}
```

Deel 1: De Skills Implementeren

Dan maken wij een IEnumeraor aan genaamd “Cooldown” en returnen we een WaitForSeconds met de cooldown time. Daar zetten we de canUse boolean aan en uit gebaseerd op de cooldownTime.

Dan in de StartCooldown() functie roepen wij deze IENumerator aan.

De StartCooldown() Functie is nu af.

```
protected void StartCooldown()
{
  StartCoroutine(Cooldown());
}

protected IEnumerator Cooldown()
{
canUse = false;
yield return new WaitForSeconds(cooldownTime);
canUse = true;
}
```

Deel 1: Het skill systeem maken

En dan is hier het volledige resultaat van de SkillBase class.

Dan gaan we nu voor een voorbeeld een subclass maken en een skill implementeren om kennis te maken met hoe je dit systeem gebruikt.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.InputSystem;

    public abstract class SkillBase : MonoBehaviour
    {
        public string name;
        public float cooldownTime = 10f;
        public bool canUse = true;

        public abstract void Skill();

        public virtual void TriggerSkill(InputAction.CallbackContext context)
        {
            if (context.performed && canUse)
            {
                Skill();
                StartCooldown();
            }
        }

        protected void StartCooldown()
        {
            StartCoroutine(Cooldown());
        }

        protected IEnumerator Cooldown()
        {
             canUse = false;
             yield return new WaitForSeconds(cooldownTime);
             canUse = true;
        }
    }
}
```

Deel 2: Het implementeren van de skills

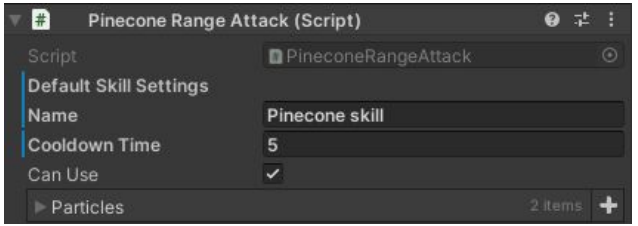
Nu de Baseclass af is, kun je heel simpel nieuwe Skills aanmaken. Door simpelweg een nieuwe class aan te maken, en dan te inherriten van de SkillBase class, en dan daarin de Skill() functie aan te passen en te overriden. Zie voorbeeld hier rechts.

In mijn geval hier rechts voeg ik een Animatie toe en particles die geactiveerd in de Code van de Skill() functie.

```
public class ForwardAttackSkill: SkillBase
    {
        [SerializeField] private Animator anim;
        [SerializeField] private ParticleSystem attackParticleEffects;

        private Player_Stats p_stats = default;

        private void Start()
        {
            anim = GetComponent<Animator>();
            p_stats = GetComponent<Player_Stats>();
        }

        public override void Skill()
        {
            anim.Play("Forward_Attack");
            attackParticleEffects.Play();
        }
    }
```

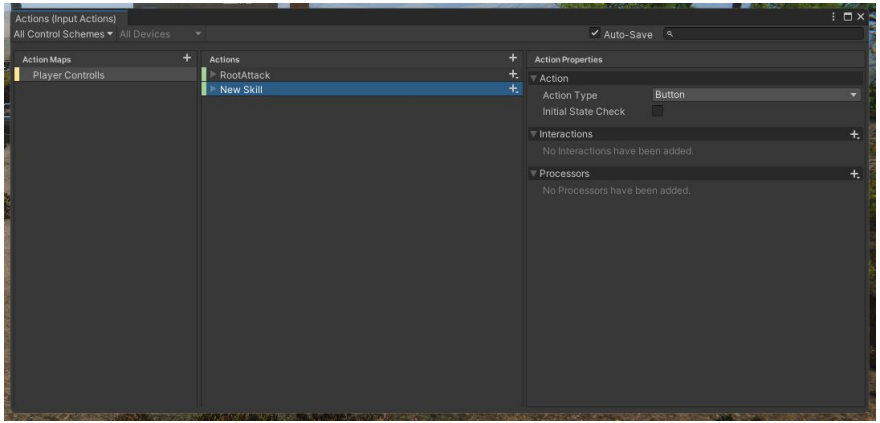
Deel 2: Het implementeren van de skills

Om dan de skill te implementeren in Unity sleep je het script in Unity op het object.



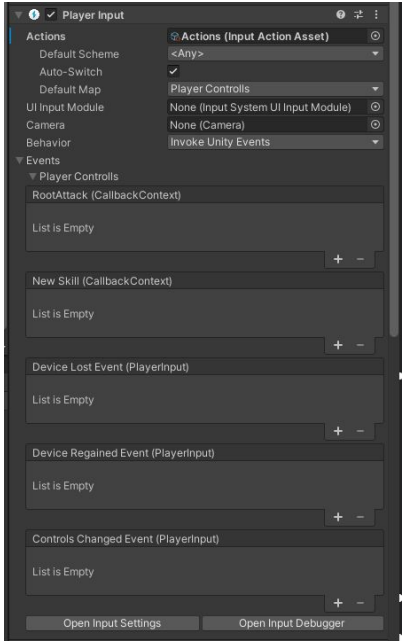
Deel 2: Het implementeren van de skills

Dan open je de InputActions game object en maak je een nieuwe key aan in de actie map. In mijn geval heet hij dus nu “New Skill”



Deel 2: Het implementeren van de skills

Open het PlayerInput script > Events en dan genoemde Action map. In mijn geval is dat “Player Controlls” maar bij jou kan dat wat anders zijn. En daar zie je de aangemaakte actie map genaamd “New Skill”

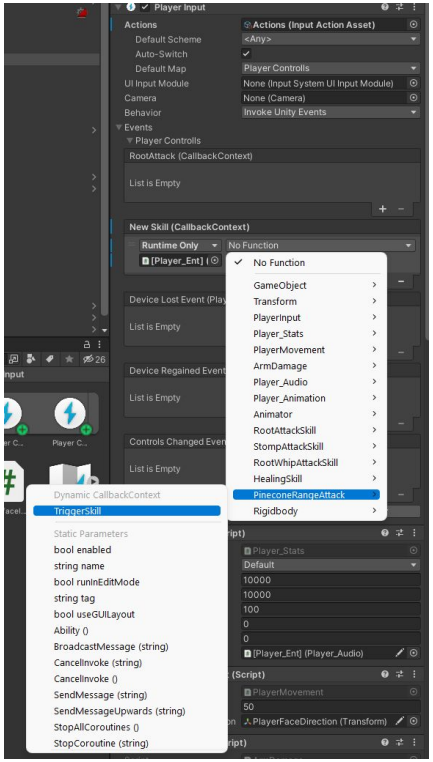


Deel 2: Het implementeren van de skills

Dan voeg je nu in de Editor de functie toe.

Let op **Zorg dat je de “Trigger SKill” functie aanklikt**.

Als je dit hebt gedaan ben je klaar voor gebruik en gaan we nu de SKill testen.



Deel 2: Het implementeren van de skills

En dan nu, als ik dan op de toets druk, dan wordt de skill zoals jij hem hebt geschreven geactiveerd. In mijn geval speelt er dus een animatie af.

Video link voor als de GIF niet werkt.

Deel 2: Het implementeren van de skills

Hoe verder? Je hebt nu alle bouwstenen om simpelweg verder skills te gaan ontwikkelen. Je hoeft alleen maar de Nieuwe class aan te maken, daarin Ability Class steeds aanpassen d.m.v. te overriden. En dan de nieuwe toets aan te maken in het Input Systeem. In unity te slepen, en het werkt!

Deel 3: Het schade systeem implementeren

Dan komen we nu aan bij het programmeren van het Schade systeem. Hierbij komen verschillende aspecten van het GameDev bij kijken.

In een melee based game wordt de schade meestal geactiveerd in een bepaald moment in de animatie. Dit heten ook wel Animation Events die tussen specifieke animatie keyframes kunnen worden uitgevoerd.

In ons systeem gaan wij ook een ragdoll toepassen waarbij als de speler de AI raakt, en AI deze sterft. Dan zal de AI een ragdoll worden en worden daar de krachten van de aanval op toegepast.

Hiervoor gaan wij de volgende scripts maken in de volgende delen:

- Deel 3.1: Damage Checker (enabled / disabled de collider die schade doet aan de AI)
- Deel 3.2: AI Physics (Als de AI dood is verandert hij in een ragdoll staat en worden de krachten van de Aanval hier op uitgeoefend)

Deel 3.1: Het maken en implementeren van de damage checker

Om te checken of wij daadwerkelijk schade hebben toegedaan aan de AI, kijken wij naar de Collider van het wapen dat de speler gebruikt, en gebruiken wij de **OnTriggerEnter** methode om te zien of hij een AI heeft geraakt en dan schaden toe doen.

Om er voor te zorgen dat deze collider niet altijd aan staat, zullen wij d.m.v. Animation Keyframes deze aan en uit zetten met.

Hiervoor gaan we een script maken genaamd **DamageChecker**.

Deel 3.1: Het maken en implementeren van de damage checker

Welke Parameters hebben wij nodig voor de DamageChecker? We hebben natuurlijk damage nodig voor schade. Daarna forceMultiplier waarmee je krachten op de ragdoll van de AI zullen uitvoeren. Daar bovenop natuurlijk de BoxCollider die we aan en uit gaan zetten d.m.v. de keyframes om te checken of we uberhaupt de vijand hebben geraakt. En daarna 2 vectoren (Previous weapon speed en current weapon speed) om te berekenen in welke richting de AI wordt geslagen en daar krachten op uit te voeren.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

    public class DamageChecker: MonoBehaviour
    {
        public float damage;
        public float forceMultiplier;
        public BoxCollider damageCollider;
        private Vector3 previousWeaponSpeed;
        private Vector3 currentWeaponSpeed;
    }
```

Deel 3.1: Het maken en implementeren van de damage checker

Dan komen we nu aan bij de functies die we nodig zullen hebben. We beginnen met de **Start()** functie. Hierin zullen we de damageCollider uit zetten zodat hij niet altijd aanstaat.

```
private void Start()
{
    damageCollider.enabled = false;
}
```

Deel 3.1: Het maken en implementeren van de damage checker

De volgende functie die we nodig hebben is de **Update()** functie. Daarin gaan wij checken hoe snel de arm zich momenteel beweegt om deze kracht op de AI Ragdoll toe te passen. Dit doen we met de **currentWeaponSpeed** door te kijken naar de positie van de **damageCollider**, en de positie van deze collider van de vorige frame (met **previousWeaponSpeed**). En deze van elkaar af te trekken. En te delen door de Time.DeltaTime om te zorgen dat er geen gekke getallen ontstaan door mogelijke framedrops.

```
private void Update()
{
currentWeaponSpeed=
(damageCollider.transform.position - previousWeaponSpeed)
/ Time.deltaTime;
previousWeaponSpeed=damageCollider.transform.position;
}
```

Deel 3.1: Het maken en implementeren van de damage checker

De volgende functie die we nodig hebben is de **OnTriggerEnter()** functie. Daarin kijken wij of het geraakte object het **AI_Physics** (wat wij straks nog gaan maken) script bevat. Zo ja dan voeren wij de **OnImpact()** functie uit en worden er krachten op de AI uitgevoerd.

```
private void OnTriggerEnter(Collider other)
{
    AI_Physics temp = other.GetComponent<AI_Physics>();
    if(temp != null)
    {
        temp.OnImpact(currentWeaponSpeed * forceMultiplier);
    }
}
```

Deel 3.1: Het maken en implementeren van de damage checker

De laatste functie die wij moeten toevoegen is de functie die wij aan aanroepen met de Animation Key Events genaamd **ToggleDamage()**.

Hiermee zetten wij de damageCollider aan en uit zodra de functie wordt geactiveerd. In de animatie keyframes kunnen wij deze aanroepen zodat hij op het juiste moment schade kan toedoen aan de AI.

```
private void ToggleDamage()
{
    damageCollider.enabled = !damageCollider.enabled;
}
```

Deel 3.1: Het maken en implementeren van de damage checker

Dan hebben wij hier nogmaals volledige het eindresultaat van het DamageChecker script met alle toegevoegde functies waarmee wij dus de Collider aan/uit zetten om te checken of we schade kunnen doen.

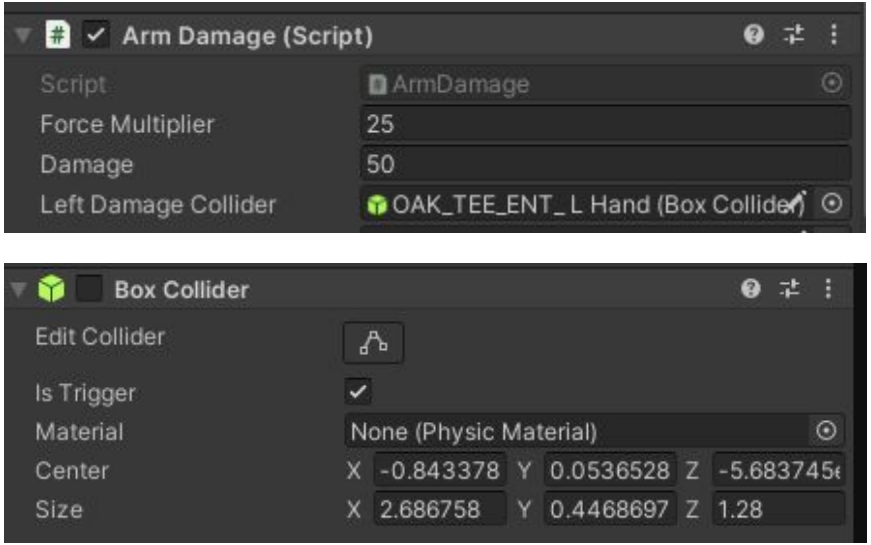
Zodra hij dan een AI object raakt dan worden hier op krachten uitgevoerd en zal de AI wegvliegen in zijn Ragdoll staat.

```
public class DamageChecker: MonoBehaviour
{
    public float damage;
    public float forceMultiplier;
    public BoxCollider damageCollider;
    private Vector3 previousWeaponSpeed;
    private Vector3 currentWeaponSpeed;
    private void Start()
    {
        damageCollider.enabled = false;
    }
    private void Update()
    {
        currentWeaponSpeed = (damageCollider.transform.position - previousWeaponSpeed) / Time.deltaTime;
        previousWeaponSpeed = damageCollider.transform.position;
    }
    private void OnTriggerEnter(Collider other)
    {
        AI_Physics temp = other.GetComponent<AI_Physics>();
        if(temp != null)
        {
            temp.OnImpact(currentWeaponSpeed * forceMultiplier);
        }
    }
    private void ToggleDamage()
    {
        damageCollider.enabled = !damageCollider.enabled;
    }
}
```

Deel 3.1: Het maken en implementeren van de damage checker

Dan gaan we nu de scripts en collider toevoegen. Let op, in mijn geval heet het DamageChecker script Arm Damage.

LET OP: Zorg dat je de collider plaatst op de plaats waar de speler schade gaat doen (dus bijv, Hand of Zwaard). En dat deze Collider voor de zekerheid uitstaat, en dat **IsTrigger** Enabled is op deze collider.

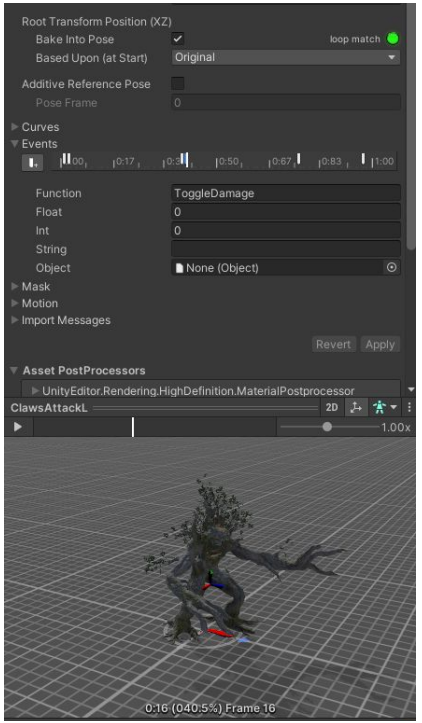


Deel 3.1: Het maken en implementeren van de damage checker

Tot slot, vergeet niet de animatie te openen in Unity, en dan daarin 2 Animatie Events te maken op de juiste Keyframes waarbij de speler begint met slaan, en stopt met slaan. Dit moet je zelf een beetje tweaken.

En als alles dan goed staat ingesteld dan gaat de Collider zelf automatisch aan en uit.

Dan zijn we nu klaar met het implementeren van de Damage Checker.



Deel 3.2: Het maken en implementeren van de AI Ragdoll

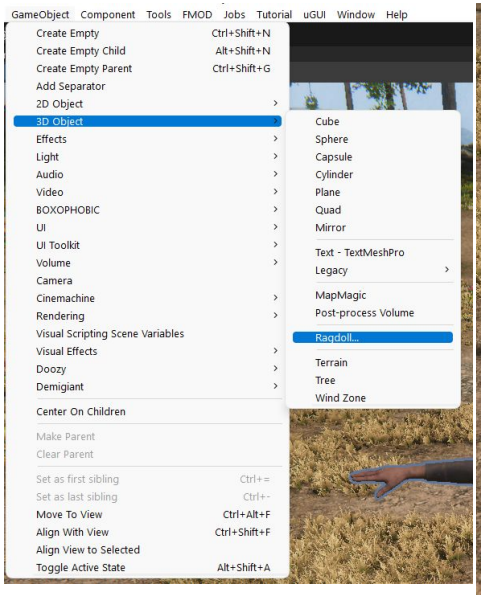
Tot slot komen wij aan bij het Ragdoll deel van de AI. Dit zorgt ervoor dat op het moment dat de AI wordt geraakt door de speler, dat hij wegvliegt door de krachten die de speler op de AI uitvoert. Om dit realistisch te maken gebruiken wij hiervoor een ragdoll.

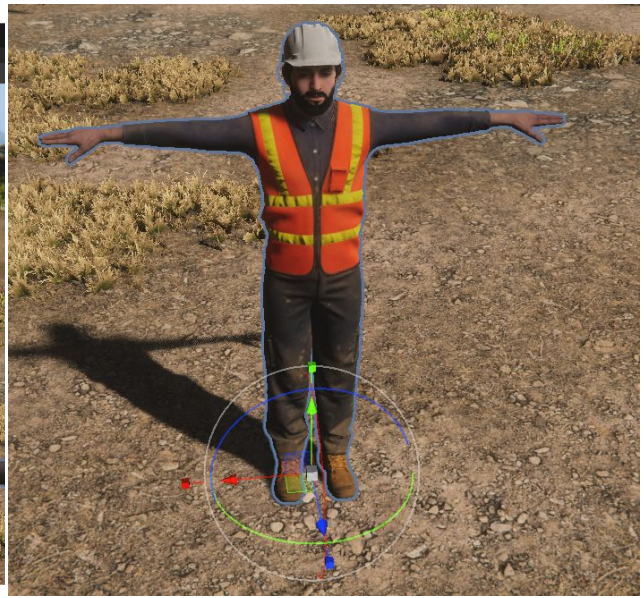
Hiervoor gaan wij enkele scripts aan en uit zetten, en dan in de staat dat zij uit staan hier krachten op uitvoeren. Laten we beginnen met het implementeren van de Ragdoll zelf.

Deel 3.2: Het maken en implementeren van de AI Ragdoll

Om de ragdoll te genereren is het belangrijk dat je een Menselijk character model hebt dat gerigged is.

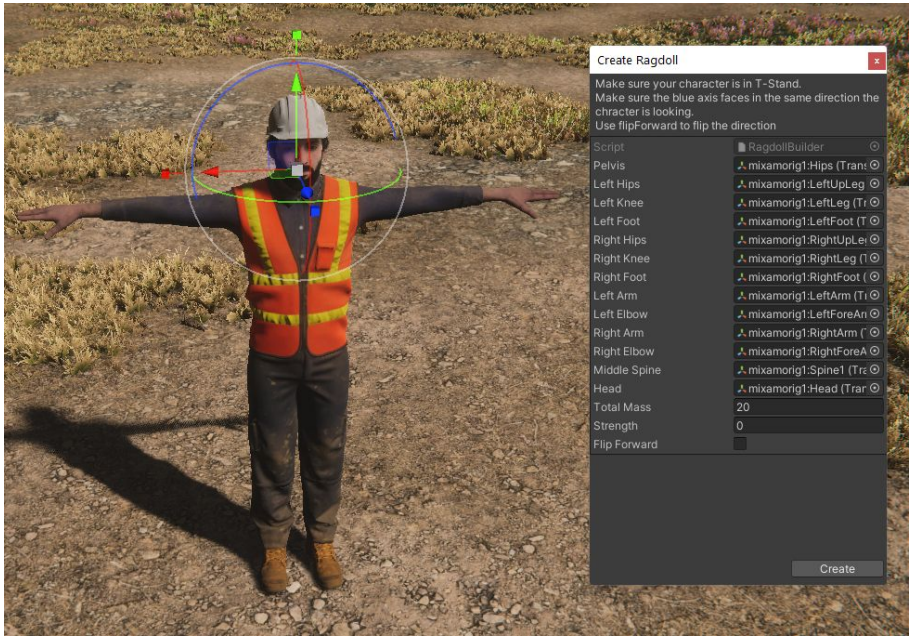
Klik op het character, ga naar 3D Object > Ragdoll om de Ragdoll te maken.





Deel 3.2: Het maken en implementeren van de AI Ragdoll

Voeg de geconfigureerde bones toe aan het menu, en klik op “Create” om er een Ragdoll van te maken.



Deel 3.2: Het maken en implementeren van de AI Ragdoll

Dubbel check even dan door het spel te starten. Als het goed is valt het mannetje nu om en dit bevestigt dan dat de Ragdoll goed werkt.

Deel 3.2: Het maken en implementeren van de AI Ragdoll

Nu de ragdoll goed werkt gaan wij de logica toepassen waarbij de ragdoll alleen geactiveerd wordt als de speler dus de AI raakt. Onze AI zal een **Navmeshagent** gebruiken om te navigeren in de map maar bij elk spel kan dit verschillen dus let daar op. Verder gebruiken wij ook een **Animator** voor animaties dus deze zullen wij ook gaan uitzetten.

Verder gaan wij de **Rigidbody's** en **Colliders**(Van de Ragdoll) aanzetten wanneer en dan de **Animator**, **Navmeshagent**, en **CapsuleCollider**(Die om het mannetje zelf zit) uitzetten wanneer de AI dus wordt geraakt.

Deel 3.2: Het maken en implementeren van de AI Ragdoll

Dan gaan we nu de waardes die wij nodig hebben toevoegen. Wat belangrijk nogmaals is om dus te weten is dat de mainCol de capsule collider is die om de AI heen zit, anders als hij in een non-ragdoll staat zit dan valt hij door de grond heen.

```
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.AI; // (Dit is voor de NavMeshAgent)
    public class AI_Physics: MonoBehaviour
    {
        public Animator anim;
        public CapsuleCollider mainCol;
        public NavMeshAgent agent;
        private List<Rigidbody> ragdollRigidbodies;
        private List<BoxCollider> ragdollBoxColliders;
        private List<CapsuleCollider> ragdollSphereColliders;
    }
```

Deel 3.2: Het maken en implementeren van de AI Ragdoll

Dan komen we nu bij de functies aan, we beginnen met de **Start()** functie waarin wij de componenten gaan zoeken voor de Ragdoll.

Letop! Wij verwijderen de mainCol van de ragdollSphereColliders natuurlijk om dat deze niet bij de ragdoll hoort maar juist om de gehele AI heen zit.

```
private void Start()
{
        anim = GetComponent<Animator>();
        agent = GetComponent<NavMeshAgent>();

        ragdollRigidbodies =
        new List<Rigidbody>(gameObject.GetComponentsInChildren<Rigidbody>());
        ragdollBoxColliders =
        new List<BoxCollider>(gameObject.GetComponentsInChildren<BoxCollider>());
        ragdollSphereColliders =
        new List<CapsuleCollider>(gameObject.GetComponentsInChildren<CapsuleCollider>());

        mainCol = GetComponent<CapsuleCollider>();
        ragdollSphereColliders.Remove(mainCol); // We verwijderen de

        ToggleRagdollPhysics(false);
}
```

Deel 3.2: Het maken en implementeren van de AI Ragdoll

De volgende functie die wij toevoegen is de **ToggleRagdollPhysics** functie. Deze zet de Animator, Navmeshagent en CapsuleCollider om de AI uit. Daarna zet hij de Ragdoll Colliders aan en de Ridigbody's ook. Wij roepen deze functie aan met een Boolean in de functie zodat je hem makkelijk kan omwisselen.

```
public void ToggleRagdollPhysics(bool state)
{
anim.enabled = !state;
mainCol.enabled = !state;
agent.enabled = !state;

foreach (Rigidbody rb in ragdollRigidbodies) { rb.isKinematic = !state; }
foreach (BoxCollider boxCol in ragdollBoxColliders) { boxCol.enabled = state; }
foreach (CapsuleCollider spheCol in ragdollSphereColliders) { spheCol.enabled = state; }
}
```

Deel 3.2: Het maken en implementeren van de AI Ragdoll

De laatste functie die wij toevoegen is de **OnImpact()** functie waarbij wij krachten uitvoeren op de AI. Hierin roepen wij de **ToggleRagdollPhysics()** functie aan en dan als deze voltooid is en de AI in een Ragdoll staat zich bevindt, dan doen wij een kracht uitvoeren op de ledematen van de AI z'n ragdoll in de foreach, en wordt hij in de richtichtig gegooid d.m.v. die kracht.

```
public void OnImpact(Vector3 force)
{
    ToggleRagdollPhysics(true);
    foreach (Rigidbody rb in ragdollRigidbodies) { rb.AddForce(force); }
}
```

Deel 3.2: Het maken en implementeren van de AI Ragdoll

En dan hebben wij hier nogmaals het eindresultaat van het **AI_Physics** script met alle bijbehorende functies.

```
public class AI_Physics: MonoBehaviour
{
        public Animator anim;
        public CapsuleCollider mainCol;
        public NavMeshAgent agent;
        private List<Rigidbody> ragdollRigidbodies;
        private List<BoxCollider> ragdollBoxColliders;
        private List<CapsuleCollider> ragdollSphereColliders;

        private void Start()
        {
        anim = GetComponent<Animator>();
        agent = GetComponent<NavMeshAgent>();
        ragdollRigidbodies = new List<Rigidbody>(gameObject.GetComponentsInChildren<Rigidbody>());
        ragdollBoxColliders = new List<BoxCollider>(gameObject.GetComponentsInChildren<BoxCollider>());
        ragdollSphereColliders = new List<CapsuleCollider>(gameObject.GetComponentsInChildren<CapsuleCollider>());
        mainCol = GetComponent<CapsuleCollider>();
        ragdollSphereColliders.Remove(mainCol); // We verwijderen de
        ToggleRagdollPhysics(false);
        }

        public void ToggleRagdollPhysics(bool state)
        {
        anim.enabled = !state;
        mainCol.enabled = !state;
        agent.enabled = !state;
        foreach (Rigidbody rb in ragdollRigidbodies) { rb.isKinematic = !state; }
        foreach (BoxCollider boxCol in ragdollBoxColliders) { boxCol.enabled = state; }
        foreach (CapsuleCollider spheCol in ragdollSphereColliders) { spheCol.enabled = state; }
        }

        public void OnImpact(Vector3 force)
        {
        ToggleRagdollPhysics(true);
        foreach (Rigidbody rb in ragdollRigidbodies) { rb.AddForce(force); }
        }
}
```

Deel 4: De eind implementatie.

En dan komen we nu aan bij de eind implementatie. In Deel 2 van deze tutorial hebben wij aan jullie al eerder uitgelegd hoe je het Skill systeem implementeert. En daarna hebben wij een hoop leuke functionaliteiten verder geïmplementeerd en die gaan wij in dit laatste deel implementeren.

Deel 4: De eind implementatie.

Het **DamageChecker** script moet worden geplaatst op het parent object van de collider waarmee de Speler slaat. Dus als je een mannetje hebt met een zwaard, plaats dan het **DamageChecker** script op de parent van dat object.

Speel een beetje met de **Force Multiplier** parameter als de AI te sterk of te zwak wegvliegt als hij wordt geraakt.

Dubbel check even of de Key Events van de Animatie ook goed zijn geïmplementeerd bij de Animatie waar het zwaard/wapen wordt gebruikt.

LETOP: Nogmaals, het is belangrijk dat je bij Collider van het zwaard de parameter “Trigger” aanvinkt. Zodat het optimaal werkt.

Deel 4: De eind implementatie.

Voor het **AI_Physics** script is het belangrijk dat je deze ook op het parent object plaatst van het Character model.

Let wel op dat dit model een Ragdoll moet hebben. Dit hebben wij eerder laten zien hoe je dit doet. Het script regelt zelf alles verder en dus hoef je niets te doen.

Deel 4: De eind implementatie.

En dan is dit het eindresultaat! Een skill systeem voor melee combat based games waarbij als je de AI raakt, zij dood gaan en veranderen in een Ragdoll staat.

(Klik op de video hier rechts om het resultaat te bekijken)

The end

Dit was het einde van de tutorial. Hopelijk is hij verder informatief geweest.

Voor verdere vragen kunt u een e-mail sturen naar het volgende email adres: niels.poelder@gmail.com

Referenties

Unity’s nieuwe InputSyteem Docs

Unity’s nieuwe InputSysteem Tutorial (Brackeys)

Link waar de tutorial live staat (Opdracht vereiste)