## Import af humanoid Mecanim model

For at kunne importere en model til brug for mecanim ind i Unity, skal 3D modellen være sat op på et bestemt måde, før den importeres til Unity. Den proces hvorved man gør en 3D model bevægelig kaldes for ”rigging”. Konkret betyder det, at man allerede i modelleringsprogrammet tilføjer et antal ”knogler” til modellen, på figur 1 er det vist hvordan knoglerne tager sig ud i Blender animationsprogrammet. I Blender er der også funktionalitet til at knytte knoglerne sammen med modellens mesh, altså dens væv.

Når modellen er blevet rigget, så kan man bevæge modellens mesh ved at manipulere med dens knogler.



Figur - modelrigging i blender

Første trin for at kunne oprette vores protagonist Mojo, er er at producere en rigget 3D model af ham, og importere den til projektmappen.

Hvis man åbner modellen i project-vinduet, kan man se hvilke gameobjekter Unity inddeler modellen i. På figur YY, de tre øverste objekter er mesh-objekter og udgør tilsammen figurens udseende. Default\_Take er et animationsobjekt som Blender automatisk genererer ved eksport.   
Hvis man har oprettet en avatar til figuren, så ses det også her som et gameobjekt for sig selv. De tre nederste ikoner repræsenterer hver 3d model som den samlede figur består af, mens armature-objektet er en reference til modellens rigging.

For at kunne benytte Mecanim systemet, skal der være oprettet en avatar af figuren. Ved at markere Mojo-modellen i projektmappen, vises importindstillingerne for 3d-modeller i inspector-vinduet. Under ”Rig”-tabben vælger man hvilken animationstype Unity kan forvente sig af modellen. Mulighederne er ”Legacy” hvorved man blot vælger Mecanim fra, og bruger Unitys gamle system i stedet. ”Generic”, hvilket resulterer i, at man selv skal sætte avataren op. Det giver mulighed for at man kan have alle mulige typer knoglestrukturer i sine figurer. F.eks. firbenede væsener.  
Sidste mulighed er ”Humanoid”, hvor Unity selv opsætter knoglerne på deres formodede position. Dette virker kun hvis der er tale om en figur med to arme, to ben, et hoved, hvilket er tilfældet her, og derfor vælges denne indstilling.   
Indstillingen *Avatar Definition* afgør om Unity skal forsøge at opsætte avataren ud fra den valgte model, eller en anden. Mojo modellen har alt hvad der er påkrævet for at oprette en avatar, så det er indstillingen her. For at gemme indstillingerne klikkes på apply, og dernæst *configure* for at kontrollere Unity´s arbejde.

For at modellen kan fungere som en humanoid kræver Unity som minimum, at der er 15 knogler til stede. Fornuftig navngivning af knoglerne Blender giver mindre manuelt arbejde nu, da det øger muligheden for at Unity selv kan placere hver knogle i den korrekte position. På figur 1 kan man se hvilke positioner der er påkrævede, i og med, at disse er hele cirkler. Cirklerne med stiplede linier omkring indikere at der kan være en knogle på den position, men at det ikke er påkrævet. Mojo har kun de påkrævede 15 knogler, 3 i hver arm, 2 i torsoen, 3 i hvert ben og 1 i hovedet.



Figur - Cirklerne med stiplede linier markere knogler som er valgfrie at tilføje.

Tilbage til Import-indstillingerne, er der to yderligere tabs med indstillinger. Den ene er ”Animations” hvorfra man kan indstille eventuelle animationer som medfølger på figuren. Som tidligere nævnt benytter vi, til vores prototype, udelukkende eksisterende animationer. Den anden tab er ”Model”, der indeholder forskellige importindstillinger for figurens mesh, normaler og materials. Der er ingen materialer på vores prototype figur ved import, så der er intet behov for at ændre i standardværdierne under dette punkt. Normals & Tangents kan med fordel indstilles når der opstår en situation hvor det er relevant at optimere på spillets størrelse, uden de endelige modeller er der ikke noget behov for at ændre i standardværdierne.

Reelt er den eneste *scale den eneste* værdi som der er behov for at indstille på denne side. Som beskrevet i **introduktionen til unity,** så er forventer fysikmotoren i Unity at 1 enhed er lig med 1 meter. Blenders enhed er beregnet på en anden måde, så for at kunne få vores model importeret i den størrelse vi ønsker, skal den opskaleres. Det er forskelligt hvilket skaleringsforhold der skal anvendes, fordi det beror på hvilken størrelse modellen har haft da den blev modelleret. I dette tilfælde passer en 0.5 opskalering til resten af game-verdenen.

## Opret Mojo som spilbar figur

Beskriv trin for trin hvordan mojo bliver ind i scenen.

Efter at have oprettet Mojo som Mecanim-kompatibel model, kan han oprettes i scenen.

Som det første trækkes modellen fra project-view og ind i scene-view. For at kunne se om hans størrelse passer, er det en god ide at lave et standard cube-objekt, til at sammenligne modellen med. Næste udfordring opstår med erkendelsen af, at Unity og Blender ikke bruger samme koordinatsystem. I Unity er Y-aksen den vertikale, mens det i Blender er Z-aksen. Hvis man ikke tager højde for dette, ender man med uhensigtsmæssig opførsel når man senere forsøger at bevæge figuren. Vores scene er sat op i en forventning om at Mojo altid bevæger sig ud af X-aksen. Men pga. den forskel ville han rent faktisk bevæge sig ud af Z-aksen.

Der er flere måder at komme omkring problemet på. Der kunne tages højde for det ved udviklingen af character controlleren, så i stedet for at flytte ham ad Unity´s X-akse, kunne man i stedet flytte ham af Y-aksen, hvilket ganske vidst ville løse det umiddelbare problem. Samtidigt ville det også i væsentlig grad bidrage til at gøre vores kode uoverskuelig, da det betyder at forskellige objekter bruger forskellige koordinatsystemer. En anden mulighed er at tilrette modellen i Blender, så den fra starten har den rigtige retning. Men det er kun en mulighed hvis vi altid selv laver alle modeller, og det er ikke givet.   
For både at kunne bevare overskueligheden, og altid have kontrollen med alle figurer, løses problemet i stedet ved at roterer modellen 90 grader om Y-aksen, kompenseres der for den rotation som modellen blev importeret med.



Figur 3 - tv. før rotation. th. efter rotation

Før der tilføjes flere komponenter, oprettes der en prefab som kan indeholde Mojo-GameObjektet fremadrettet. Den oprindelige figur indeholder ingen af de komponenter som skal tilføjes fra dette punkt og frem, ved at gemme Mojo i en prefab, og blive ved med at opdatere den som processen skrider frem, kan vi nemt oprette en ny instans af GameObjektet, hvis det skulle blive nødvendigt.

### Opsætning af animator controller-komponenten

Da Mojo er oprettet som en Mecanim kompatibel rig, oprettes der automatisk en Animator-komponent på gameobjektet når han trækkes ind i scenen. Animator-komponenten kræver to andre komponenter for at kunne anvendes: Avataren, som vi oprettede tidligere, og en controller. Desuden er der mulighed for at slå *Apply Root Motion* til, hvilket medfører at selve bevægelsen af objektet kommer fra den animation som afspilles i den givne stund. Vi anvender denne metode til at bevæge vores figur, fordi de animationer vi benytter os af passer i tempo. Vi udvikler selv vores produktionsanimationer også, og dermed kan vi selv sikre at de også er afstemt efter hinanden.   
*Animate Physics* vil, hvis funktionen er slået til, betyde at animationen afvikles i Unity´s fysik loop. Hvilket kun er til nytte hvis GameObjektet, som Animatoren er tilknyttet har en Kinematisk Rigidbody. Det er Mojo´s ikke, og derfor er funktionen slået fra.   
Sidste egenskab som der skal tages hensyn til ved oprettelsen er *Culling Mode****.***  ”Based On Renderer”-muligheden medfører at GameObjektet kun animeres når de renderer som er tilknyttet objektet ikke er synlige. Den anden mulighed ”Always Visible” bevirker at GameObjektet altid animeres, uanset om det er synligt eller ej. I forhold til Mojo er det ikke så vigtigt, da han altid vil være synlig på skærmen. Men generelt er der ingen grund til, at animere et objekt som spilleren ikke kan se. Det tjener kun til at optage processorkraft. Uanset hvilket *Culling mode* man vælger, så bliver objekterne flyttet, selv om de er uden for skærmen så længe *Root Motion* benyttes.

Controller-komponenten indeholder den tilstandsmaskine som kontrollerer de skift i animationer som vi har behov for på vores protagonist. I vores GDD er det beskrevet hvilke features Mojo skal have til sin rådighed. Det vil sige, han skal kunne løbe, hoppe, dobbelthoppe, slide og skyde. Controller-komponenten skal opsættes så der er overgange imellem de relevante tilstande.



Figur - Animator komponenten efter oprettelsen af controller komponenten.

Selve komponenten oprettes som et almindeligt asset igennem unity, og navngives ”PlayerController”. Dernæst tilføjer vi den, for nuværende, tomme komponent til Animator-komponenten.  
For at kunne redigere i controlleren, åbnes Animator-Viewet.



Figur - PlayerControlleren

Herfra oprettes de forskellige tilstande, og de transitioner som knytter dem sammen. Desuden kan man her oprette de parametre som benyttes til at udløse en transition. Der er fire muligheder: int, float, bool og trigger, og de kaldes alle via script, som vi kommer til senere. Den eneste vi benytter er ”trigger”, som er en afart af en boolean. Når den bliver udløst sættes den til ”true” indtil transitionen er gennemført, hvorefter den uden videre venter tilbage til ”false”-tilstanden.

En State indeholder en animation, der skal afspilles under en given omstændighed. Der er altid en animation, som skal være standard. I figur 5 kan man se, at den er markeret med orange. I vores spil står protagonisten aldrig stille. Derfor er ”Run” vores udgangspunkt. For at tilføje en animation, markeres den ønskede tilstand, og animationen tilføjes til *motion*-egenskaben i inspector-vinduet. *Speed* indikerer hvor hurtigt animationen skal afspilles relativt til den hastighed hvormed den er optaget. *Foot IK* angiver om der skal tages højde for invers kinematik i animationen. Kort fortalt er det en metode der benyttes for at det skal se mere realistisk ud hvis en figur bevæger sig på en ujævn overflade. Det gør ingen af vores figurer, hvorfor vi har slået det fra. *Transitions*-feltet angiver hvilke ind og udgående transitioner der er på den valgte tilstand. Solo og Mute-checkfelterne er værktøjer som kan anvendes til at debugge tilstandsmaskinen.



Figur - Indstillingsmuligheder for en tilstand

Hvis spilleren hopper, skal der være en overgang til at indikere den handling. Derfor er der imellem Run og Jump oprettet en transition. På figur 7 ses det, at den ”Condition” der kræves for at udløse transitionen er at JumpParam er udløst. Det fremgår også, at overgangen er sat til atomic hvilket bevirker, at overgangsanimationen altid færdigøres før en ny transition kan udløses. Kurven under indikerer den valgte overgangsfase. Hvordan den indstilles vil være forskellige fra gang til gang, og dikteres af hvordan den mest glidende overgang ser ud. Da vi bruger prototype animationer, er det indstillinger som skal revurderes ved implementeringen af produktionsanimationerne.

Af figur 7 fremgår det også at, at der transitionernes tilbage til Run-tilstanden efter 0,85 sekund. Da vi har et dobbeltjump, skal det altså foretages inden for den tidsperiode for at udløse den næste transition.

Af transitionerne på figur 5 fremgår det, at man kun kan hoppe, hvis man kommer fra løb, og kun kan dobbelthoppe hvis man allerede er i gang med at hoppe. På samme måde kan man kun slide, hvis man løber. Med det, er den ene side af vores character controller færdig.



Figur - Øv. transition fra Run til Jump. ndst. Transition fra Jump til Run.

## Tilføjelse af yderligere komponenter

Introducer det faktum at der skal flere komponenter til for at få det til at spille.

### Rigidbody komponenten

En rigidbody-komponent gør det muligt for det GameObjekt som den tilknyttes at blive påvirket af Unity´s fysikmotor. Vi bruger kraft til at bevæge spilleren, hvilket ikke ville kunne lade sig gøre uden rigidbody, ligeledes forlader vi os på tyngdekraft i forbindelse med vores jump metoder.

Der findes andre måder at bevæge GameObjekter på i Unity, f.eks. kan man bevæge objektets transform direkte, eller man kan benytte en character controller-komponent som kommer med Unity. Det er et langt stykke hen af vejen et spørgsmål om at træffe et valg. Den ene metode kan ofte være lige så god som den anden. Vi har valgt at benytte fysik til at bevæge vores pro- og antagonister fordi vi mener det giver os den fornødne direkte kontrol med de objekter.

Attributten *mass* repræsenterer objektets vægt. Massen har en betydning ifh til hvor meget kraft der skal til for at bevæge objektet. Da vi bestemmer både vægt og kraft (også tyngdekraft), er standardværdien 1 blevet fastholdt, og så indrettes påvirkningskræfterne efter det.

*Drag* og *Angular Drag* henviser til vindmodstand, hvilket ikke er en faktor i vores spil hvorfor vi bibeholder standardværdierne 0 og 0.05.

For at vi kan drage nytte af tyngdekraft, skal *Use Gravity* være slået til. *IsKinematic* bevirker at objektet ikke påvirkes af fysikmotoren. I den sammenhæng er det værd at bemærke, at der kan være andre grunde til at bruge en rigidbody, end det vi anvender den til. Til vores formål ville *IsKinematic* aldeles modvirke hensigten.

*Interpolate* og *Collision Detection* bruges til hhv. at smidiggøre animationen hvis den ”hakker”, og til at sikre at objekter der bevæger sig meget hurtigt ikke slipper igennem kollisionsmaskeringen. Vi har ladet begge dele stå på deres standardindstillinger, da vi dels ikke har et behov for at ændre det, og dels at det er de mindst ressourcekrævende indstillinger.

De sidste indstillinger er objektets *Constraints*. Som nævnt bevæges Mojo ved hjælp af kraft, ligesom alt hvad der findes i den virkelige verden. Men han har ingen balanceevne, så for at han ikke skal vælte omkuld kan vi låse hans rotation omkring både X-, Y- og Z-aksen. Det er ligeledes muligt at låse objektets position på alle tre akser. Det gør vi dog ikke, fordi det giver uhensigtsmæssigheder i kombination med at vi anvender *Root Motion* til fremdriften.

### Capsule Collider, fysik-komponenten

For at undgå, at Mojo falder igennem de objekter som han står på tilføjer vi en Capsule Collider-komponent. Der findes en række forskellige collider-komponenter indbygget i unity. Vi benytter en capsule, fordi formen passer godt på en humanoid.



Figur - Mojo omgivet af Capsule Collider

Attributten *Is Trigger* angiver om Collideren skal fungere som en trigger eller, som her, om den skal være en fysik collider.

*Material* åbner mulighed for, at man kan tilføje et fysikmateriale til komponenten. Ved hjælp af et sådant kan man påvirke friktionen mv. imellem to gameobjekter. Det er ikke indenfor scope af denne prototype at benytte fysikmateriale.

De resterende egenskaber benyttes til at indstille colliderens størrelse og position ifh. til det GameObject som den er tilknyttet. På figur 8 er den indstillet til at dække Mojo, og kan ses som et grønt grid over selve figuren.



Figur 9 - Capsule Collider-komponenten

### Capsule Collider, Trigger-komponenten

Udover at have et behov for ikke at falde igennem objekter, skal Mojo også bruge en trigger collider. Med en sådan tilknyttet, kan andre objekter bekræfte når deres egen trigger collider interagerer med Mojo´s. Fordi der allerede er tilknyttet en collider på Mojo´s øverste GameObject, er det ikke praktisk at tilknytte en anden af samme type, da det forvirrer den metode hvormed man kalder komponenter. Løsningen er at oprette et tomt GameObjekt, og gøre den til et barn af Mojo GameObjektet. Derefter kan man tilføje en Capsule Collider til det objekt, og give den de samme dimensioner som fysik collideren. Eneste forskel er, at til denne collider skal *Is Trigger*-checkboksen vinges af.

### PlayerScript-komponenten

PlayerScriptet oprettes fra Asset-menuen som et C#-script, og placeres i projektmappen under Scripts.

Formålet med Scriptet er at forbinde de allerede oprettede komponenter, således at den opførsel som de er tilføjet for at bidrage med, kommer ordentligt til udtryk.

Det første vi gør i Scriptet, er at erklære at UnityEngine skal anvendes, dernæst scriptets navn, og at vi nedarver fra MonoBehaviour.

Dernæst oprettes en række lokale og private variabler. De variabler som oprettes som public, bliver vidst i Unity-editoren, på samme måde som de attributter der vises i f.eks. RigidBody-komponenten. Det har begrænset praktisk betydning hvilke variabler vi sætter til hhv. privat og public, da vi er så lille et team som tilfældet er. Ideen med at have denne opdeling er, at en designer, producer eller grafiker skal kunne justere i visse variabler uden at skulle håndtere koden. Fordi det er vores mening, at det er en god måde at effektivisere samarbejdet på, forsøger vi hele tiden at skelne til hvad der skal kunne ændres af alle, og hvad der er programmørens arbejde at ændre i.

Som det fremgår af Figur. XXX, kan man ændre i den kraft som objektet bliver påført når man hopper. Det er ligeledes muligt at ændre i tyngdekraften, samt hvor kraftigt et swipe på skærmen hhv. op og ned skal være for at blive evalueret til et swipe.

De værdier afgør i høj grad om gameplayet fungerer som vi ønsker det, og ved at gøre dem public kan alle i teamet let teste forskellige kombinationer af.

Super, men hvilken funktionalitet er der i selve koden, og hvornår starter det?

Ved GameObjektets oprettelse køres scriptets Start() som det første. Og for eneste gang i øvrigt, indtil objektet oprettes på ny.

I den får vi cachet, altså lagret, referencer til de komponenter som skal bruges i løbet af scriptet. Formålet med at gøre det på denne måde, er at det er ”dyrt” i ressourceforbrug at benytte de metoder som fremfinder komponenterne, hvis man gør det for. F.eks. hver frame. I vores prototype ville det ikke være mærkbart, men det der er ingen grund til at gøre det anderledes.

Figur 10 - PlayerScript Start()

void Start () {

\_anim = GetComponent<Animator>();  
 \_psysCol = GetComponent<CapsuleCollider>();  
 \_body = GetComponent<Rigidbody>();  
 \_collisionCol = GameObject.Find("CollisionObj").GetComponent<CapsuleCollider>();  
}

GetComponent<T>() gennemsøger det objekt som scriptet er tilknyttet efter en komponent af den type som er angivet. Det er på grund af denne metode, at det ikke er hensigtsmæssigt at have to komponenter af samme type på det samme GameObjekt. Søgningen starter nemlig altid samme sted, og stopper når den første komponent er fundet. Ret beset findes der en alternativ metoder: GetComponents<T>() hvormed man kan søge flere af samme type. Men i givet fald skulle det håndteres i en form for liste, og dernæst alligevel gemmes i hver sin variabel, hvormed det er en mere akavet løsning end den valgte.

For at finde frem til den anden collider benytter vi metoden Find(string), der tager navnet på det eftersøgte gameobjekt som parameter. Da det ikke er gameobjektet vi skal bruge, benyttes GetComponent<T> direkte på det fundne gameobjekt.

Efter Start() er afsluttet går scriptet direkte ind i Update(), som bliver kaldt for hver frame, hvilket i praksis vil sige, at alt hvad der foregår i denne metode kaldes mange gange i sekundet.

For PlayerScriptet er der tre separate trin der udføres i denne metode. Først opdateres den private variabel \_currentBaseState, med den nuværende aktive tilstand i vores Controller-komponent (som jo er tilknyttet Animatorkomponenten, hvorfra denne oplysning trækkes.)

Den næste linie i Figur 11, sikrer at Mojo altid løber lige ud. De animationer der er tilknyttede er ikke hundrede procent i vatter, hvilket uden denne linie, der for hvert kald til metoden nulstiller Mojo´s position på Z-aksen. Under kapitlet om Rigidbody blev det nævnt, at det er muligt at låse Z-aksen helt. Men gør man det, så forhindrer man samtidigt at en animation i det hele taget kan bevæge sig på de låste akser, og det er mildest talt ikke et kønt syn. Derfor håndterer vi det således.

void Update () {

\_currentBaseState = \_anim.GetCurrentAnimatorStateInfo(0);   
 transform.position = new Vector3(transform.position.x, transform.position.y, 0);

#if UNITY\_ANDROID  
 SwipeControls();  
#endif

#if UNITY\_EDITOR  
 KeyboardControls();  
#endif

Figur 11 - PlayerScript Update()

De to if-sætninger er en speciel Unity-konstruktion, der sikrer platformspecifik kompilering. Alt hvad der ligger under UNITY\_ANDROID bliver kun kompileret hvis man bygger et Android build. Det giver udviklere en mulighed for at designe scripts med henblik på udgivelse til flere platforme. Alternativt ville hele scriptet skulle kopieres, blot for at ændre det som er platformsspecifikt.

Vi anvender det her, for at sikre at vi har input-controls teste spillet både på vores mobile enheder, og i Unity editoren. Da KeyboardControls() kun er til testbrug, vil vi ikke uddybe den nærmere.   
Den interessante af de to, er SwipeControls() der tager touch input. I den første if-sætning kontrolleres det om der er registreret mere end 0 berøringer, og om den første berøring er i Moved-fasen.  
Hvis det er tilfældet så findes berøringens deltaposition, altså dens position i denne frame, kontra der hvor den blev registreret sidst.

Vi er interesserede i swipet, hvis det bevæger sig enten op, eller ned. Hvis deltapositionen, som er en todimensionel vektor, er større end den værdi som er angivet til at tælle som et opadgående swipe, og den nuværende controller-tilstand er Run, så kaldes Jump(), og vi udløser den trigger der transitionerer controller-tilstanden fra Run til Jump. Ydermere ændres debug-teksten.

Hvis swipet er opadgående, og controller-tilstanden evaluerer til Jump, så udløses DoubleJump-transitionen. Og Jump-metoden kaldes på ny.

Figur 12 - SwipeControls() håndtere brugerinput

private void SwipeControls()

{

if (Input.touchCount > 0 && Input.GetTouch(0).phase == TouchPhase.Moved)

{

//Find bevægelsen fra sidste frame til nu

Vector2 deltaTouchPos = Input.GetTouch(0).deltaPosition;

//Hvis swipet er opadgående og spilleren har jordforbindelse

if (deltaTouchPos.y > swipeUp && \_currentBaseState.nameHash == \_runState)

{

Jump();

\_anim.SetTrigger("Jump"); //Opdaterer animatoren

\_demoText = "Jump";

}

//hvis swipet er opadgående, og spilleren i forvejen i luften

else if(deltaTouchPos.y > swipeUp && \_currentBaseState.nameHash == \_jumpState)

{

Jump();

\_anim.SetTrigger("DoubleJump"); //Opdaterer animatoren

\_demoText = "Double Jump";

}

//Hvis swipet er nedadgående og spilleren har jordforbindelse

else if (deltaTouchPos.y < swipeDown && \_currentBaseState.nameHash == \_runState)

{

\_demoText = "Duck";

\_anim.SetTrigger("SlideParam"); //Opdaterer animatoren

Slide();

StartCoroutine(WaitForSlide());

}

}

}

Hvis swipet derimod er nedadgående, altså en negativ y-værdi på deltapositionsvektoren, og Mojo er i løb, så udløses Slide-transitionen. Slide() ændrer størrelsen, og placeringen af den fysiske collider som er tilknyttet Mojo.

Grunden til denne manøvre er, at Mojo´s collidere skal ændre størrelsen. Animationen som sådan har nemlig ingen påvirkning på collider-komponenterne. Det vil sige, at selv om man kan se Mojo dukke sig på skærmen, så ville han stadig kollidere med eventuelle forhindringer som han ville have ramt hvis han stod op. Jf. koden i figur 13 sættes højden på fysik collideren til en fjerdedel af normal, ligesom dens centrum flyttes end til en fjerdedel af dens normale placering.

void Slide()

{

\_collisionCol.enabled = false;

\_psysCol.height = 0.25f;

\_psysCol.center = new Vector3(0, 0.25f, 0);

}

Figur 13 - Slide() sætter collidere til størrelse hvor de kan gå under forhindringer

Coroutinen WaitForSlide() har til formål at sikre, at fysik collideren ikke bliver sat tilbage til sin oprindelige position, før slide-animationen er udført. Derfor venteperioden på 1.2 sekund, før StandUp() kaldes.

Figur 15 - Coroutine eksempel

IEnumerator WaitForSlide()

{

//Debug.Log("waitforit");

yield return new WaitForSeconds(1.2f);

StandUp();

}

Efter slide-animationen er færdig, transitionerer tilstanden tilbage til Run. Derfor skal vi sørge for at Mojo´s collidere igen passer med hans oprejste form. StandUp-metoden som vidst i Figur 16 tilser dette.

Figur 16 - StandUp()

void StandUp()

{

\_psysCol.height = 1f;

\_psysCol.center = new Vector3(0, 0.5f, 0);

\_collisionCol.enabled = true;

}

## Opsummering

Afslutningsvis opdateres Mojo-prefabben med alle tilføjelser, og han placeres på det sted i scenen hvorfor han skal starte ved spilstart.