

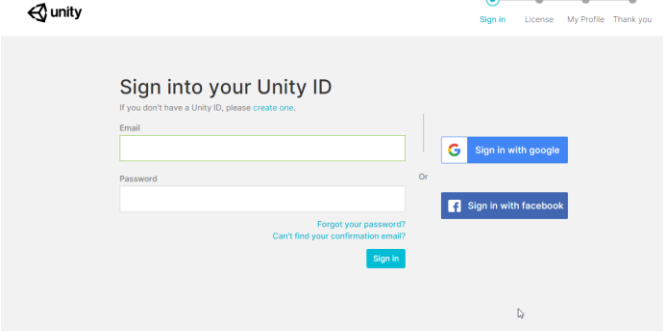
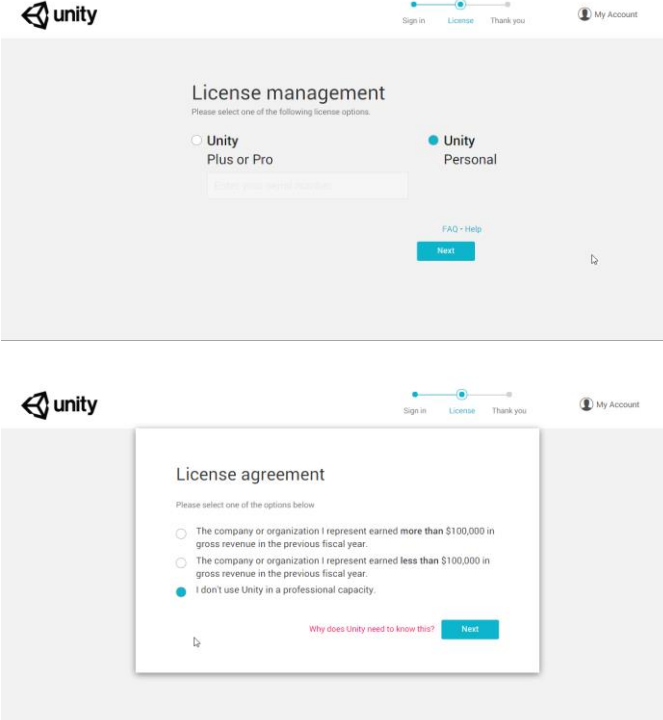
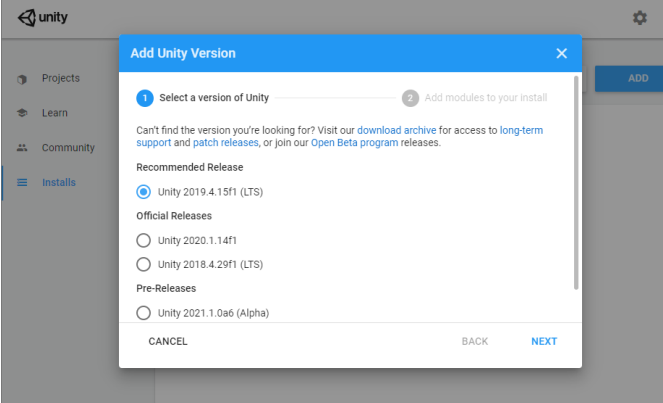
**howest**  
hogeschool

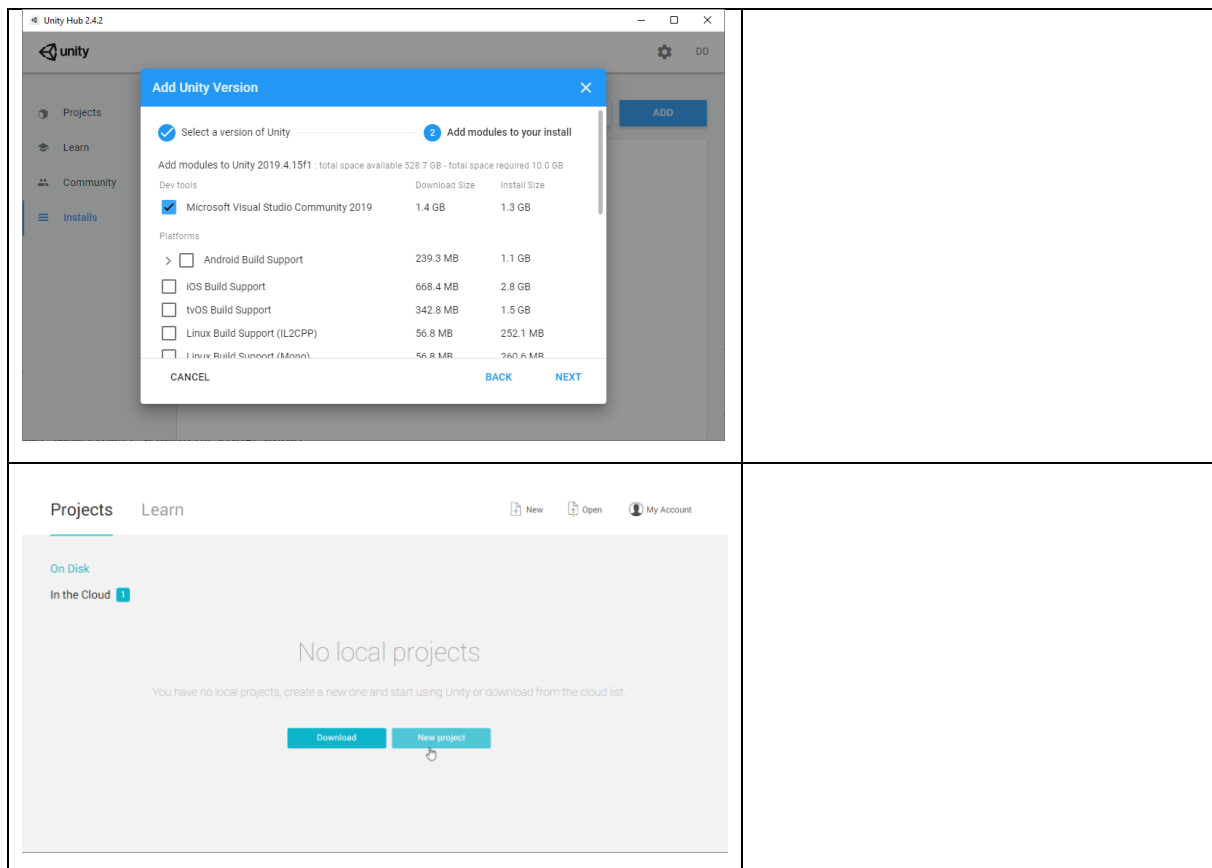
# Game Development in Unity

<b>1</b>	<b>INLEIDING</b>	<b>3</b>
<b>1.1</b>	<b>Unity installeren</b>	<b>3</b>
<b>1.2</b>	<b>Een nieuw project maken</b>	<b>4</b>
<b>1.3</b>	<b>Een nieuwe scene maken</b>	<b>6</b>
<b>1.4</b>	<b>De speler maken</b>	<b>7</b>
1.4.1	GameObjects	7
1.4.1.1	3D Objects	7
1.4.1.2	Asset store	7
1.4.1.3	het inspector tabblad	7
1.4.1.4	Components	8
<b>1.5</b>	<b>Het speelveld maken</b>	<b>9</b>
<b>1.6</b>	<b>Lightning</b>	<b>9</b>
<b>1.7</b>	<b>Materials</b>	<b>9</b>
<b>1.8</b>	<b>Reageren op player input</b>	<b>10</b>
1.8.1	Input system package	10
1.8.2	Script koppelen	11
1.8.3	Essentiële basisconcepten voor het script	11
1.8.3.1	Frames	11
1.8.3.2	Analyse van het scriptsjabloon	12
1.8.3.3	event functions en execution order	13
1.8.3.4	SendMessage()	14
1.8.3.5	Vectoren	15
1.8.3.6	2D vs. 3D	17
<b>1.9</b>	<b>De bal gaat aan het rollen ...</b>	<b>18</b>
1.9.1	De bal bewegen	18
1.9.2	De snelheid van de bal controlleren	19
1.9.3	De camera de bal laten volgen	20
<b>1.10</b>	<b>Het speelveld afbakenen</b>	<b>24</b>
<b>1.11</b>	<b>Gameplay toevoegen</b>	<b>25</b>
1.11.1	Collectable items	25
1.11.1.1	Een collectable item maken	25
1.11.1.2	Een script voor het collectable item	26
1.11.1.3	Het collectable item Herbruikbaar maken	27
1.11.2	collision (botsing) detecteren	28
1.11.2.1	Collision detecteren	28
1.11.2.2	Unity tag system	29
1.11.2.3	Efficiënte collisie met een rigidbody	31
1.11.3	GUI maken	32
1.11.3.1	GUI maken	32
1.11.3.2	De tekst aanpassen vanuit het script	34
1.11.4	Oefening – win message	36

# 1 INLEIDING

## 1.1 UNITY INSTALLEREN

	<p>Log in met je bestaand account of maak een nieuwe Unity account aan.</p>
	<p>Kies Unity Personal en selecteer dat je geen professioneel gebruik toepast.</p>
	<p>Voeg via de Unity Hub -&gt; Installs een nieuwe installatie toe van Unity. Neem hiervoor de recommended release.</p>

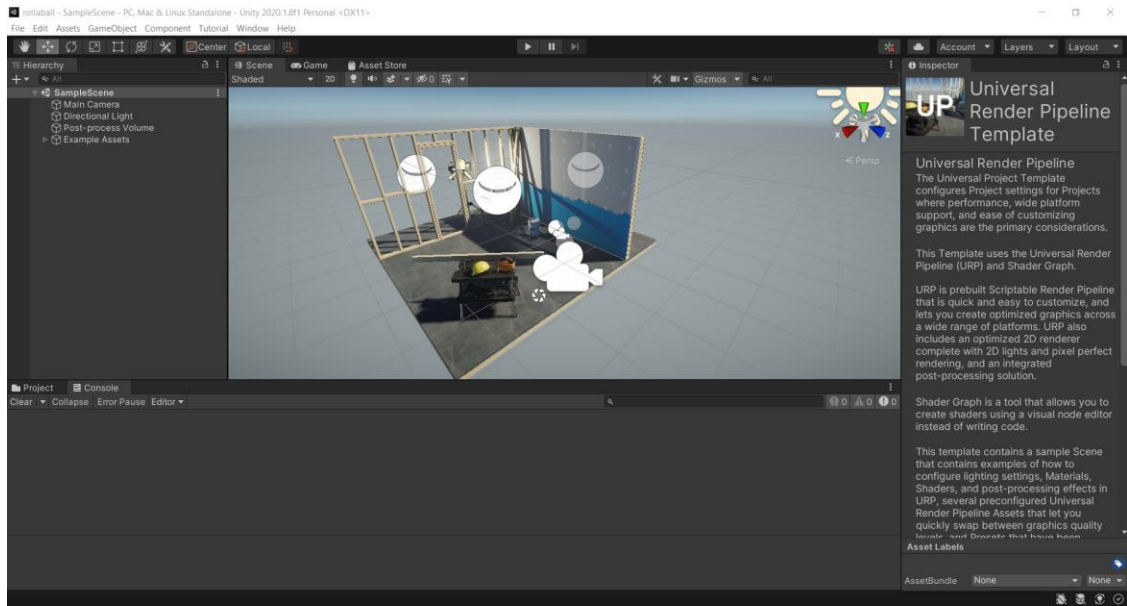


## 1.2 EEN NIEUW PROJECT MAKEN

Start de Unity Hub en login in met account. Selecteer Projects > New om een nieuw project aan te maken. Je krijgt een aantal templates te zien om een Unity project aan te maken. Dit scherm doet erg denken aan het 'New Project' scherm uit Visual Studio.

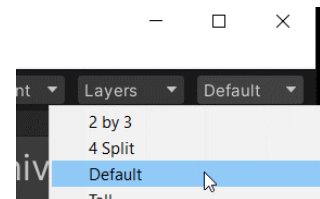
Kies hier voor 'Universal Render Pipeline' (URP) als template. Door te kiezen voor deze template, ondersteunt je project bijna alle belangrijkste platformen: WebGL, PS4, Xbox One, Microsoft Windows, Microsoft Hololens, Oculus, Android, iOS, enzovoort. Bovendien zorgt het voor performantere en mooiere renders (weergaves) van je spel. Kies bovendien een projectnaam "Rollaball" en locatie voor je project en klik op 'Create'. Je Unity project wordt aangemaakt. Dit kan enige tijd duren.

Na het laden wordt een start-scene aangeboden. Deze toont de basisfeatures van URP (lightning, texturing, enzovoort):

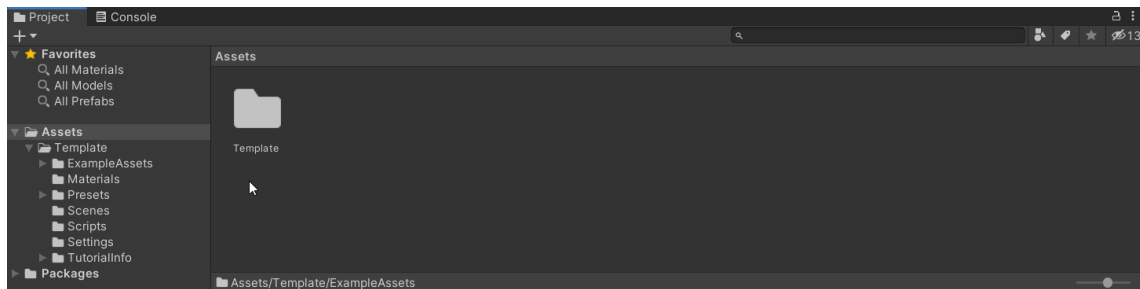


Om eenvoudig te kunnen werken, stellen we alvast een basis layout in voor Unity zelf. Ga in de rechterbovenhoek naar 'Layout' en kies er voor 'Default'.

Vervolgens zullen we het startsjabloon opzij zetten, zodat we een eigen project kunnen maken. Klik op 'File' > 'New Scene'. De startscene verdwijnt.

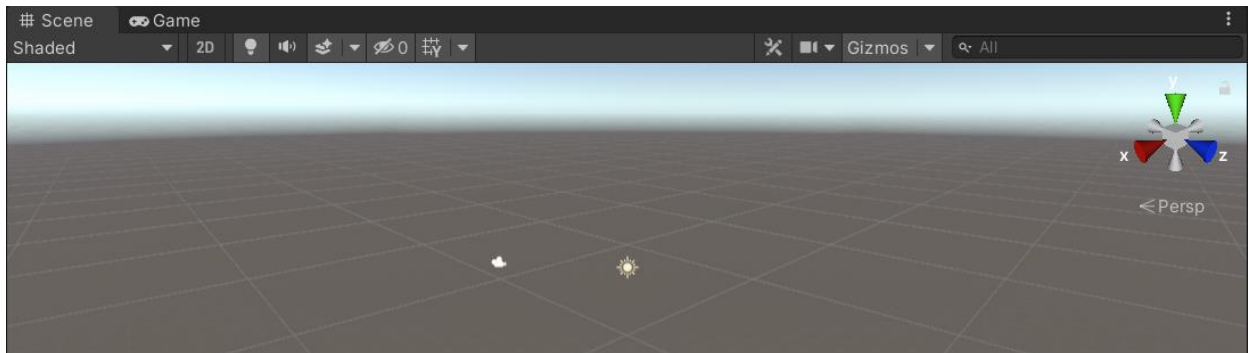


Onderaan zie je een tab met de naam 'Project', waar je tevens een map 'Assets' terugvindt. Deze map bevat alle files die ons spel nodig heeft. De files van de template hebben we niet onmiddellijk nodig. Selecteer de Assets folder en maak er een nieuwe map 'Templates'. Dit doe je door te klikken met de RMB > Create > Folder. Verhuis vervolgens alle reeds bestaande mappen naar deze map. Houd je mappenstructuur ordelijk, dit is belangrijk om later bestanden terug te vinden.

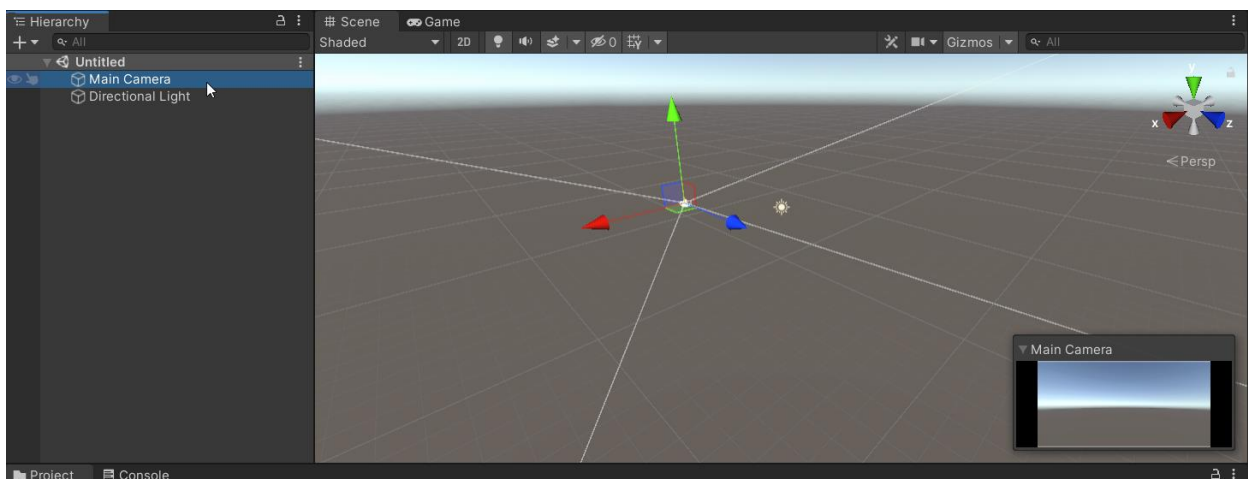


### 1.3 EEN NIEUWE SCENE MAKEN

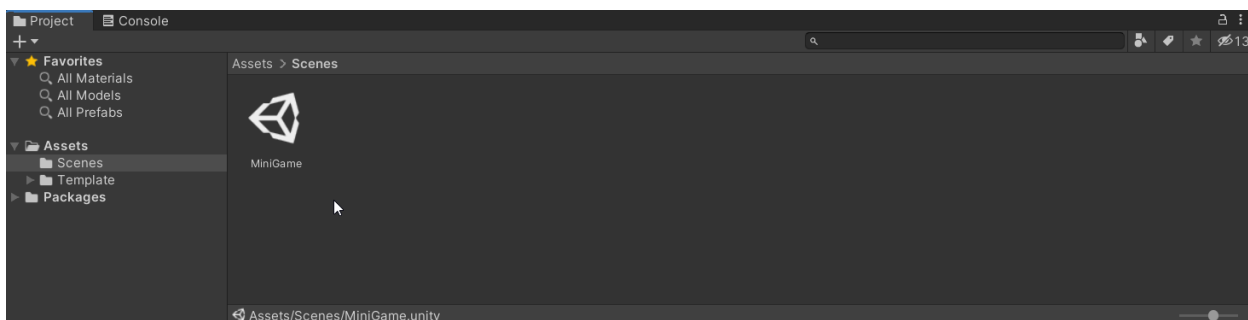
Daarnet klikte je reeds op 'File' > 'New Scene'. Dit maakte alvast een nieuwe scene aan. Beschouw een Scene als een level in je spel: een scene bevat een unieke omgeving, decoratie. Achterliggend heeft elke scene (level) eigen bestanden en programmacode. Deze wordt weergegeven in het *Scene tabblad*:



Bovendien heeft je scene reeds 2 componenten: een *Camera* die verantwoordelijk is voor het weergeven van de scene en een *Directional light* die dient als lichtbron in je scene om de objecten die erin staan te verlichten. In het 'Hierarchy' tabblad worden deze twee componenten alvast getoond:



We zullen ons 'level', de huidige *Scene*, eerst en vooral opslaan. Klik 'File' > 'Save As ...'. Maak in de Assets folder een nieuwe folder met de naam *Scenes*. Hier zullen we al onze scenes bewaren. Sla er de Scene op met als naam '*Minigame*'.



## 1.4 DE SPELER MAKEN

### 1.4.1 GAMEOBJECTS

Elk object in Unity is een GameObject. De camera en het licht die we daarnet al leerden kennen zijn dit dus ook. Op zichzelf doet een GameObject niet veel. Ze dienen namelijk als een container voor *Components*. Deze *Components* implementeren de functionaliteit van het object in kwestie. We onderzoeken dit terwijl we ons spelerobject maken:

#### 1.4.1.1 3D OBJECTS

Unity bevat reeds een heleboel ruimtelijke figuren die we out-of-the-box kunnen gebruiken. Voor ons spel gebruiken we een bol (Eng: Sphere) als object voor de speler. Het spreekt voor zich dat er complexe 3D-figuren gemaakt kunnen worden om in je game te gebruiken. Deze taak wordt doorgaans vervuld door **3D Artists en Designers**. Doordat we echter de basisfiguren reeds ter beschikking hebben, kunnen we snel aan de slag met het maken van een **prototype**.

Tik in het 'Hierarchy' tabblad *RMB > 3D Object > Sphere*. Er verschijnt een bol in de scene. Het 'Inspector' tabblad bevat nu de gegevens van de Sphere omdat deze geselecteerd staat.

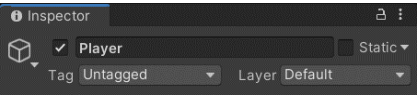

#### 1.4.1.2 ASSET STORE

In de Unity Asset Store (<https://assetstore.unity.com/>) kunnen verschillende assets, tools, ... gedownload worden. Sommigen hiervan zijn gratis, anderen te betalen.

#### 1.4.1.3 HET INSPECTOR TABBLAD

We voegden tot dusver toe een GameObjec toe, meer bepaald een Sphere (3D Object).


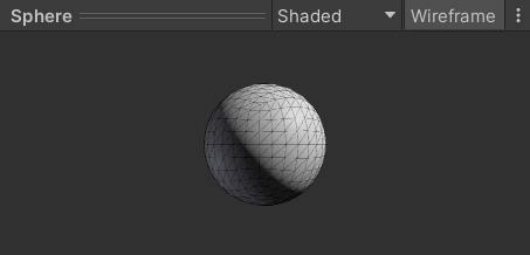
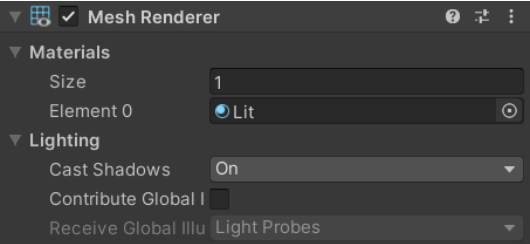
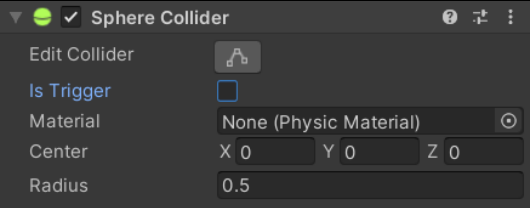
Selecteer je Sphere (die later dienst zal doen als het spelerobject). En analyseer het 'Inspector'-tabblad. Deze bevat alle eigenschappen van het geselecteerde object. We bespreken er alvast twee:

Inspectorveld	Bevat een naam voor het object	
Transform	<p>Dit bepaalt de positionering in de 3D ruimte a.d.h.v. een X, Y en Z-coördinaat samen met de rotatie en schaal van het object.</p> <p>Belangrijk om weten is dat we naar X: 0, Y: 0, Z: 0 verwijzen als het <b>origin point</b>. Vanaf dit punt worden alle posities in het spel berekend.</p> <p>Bovendien is het relevant dat weten dat de standaard 3D object een scale (grootte) hebben van X: 1, Y: 1, Z: 1. In Unity termen spreken we over 1 Unity Unit, wat overeenkomt met een schaal van 1 meter. Deze schaal kan echter aangepast worden.</p>	

Pas de naam van het object alvast aan naar 'Player'.

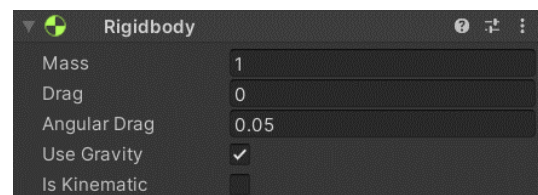
#### 1.4.1.4 COMPONENTS

Zoals eerder vermeld doet een GameObject op zichzelf niets. Het is een container voor verschillende Components. In de Inspector zien we een aantal gekoppelde components: een Sphere Mesh Filter en een Mesh renderer. Deze componenten doen het volgende:

<b>Sphere (Mesh Filter)</b>	<p>Bevat de 'Mesh', het geraamte, van de Sphere. Het zijn polygonen die de bol opbouwen.</p> <p>Dubbelklik op de Mesh om ze zichtbaar te maken. Je ziet het wireframe van de sphere. Merk op dat deze uit driehoeken zijn opgebouwd.</p>	 <p>(na dubbelklik):</p> 
<b>Mesh Renderer</b>	<p>Component verantwoordelijk om het object (met Mesh) te tonen. Met eigenschappen zoals het gekoppelde 'Material', hoe het object reageert op Lightning, enzovoort.</p> <p><i>Vink de Mesh Renderer een af en aan. Het object wordt onzichtbaar in de scene.</i></p>	
<b>Sphere Collider</b>	<p>Component verantwoordelijk om te detecteren of het botst met een ander GameObject. Bevat ook eigenschappen over de collider zelf.</p>	

Ons GameObject kan dus nog niet echt veel doen. Klik eventjes op de Play-knop bovenaan de Scene (▶ || ▶). Het spel start op in test mode. Er gebeurt echter weinig. De bal zweeft gewoon in de lucht. We voegen nu een extra Component, eigenschap, toe aan de bal. Klik nogmaals op de Play-knop en keer terug naar de Inspector van de bal. Klik onderaan op "Add Component". Kies vervolgens voor 'Fysics' > 'Rigidbody'.

De Rigidbody component zorgt ervoor dat de bal intrageert met de **fysics engine** van Unity. Simpel gezegd: de wetten van de fysica zijn nu van toepassing op de bal. Klik nogmaals op de Play-knop: de bal valt naar beneden, net zoals dat in de echte wereld zou gebeuren. De bal valt





nu echter eeuwig naar beneden en is niet meer toegankelijk in de scene. Bij de eigenschappen van de Rigidbody Component zie je nog andere settings kan aanpassen waaronder bijvoorbeeld de massa.

## 1.5 HET SPEELVELD MAKEN

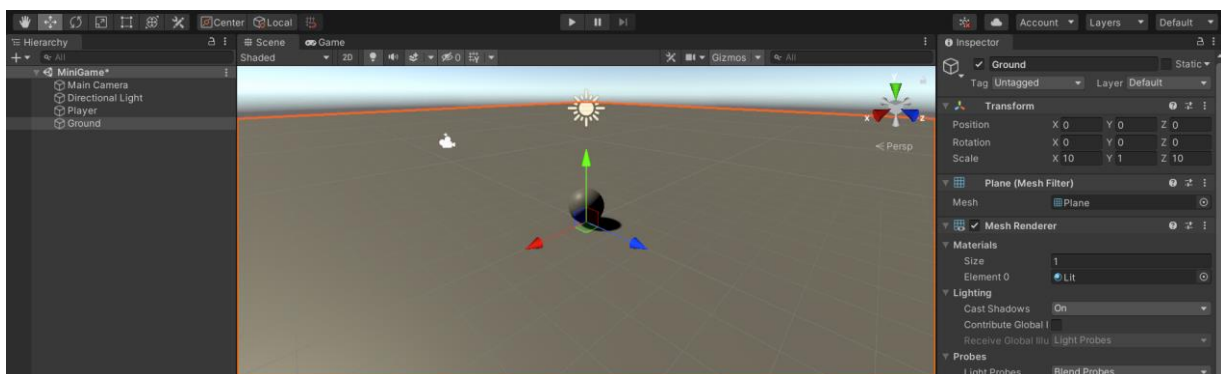
Voorlopig valt onze bal naar beneden, waarna het verdwijnt voor eens en eeuwig. Dat komt omdat we nog geen speelveld hebben gemaakt waarop onze bal rolt.

In het hierarchy tabblad tik je **RMB > 3D Object > Plane**. Het Plane GameObject wordt netjes geplaatst in het origin point, maar dit zorgt voor wat problemen: het snijdt de bal doormidden. We kunnen dit simpel oplossen door de bal een halve unit naar boven te verplaatsen. Stel de position van de bal in op X: 0, Y: **0.5**, Z: 0.

Verken het volgende:

- Stel een grotere waarde in voor de Y-positie van de bal: de bal valt van hoog naar beneden op het speelveld (Plane).
- Selecteer het Plane en stel de X-rotatie in op -5: de bal rolt rustig naar beneden.
- Ga naar de Inspector van het bal en vink de Sphere Collider af. Preview je game: de bal valt door het speelveld omdat het niet meer weet wanneer het met iets botst.
- Stel alle waardes terug in zoals ze hoorden na het testen ...

Het speelveld is nogal klein. Schaal het groter door in de Inspector de *Scale* property in te stellen op X: 10, Y: 1, Z: 10. Het plane wordt dus in de lengte en breedte 10 Unity Units oftewel 10 meter lang. Geef het bovendien de naam 'Ground'.



## 1.6 LIGHTNING

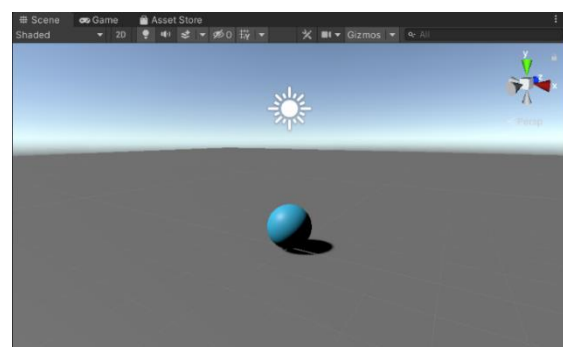
Het Unity project kreeg standaard een 'skybox' en een lichtbron mee. Deze lichtbron stelt hier als het ware de zon voor. Selecteer het Directional Light in het *Hierarchy* tabblad en vink het eens af. Het spel wordt donker, maar je ziet wel nog de sky box. Vink het Light weer aan en pas de kleur aan naar keuze. In de cursus kiezen we voor puur wit, rgb(255, 255, 255).

Nu het licht geselecteerd staat kan je de richting ervan aanpassen. Pas de rotation van het licht aan naar X: 50, Y: 50, Z: 0.

## 1.7 MATERIALS

Materials zijn een combinatie van een Texture en een Color. In ons project maken verlopig geen gebruik van Textures.

Maak in de Assets map een nieuwe map en geef het de naam 'Materials'. Klik **RMB > Create > Material** en geef het de naam 'Player'. Selecteer het in stel onder *Surface Inputs* de Base Map in door op het kleur te klikken. Geef de waarde #44c8f5 in om het Howest-blauw in te

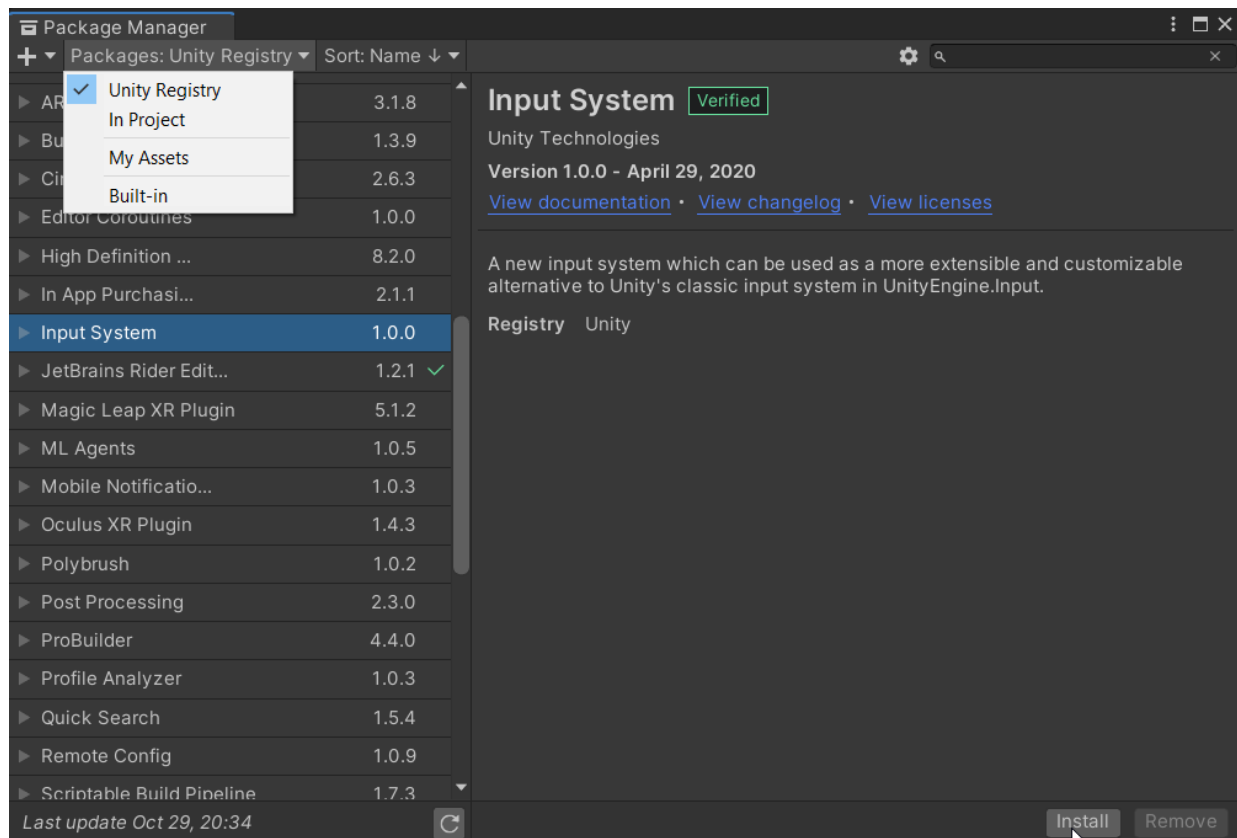


stellen. Sleep vervolgens het zonet aangemaakte Material op de bal om het erop toe te passen. Selecteer de bal en navigeer nogmaals naar de Inspector. Merk op dat onder de 'Mesh Renderer' je Material werd toegevoegd.

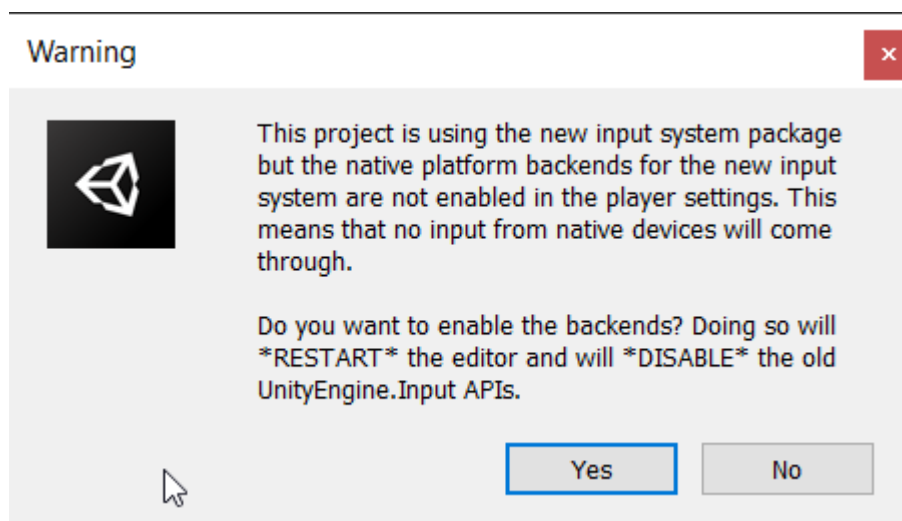
## 1.8 REAGEREN OP PLAYER INPUT

### 1.8.1 INPUT SYSTEM PACKAGE

Om geen custom code te hoeven schrijven die luistert naar user inputs, installeren we een package die deze basisfunctionaliteit voor ons zal ondernemen. Ga in de menubalk naar *Window > Package Manager* en zoek er naar de *Input System* Package van Unity Technologies. Installeer het in je project.

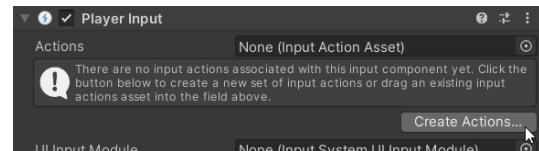


Mogelijks krijg je deze waarschuwing:



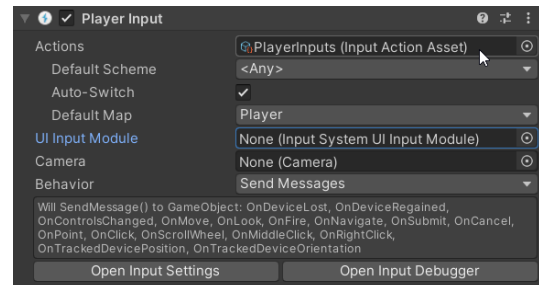
Indien je deze waarschuwing krijg, dan klik je op 'Yes' (en later op 'Save'). Het project zal opnieuw opstarten. Vervolgens ga je naar *File > Build Settings*. Daar kies je bij 'PC, Mac & Linux Standalone' voor de 'x86\_64' Architecture optie. Sluit het menu af.

Maak in de map *Assets* wederom een nieuwe map met de naam *Input*. Ga vervolgens naar de Inspector van de speler (bal) en voeg een Component toe. Zoek ditmaal naar 'Player Input' (de Package die we zonet installeerden). Nu de 'Player Input' Component toegevoegd is klik je op de 'Create Actions ...' knop van het component om default acties op de bal toe te voegen.



Een nieuw dialoogvenster gaat open. Sla het bestand op in de map die je zonet aanmaakte en geef het de naam 'PlayerInputs'.

Vervolgens koppelen we het script aan het Player GameObject. Klik naast 'Actions' op het kleine bolletje en koppel de PayerInputs file.



In het infoschermpje lezen we alvast dat de Player Input Component een SendMessage() kan uitvoeren naar het GameObject waaraan het gekoppeld is. De SendMessage()-methode is een gekoppeld aan het GameObject en is in staat om andere methodes van het GameObject te triggeren. Later meer.

## 1.8.2 SCRIPT KOPPELEN

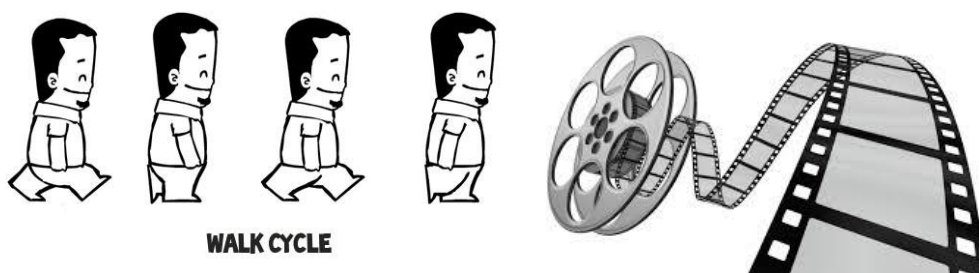
Een script is een stukje code die uitgevoerd kan worden door een GameObject.

Maak in de *Assets* map een nieuwe map met de naam '*Scripts*'. Open de map en tik *RMB > Create > C# script*. Maak een nieuwe file aan met de naam '*PlayerController*'.

Vervolgens willen we het (nu nog lege) script toevoegen aan onze speler zelf. Ga naar de bal, de speler en klik op 'Add Component'. Vervolgens zoek je naar '*Player Controller*' (het script dat je net maakte) en voeg je het toe. Onze player heeft nu dus een eigen component gekoppeld die verantwoordelijk zal zijn om de speler te kunnen laten bewegen op basis van user input. Voorlopig is dit script echter nog leeg.

## 1.8.3 ESSENTIËLE BASISCONCEPTEN VOOR HET SCRIPT

### 1.8.3.1 FRAMES



Iedereen kent een filmrol wel. Hierop staan telkens beelden die telkens lichtjes anders zijn van elkaar. Door deze stilstaande beelden snel na elkaar af te spelen, wordt de illusie van een bewegend beeld geschept. In de afbeelding linksboven zien we 4 frames. Wanneer we deze snel in volgorde zouden afspelen, dan lijkt het alsof het mannetje wandelt. Wanneer je TV kijkt bestaat de video echter doorgaans uit 25 frames per seconde. Games gebruiken dan weer minimaal 30 fps (frames per second) en bij voorkeur zelfs 60 fps voor een vloeiende ervaring.

### 1.8.3.2 ANALYSE VAN HET SCRIPTSJABLOON

Ga in de Assets/Scripts folder naar het PlayerController script en dubbelklik erop. Visual Studio gaat open en we zien de programmacode van het script.

De PlayerController script erft over van MonoBehaviour. Dit is de base class waar elk script van over **moet** erven. Het voorziet ook een aantal methodes die door de Unity Engine gecalled kunnen worden. We bespreken de belangrijkste die we momenteel nodig hebben:

void Start()	Wordt gecalled vlak voordat het eerste frame geüpdated wordt.
void Update()	Methode die <b>elk frame</b> wordt gecalled. Deze methode wordt gecalled vóór het renderen van een frame. Per computersysteem verschilt de framerate.  Hier komt de meeste code terecht.

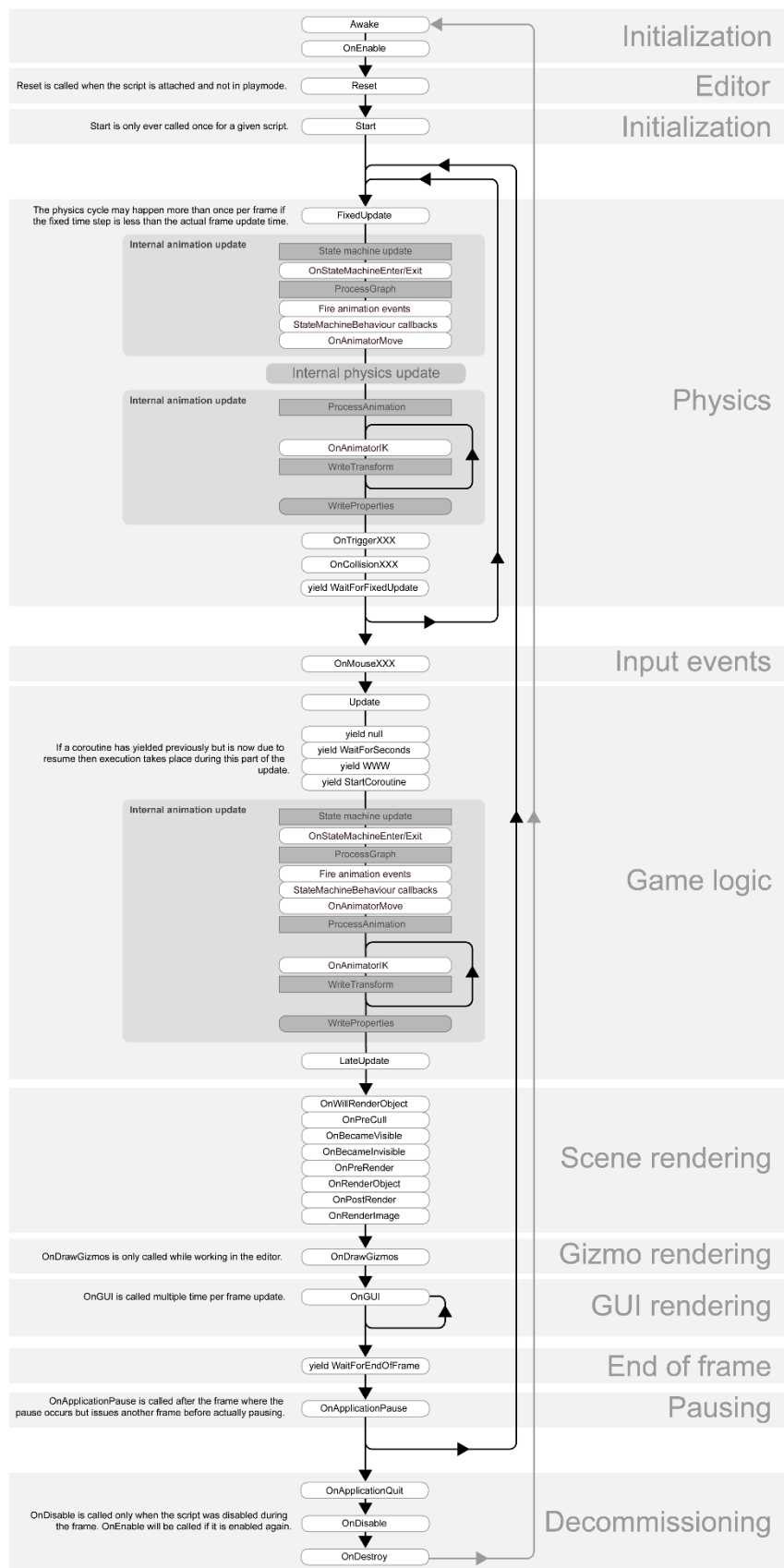
### 1.8.3.3 EVENT FUNCTIONS EN EXECUTION ORDER

De Start() en Update() zijn 'event functions'. Het zijn geen gewone methodes zoals je ze kent (waar ze linear worden uitgevoerd in de programmacode). Event functions zijn onderhevig aan Unity zelf: Unity roept de functies op. Wanneer een welbepaalde event function uitgevoerd werd, wordt de controle teruggegeven aan Unity waarop het een nieuwe (andere) event function oproept.

Concreet komt het er dus op neer dat Unity kiest om éérst de Start() event function uit te voeren. Wanneer deze uitgevoerd werd, krijgt Unity terug de touwtjes in handen. Vervolgens wordt de Update() event function door Unity gecalled.

Er zijn echter veel meer event functions beschikbaar. Een volledige waslijst vind je terug op de documentatiepagina van de MonoBehaviour klasse: <https://docs.unity3d.com/ScriptReference/MonoBehaviour.html>

Het is echter van belang om te weten dat de event functions niet zomaar willekeurig worden gecalled. Unity gebruikt een vaste volgorde waarin event functions uitgevoerd worden. Naast een schematische voorstelling van die volgorde.



#### 1.8.3.4 SENDMESSAGE()

Naast de event functions (die automatisch in een bepaalde volgorde worden gecalled), kent Unity een systeem om methodes op een gelijkaardige manier te triggeren.

Aanschouw volgend codevoorbeeld:

```
using UnityEngine;

public class Example : MonoBehaviour
{
    void Start()
    {
        gameObject.SendMessage("ApplyDamage", 5.0);
    }
}

public class Example2 : MonoBehaviour
{
    public void ApplyDamage(float damage)
    {
        print(damage);
    }
}
```

Een script is steeds gekoppeld aan een GameObject en een GameObject kan overigens meerdere scripts hebben. Via de gameObject property is het mogelijk om via de .SendMessage("MethodeNaam", params) methode een andere methode uit te voeren. Belangrijk om hierbij op te merken is dat indien er meerdere scripts en/of GameObjects een ApplyDamage methode hebben, deze allemaal uitgevoerd zullen worden.

In het infoscherm van de Player Input Component lezen we eerder een klein infotekstje:

*"Will SendMessage() to GameObject: OnDeviceLost, **OnMove**, [...]"*.

Dit betekent dus dat de Player Inputs die we net maakten methodenamen definiëren die we kunnen om te reageren op de inputs van de speler. Deze methodes kunnen we noteren in eender welk (gekoppeld) script.

### 1.8.3.5 VECTOREN

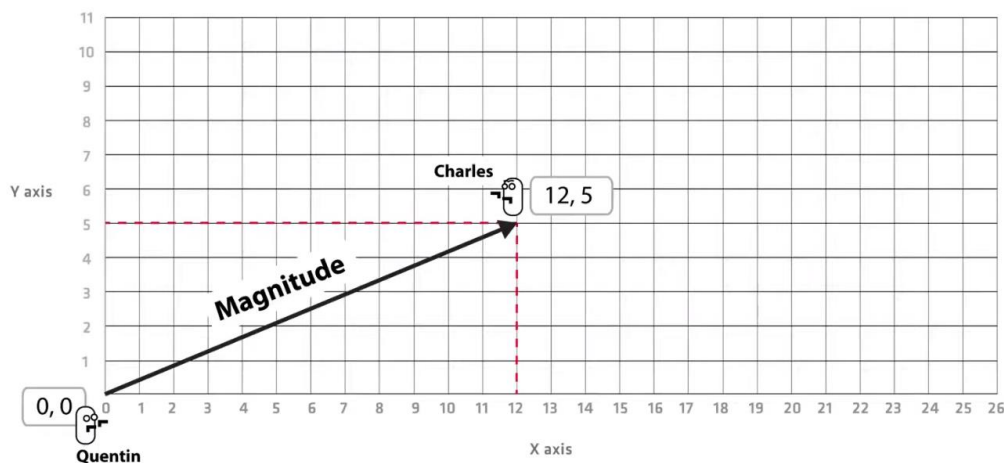
Vectoren zijn een essentieel wiskundig begrip om een richting en een afstand/*magnitude* aan te geven. Ze worden in game development gebruikt om belangrijke eigenschappen zoals de positie van een karakter, de snelheid waarmee iets beweegt of de afstand tussen twee objecten te beschrijven.

Unity kent Vectoren voor 2D, 3D en 4D-space. Het zijn structs die respectievelijk de naam Vector2, Vector3 en Vector4 hebben.

Om écht aan de slag te kunnen met Unity, is het aangeraden om het concept van vectoren volledig door te nemen en te begrijpen. Wat betreft dit cursusvoorbeeld, houden we het echter bij een oppervlakkige uitleg, beginnend in 2D.

#### DE LENGTE OF AFSTAND TUSSEN TWEE PUNTEN AFLEIDEN

Een vector is een lijn die getrokken wordt tussen twee punten. Deze wordt steeds getrokken vanuit een **origine (X:0, Y:0 in het schema)** naar eender welke positie in de 2D-ruimte, (X: 15, Y:5 in het schema). In het schema hieronder vergelijken de afstand tussen twee objecten, Quentin en Charles. Door het kennen van de twee punten (de positie van Quentin en Charles) kunnen afleiden wat de afstand tussen hen beide is. Wanneer we spreken over de afstand tussen twee punten, dan benoemen we dit ook als de *magnitude* tussen twee punten.



We kennen dus zowel de positie van Quentin (X: 0 en Y: 0) als de positie van Charles (X: 12, Y: 5). We weten ook dat de wapens die ze in hun handen hebben 12 eenheden ver kunnen schieten. Kunnen ze elkaar neerschieten?

Langs de **X-as** is de afstand dus **12** eenheden, langs de **Y-as** is de afstand **5** eenheden. Wanneer we de afstand willen tussen de twee willen kennen, dan kunnen we dit afleiden aan de hand van de stelling van Pythagoras:

$$\begin{aligned}x^2 + y^2 &= \text{Magnitude}^2 \\ \text{Magnitude} &= \sqrt{x^2 + y^2} \\ \text{Magnitude} &= \sqrt{12^2 + 5^2} \\ \text{Magnitude} &= \sqrt{144 + 25} \\ \text{Magnitude} &= \sqrt{169} \\ \text{Magnitude} &= 13\end{aligned}$$

De afstand tussen Quentin en Charles is 13 eenheden. Ze kunnen maar 12 eenheden ver schieten, dus ze kunnen elkaar niet vermoorden.

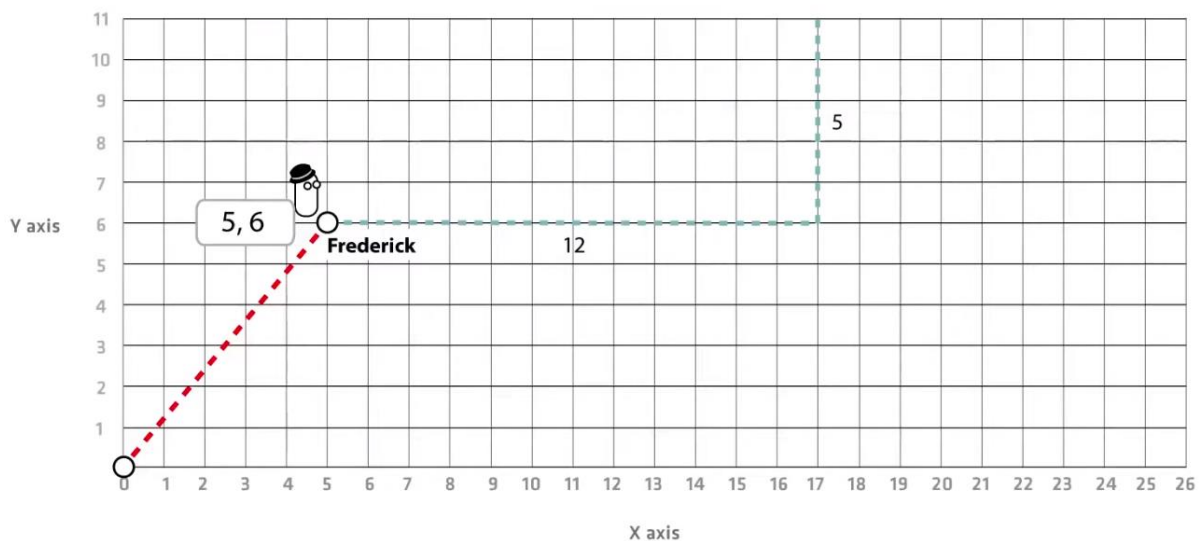
## BEWEGING EN SNELHEID AANGEVEN MET VECTOREN

Hierboven leerden we dat:

- we positie kunnen bepalen relatief t.o.v. het **origin point**: indien we vanuit Quentin (X: 0, Y: 0) 12 eenheden langs de X-as bewegen en 5 eenheden langs de Y-as, dan komen we uit bij Charles.
- we verder uit die gegevens de afstand tussen de twee kunnen bepalen.

We kunnen vectoren echter ook gebruiken om een snelheid aan te geven. Snelheid betekent de verandering van positie overheen een bepaalde tijdsduur. We bespreken dit concept a.d.h.v. een voorbeeld:

Frederick heeft een snelheid van X: 12, Y: 5 per uur. Hij beweegt dus 12 eenheden op de X-as en 5 eenheden op de Y-as per uur.



We vertrekken wederom vanuit een origin point. Dit is de huidige positie van Frederick. Vervolgens tellen we vectorwaarden op bij zijn huidige positie.

$$(\text{huidige } x + \text{vector } x) = \text{nieuwe } x \text{ positie}$$

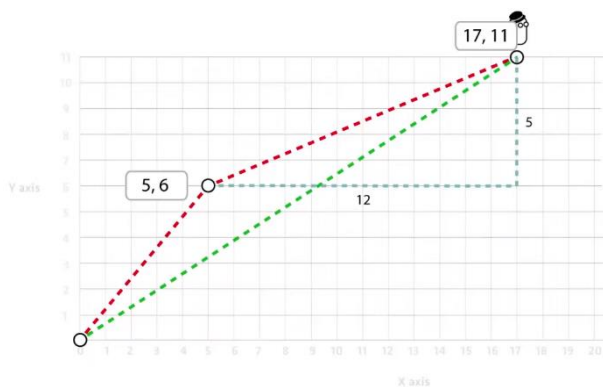
$$\text{nieuwe } x \text{ positie} = 5 + 12$$

$$\text{nieuwe } x \text{ positie} = 17$$

$$(\text{huidige } y + \text{vector } y) = \text{nieuwe } y \text{ positie}$$

$$\text{nieuwe } y \text{ positie} = 6 + 5$$

$$\text{nieuwe } y \text{ positie} = 11$$

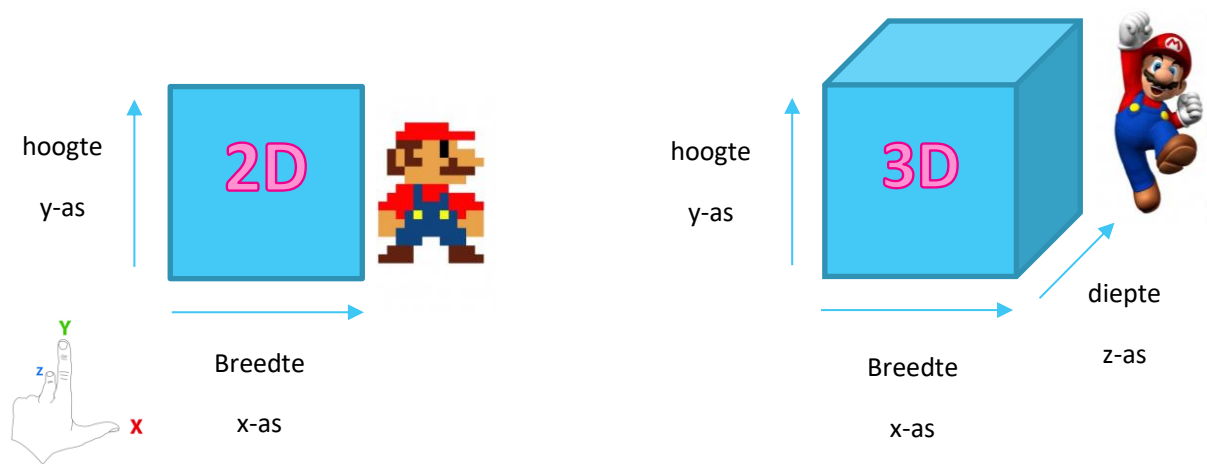


Herinner je je nog het origin point? Dit is het punt van waaruit alles berekend wordt in Unity. Na het verplaatsen van Frederick, heeft hij een nieuwe positie gekregen. De groene lijn stelt de nieuwe positie voor, relatief t.o.v. het origin point (0, 0).



### 1.8.3.6 2D VS. 3D

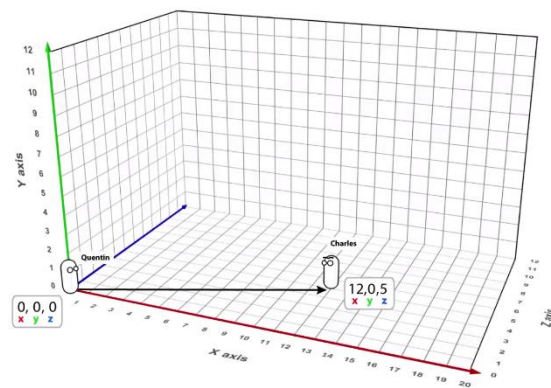
In het voorbeeld hierboven werkten we in 2D. 2D betekent dat we 2 dimensies kennen: de lengte en de hoogte. Cartoons zijn doorgaans 2D. 3D betekent dat we een extra dimensie toevoegen, namelijk de diepte.



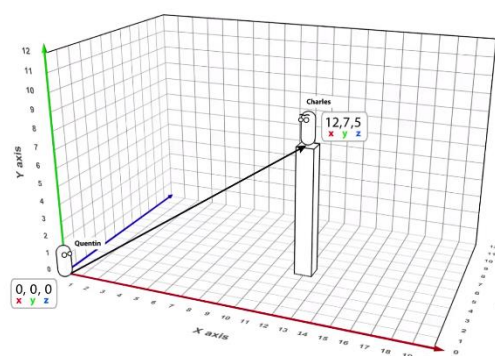
Unity gebruikt een linkshandig coördinatensysteem. Ben je dus in de war welke as welke is? Gebruik dan het truukje met je linkerhand.

We halen Quentin en Charles er weer bij en zetten ze op exact dezelfde posities als de vorige keer. Eerder stonden ze in een 2-dimensioneel veld. We plaatsen ze nu in een 3-dimensioneel veld. Merk op dat de Y-as uit het 2-dimensioneel veld vervangen wordt door de Z-as wanneer we spreken over 3 dimensies.

Quiten staat nu op het orgin point (X: 0, Y: 0, Z: 0) en Charles op X: 12, Y: 0, Z:5.



We kunnen nu ook de derde dimensie gebruiken en Charles op een podium zetten. Zijn positie wordt dan X: 12, Y: 7, Z:5.



Indien we nu willen weten wat de afstand is tussen Quentin en Charles, dan gebruiken we dezelfde formules als die in 2D, maar met de toevoeging van de Z-as:

$$x^2 + y^2 + z^2 = \text{Magnitude}^2$$

...

## 1.9 DE BAL GAAT AAN HET ROLLEN ...

### 1.9.1 DE BAL BEWEGEN

Open de PlayerController script en voeg volgende code toe:

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

public class PlayerController : MonoBehaviour
{
    private Rigidbody playerBall;
    private float movementX;
    private float movementY;

    // Start is called before the first frame update
    void Start()
    {
        playerBall = GetComponent<Rigidbody>();
    }

    // Update is called once per frame
    void Update()
    {
    }

    void OnMove(InputValue movementValue)
    {
        Vector2 movementVector = movementValue.Get<Vector2>();
        movementX = movementVector.x;
        movementY = movementVector.y;
    }

    void FixedUpdate()
    {
        Vector3 movement = new Vector3(movementX, 0.0f, movementY);
        playerBall.AddForce(movement);
    }
}
```

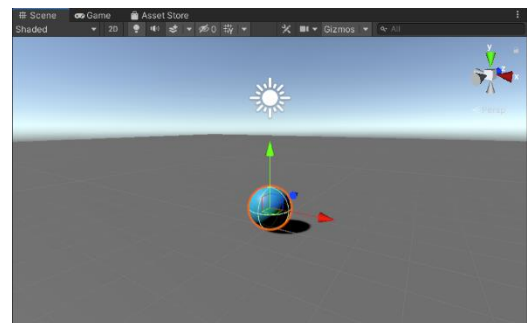
De Start() event function wordt al eerste gecalled. Hier wordt het Rigidbody component dat aan het script gekoppeld is (en ervoor zorgde dat het kan intrageren met de fysics engine) opgevraagd wordt. Dit is dus het Rigidbody component van de bal.

De tweede methode is de OnMove methode. Deze wordt automatisch gecalled via de .SendMessage()-methode die gekoppeld werd voor de PlayerController. Wanneer de speler op de (default ingesteld) toetsen drukt. We kunnen alvast de pijltjestoetsen of WASD-toetsen gebruiken om te bewegen.

De OnMove methode ontvangt een InputValue object, waaraan we op zijn beurt kunnen vragen om een Vector2 object terug te geven. De x- en y-waardes van deze vector bewaren we als privaat veld binnen het script (movementX en movementY). Merk op dat deze van het type float zijn; een float zorgt voor meer precisie dan een int. Wanneer we straks in 3D-space willen bewegen hebben we namelijk nood aan een Vector3: we zetten straks eigenhandig de vector om van Vector2 naar Vector3.

Neem het schema van de execution order er bij ter naslag. Wanneer we physics willen toepassen op een GameObject, dan worden de nieuwe physics eigenschappen toegekend tijdens de FixedUpdate() event function. Het is dan maar logisch om physics gebonden logica in deze methode uit te voeren. Hier maken we op basis van de inputdata zijn Vector2 een nieuw Vector3 object aan met dezelfde waardes. We willen tenslotte bewegen in 3D-space en niet in 2D-space.

We willen bewegen van punt A naar punt B (herinner je Quentin en Charles ...). Die beweging kan voorgesteld worden m.b.v. van een (gepaste) vector. Het GameObject kunnen we nu laten bewegen door op zijn gekoppelde Rigidbody de .AddForce()-methode uit te voeren met een specifieke vector. De vector die we meegeven geeft dus aan naar waar we willen bewegen. De bal zal dus bewegen in de aangegeven richting. Merk ook op dat de bal zijn positie in het coördinatenstelsel kent (zie figuur).



Sla je script op, ga terug naar Unity en druk op de Play-knop. Druk op de pijltjestoetsen om de bal te laten rollen.

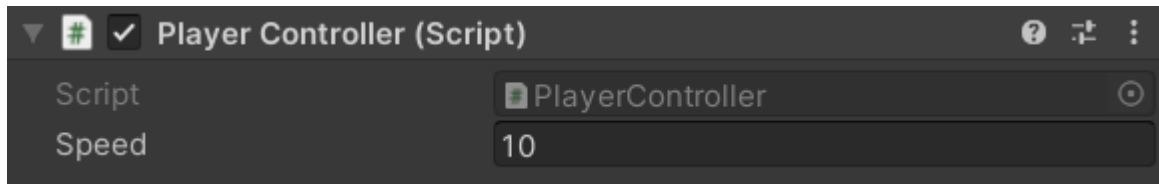
### 1.9.2 DE SNELHEID VAN DE BAL CONTROLLEREN

Wanneer een veld als public gedeclareerd wordt, dan wordt het beschikbaar in de Inspector. Voeg in het PlayerController script een het volgende toe:

```
public class PlayerController : MonoBehaviour
{
    public float speed = 0;
    private Rigidbody playerBall;
    private float movementX;
    private float movementY;
    ...
    void FixedUpdate()
    {
        Vector3 movement = new Vector3(movementX, 0.0f, movementY);
        playerBall.AddForce(movement * speed);
    }
}
```

In de code vergroten we simpelweg de waarde van de vector(en) waar we naartoe willen bewegen a.d.h.v. de variabele speed. We voegen dus meer force toe. Door de speed variable public te declareren wordt het toegankelijk in de inspector onder het Player Controller component om het daar in te stellen. De speed value is nu in Unity terminologie *exposed*.

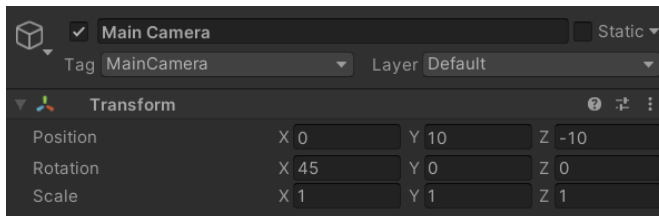
Ga naar het Inspectorscherm van de bal en stel bij het PlayerController script de speed value in op 10:



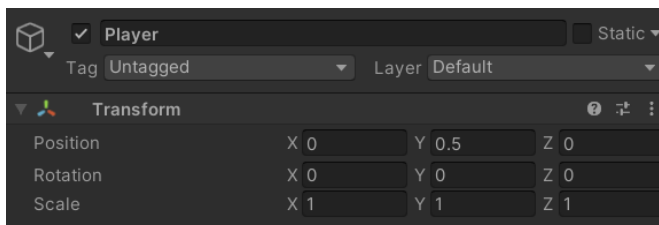
Test je game nog eens uit: de bal beweegt nu veel vlotter! Niet tevreden met het resultaat? Je kan nu gemakkelijk de snelheid tweaken in de inspector!

### 1.9.3 DE CAMERA DE BAL LATEN VOLGEN

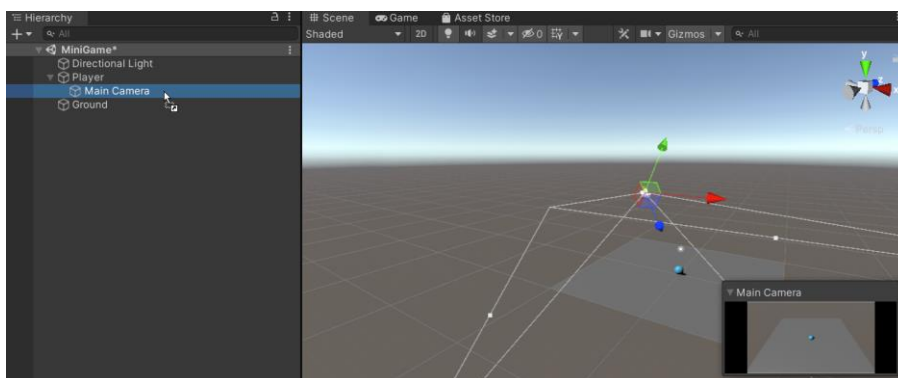
We beginnen met het instellen van een typische Third-person setup van de camera, waarin we in een hoek van 45 graden op de speler neerkijken. Stel de camera in op de volgende waarden:



Ter volledigheid hier ook de Transform eigenschappen van de speler:

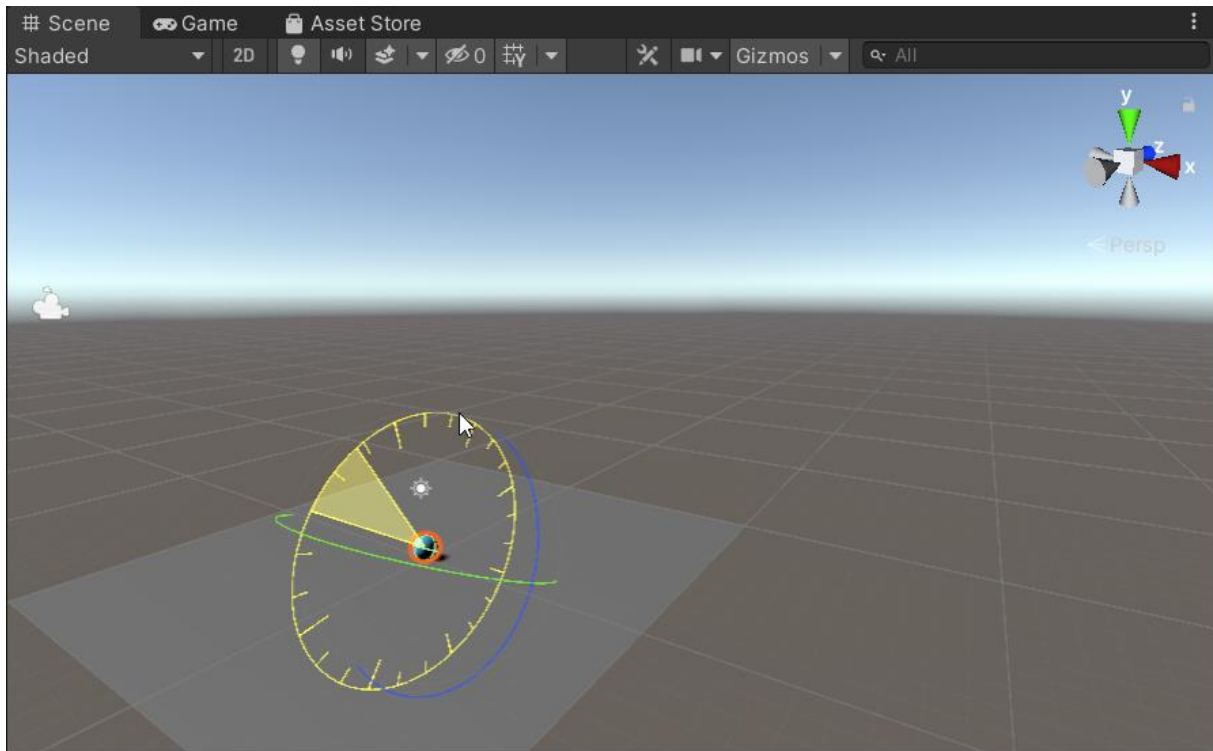


Wanneer we de camera de bal willen laten volgen, dan wordt de Camera afhankelijk van de positie van de bal. Op zich is dit een gemakkelijke klus: je sleep in het *Hierarchy* venster de Camera op de Player en je bent klaar:

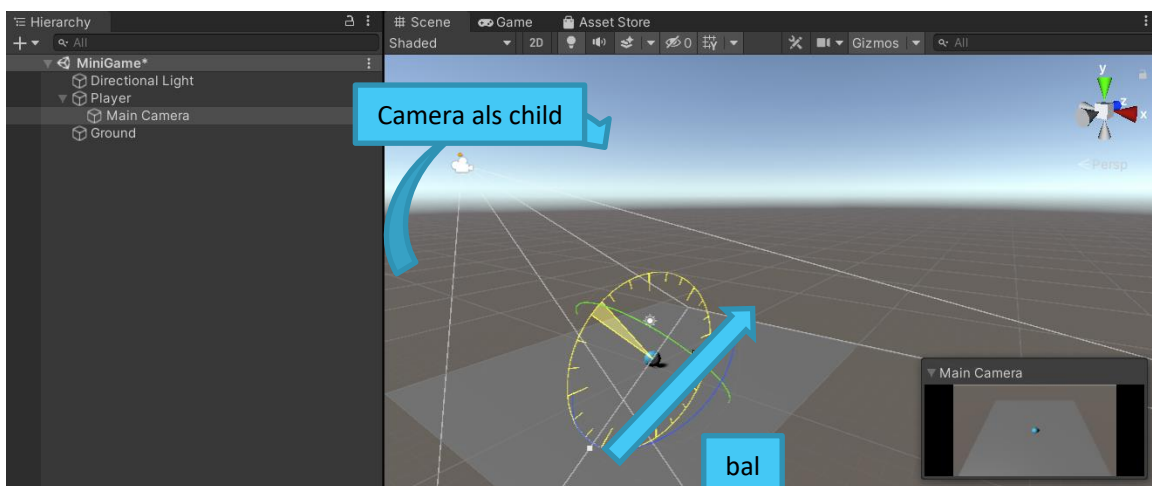


Probeer het spel echter eens uit. We verkrijgen niet het gewenste effect: de camera tolt rond de bal. Dit komt omdat de transform (hier de rotation) eigenschappen zowel op de bal als op de camera toegepast worden. De bal rolt dus vooruit langs de (x-) as, maar de camera doet dit òòk. Dat is niet wat we willen; we willen dat de camera de speler volgt alsof hij er als een drone achterna vliegt.

Deze figuur schets het probleem nogmaals:



Wanneer de bal vooruit rolt, dan roteert de bal lang de (gele) x-as vooruit. Dat roteren gebeurt echter ook op camera wanneer we het koppelen als child:



Op het bovenstaande screenshot heeft dus zowel de bal naar voren gerold, terwijl de camera mee ronddraaide. Reden: wanneer de bal rolt, wordt de 'rotation' property aangepast, maar deze wordt ook tegepast op de camera.

**Opmerking:** mocht de bal niet rollen, dan hadden we dit probleem niet! Een normale speler roteert namelijk niet om te bewegen ...

Dit probleem kunnen we oplossen door een eigen script te schrijven. We schrijven een eigen script waarin we enkel de positie aanpassen en niet de rotatie.

Sleep eerst de Main Camera weer los van de Player zodat het geen child meer is. Maak vervolgens in de *Scripts* map een nieuw script en geef het de naam *CameraController*. Voeg vervolgens aan de Camera een nieuw Component toe en koppel de CameraController.

Schrijf er de volgende code:

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

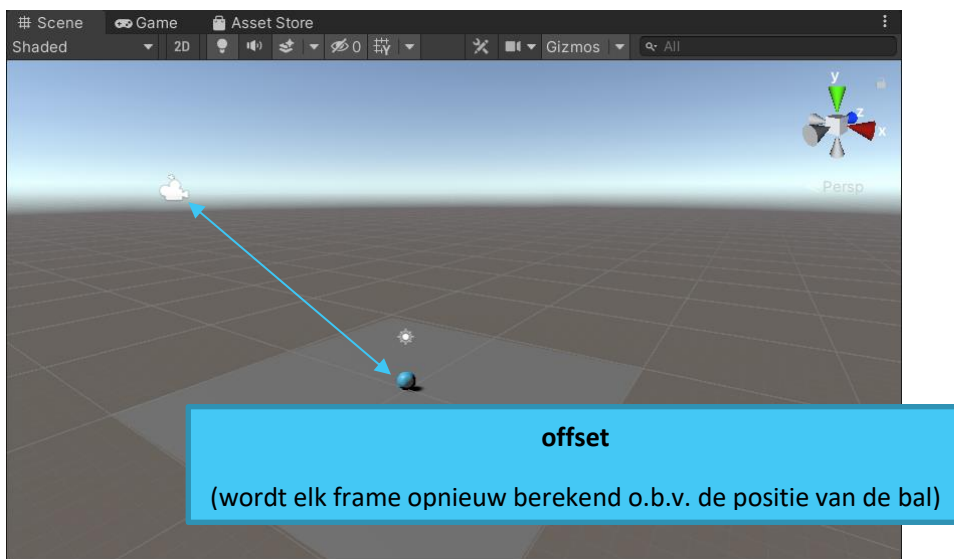
public class CameraController : MonoBehaviour
{
    public GameObject player;
    private Vector3 offset;

    void Start()
    {
        offset = transform.position - player.transform.position;
    }

    void Update()
    {
        transform.position = player.transform.position + offset;
    }
}
```

We houden bij:

- Een **public** referentie naar het GameObject van de speler/bal bij
- Een Vector3 met de naam offset. Dit is de lijn die we trekken vanuit de camera naar de bal om de afstand tussen de twee te kunnen bepalen.



In de Start()-methode bepalen we de afstand tussen de camera en de bal? Deze wordt voorgesteld door een Vector3. We voeren dit uit in de Start() event function omdat we dit onmiddellijk willen toepassen. De afstand tussen de speler en de bal is hetzelfde als de positie van de camera aftrekken van de positie van de speler.

Wanneer de speler vooruit rolt, dan krijgt hij een nieuwe positie. De positie verandert frame per frame, dus in de Update() event function kunnen we frame per frame de positie van de camera aanpassen volgens de nieuwe positie van de bal. Dit doen we door de speler zijn positie op te tellen met de offset. Dit is de nieuwe positie van de camera (voor dat frame).

Elk script in Unity heeft echter een Update() event function. In ons voorbeeld heeft zowel de bal als de camera bijvoorbeeld deze functie. We kennen dan wel de execution order van alle methodes, maar wat we niet weten is welk script eerst aan bod zal komen. Eerst de Update() van de bal? Of eerst de Update() camera?

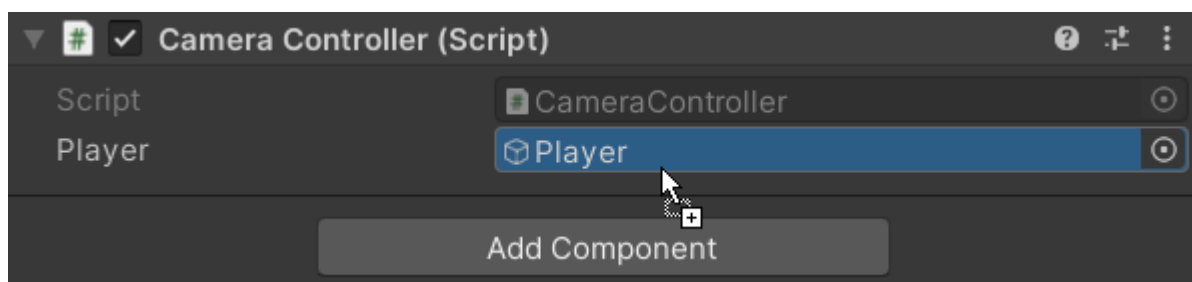
Dit probleem kunnen we oplossen door de code te schrijven in de LateUpdate() event function i.p.v. in de Update() function. Deze wordt pas uitgevoerd na alle Updates() (raadpleeg nogmaals het schema indien nodig):

```
void LateUpdate()  
{  
    transform.position = player.transform.position + offset;  
}
```

Door de LateUpdate() methode aan te passen zijn we dus zeker dat alle GameObject op hun juiste plaats staan vooraleer we eigenhandig aanpassingen maken aan de camera. De speler moet nu dus eerst bewogen hebben vooraleer we de camerapositie aanpassen.

Nu rest ons nog enkel de speler en aan de Camera te koppelen. Herinner je je dat we het GameObject public zetten? Dat is omdat we in de Inspector de speler zouden kunnen vasthangen aan de Camera.

Zet de inspector van de camera open en sleep nu Player (uit de *hierarchy*) in het Camera Controller component's Player-veld:



Test je game nogmaals uit. De camera volgt nu de speler zonder te roteren.

## 1.10 HET SPEELVELD AFBAKENEN

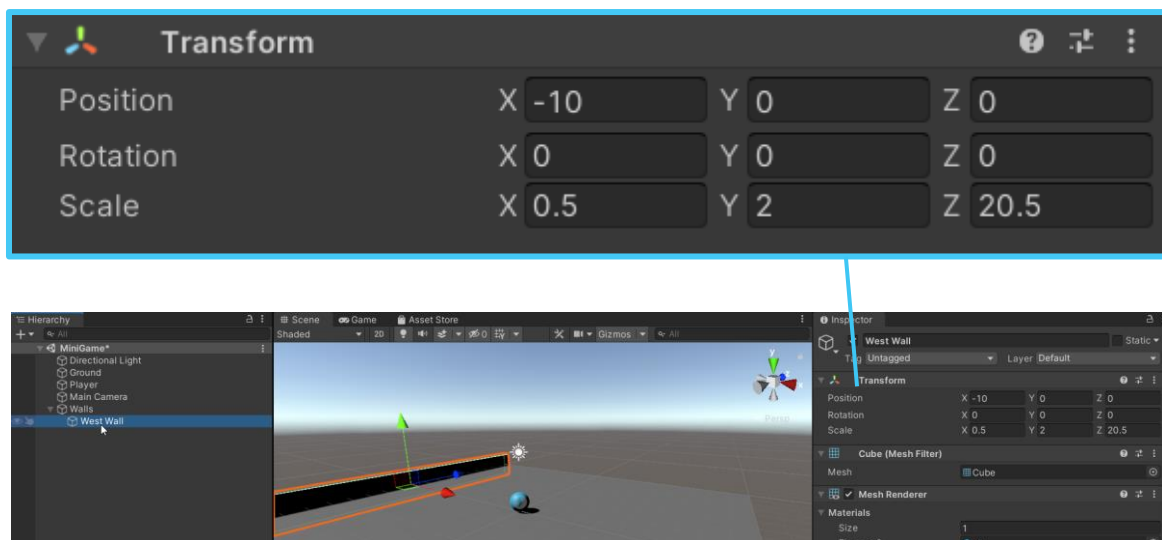
De bal kan op dit moment van de rand van het spel vallen. Laat ons dit eerst oplossen.

Maak een nieuw GameObject in de *hierarchy* en noem het *Walls*. Doet ik via *RMB > Create Empty*. We zullen 4 muren toevoegen en deze bundelen onder dit GameObject.

Maak nu een 3D GameObject via *RMB > 3D Object > Cube* en noem het '*West Wall*'. We willen echter dat de transform properties ingesteld worden op hun default values en dat het element in het origin point komt te staan. Dit kan je doen door te rechtsklikken op de transform titel in de inspector en vervolgens te kiezen voor '*reset*'. Sleep nu de West Wall op het Walls GameObject zodat het een child ervan wordt.

*Tip: door in de hierarchy een GameObject te selecteren kunnen we er snel naartoe zoomen door op F te drukken.*

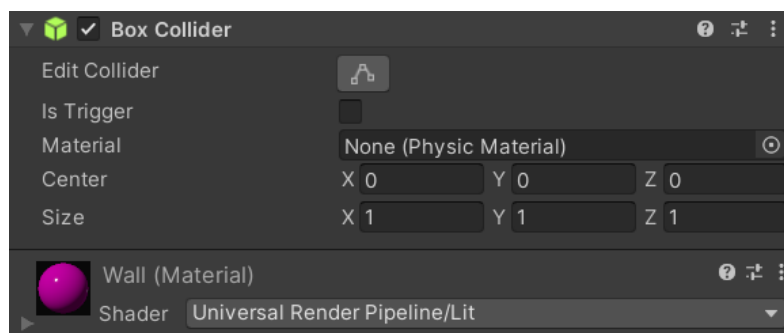
We passen nu de scale en de position aan om de kubus te veranderen van vorm en plaats:



Herhaal deze stappen voor de East, North en South wall. Voeg indien gewenst een Material toe voor de muren.

Om niet telkens de muur opnieuw te moeten maken kan je rechtsklikken op de West Wall en kiezen voor '*Duplicate*'. Pas vervolgens de *transform properties* aan. Let erop dat je alles onderbrengt onder het Wall GameObject.

Test wanneer je klaar bent je game, volledig uitgerust met muren, uit. Merk op dat je tegen de muur botst en er niet door rolt. Dit komt omdat Cubes standaard een box collider hebben:



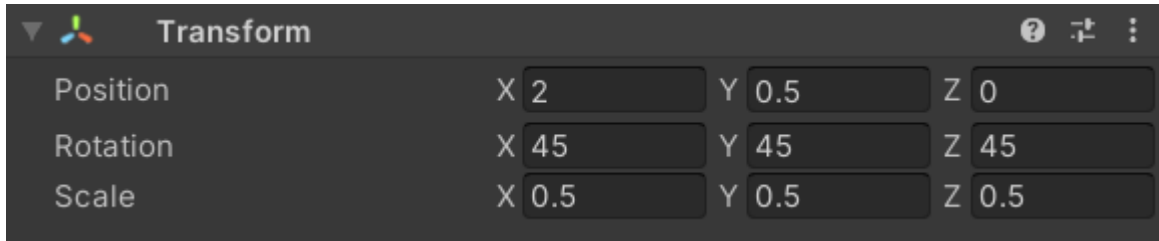


## 1.11 GAMEPLAY TOEVOEGEN

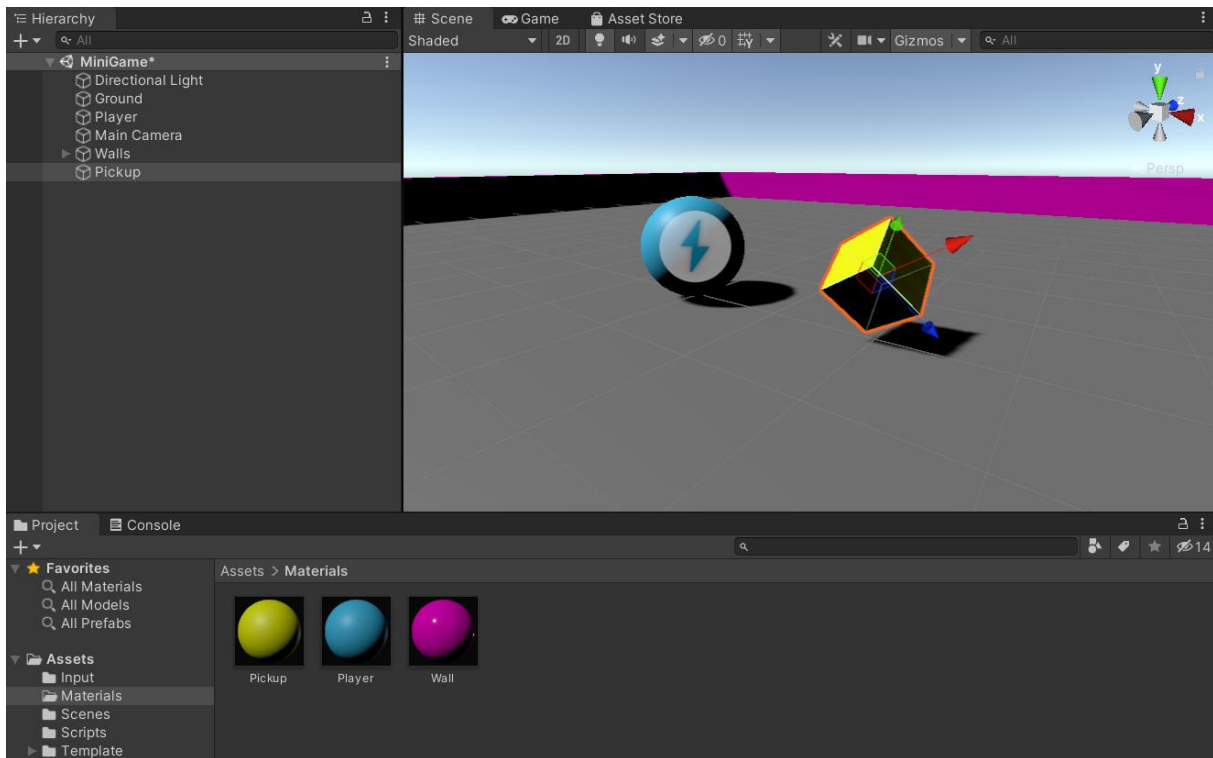
### 1.11.1 COLLECTABLE ITEMS

#### 1.11.1.1 EEN COLLECTABLE ITEM MAKEN

Voeg een nieuwe Cube toe en geef het de naam 'Pickup'. Geef het eventueel een eigen Material en stel de transform properties als volgt in:



Je game ziet er nu ongeveer zo uit:



### 1.11.1.2 EEN SCRIPT VOOR HET COLLECTABLE ITEM

Een gewone, stilstaande en zwevende, kubus is nogal saai en trekt niet onmiddellijk de aandacht van een speler. Om dit op te lossen zullen we ervoor zorgen dat het collectable item roteert. Hiervoor moeten we een eigen script schrijven.

Deze keer maken we het script aan via een andere weg. Ga naar het Pickup GameObject en voeg een component toe. Tik 'Rotator' in. Er wordt voorgesteld om een script aan te maken. Druk Enter om te bevestigen. Wanneer we op deze manier een script toevoegen, dan wordt het toegevoegd aan de rootfolder (Assets). Versleep het bestand naar de Scripts map georganiseerd te blijven werken.

Deze keer kunnen we de Update() event function gebruiken. Deze function wordt elk frame gecalled. We kunnen ditmaal wél deze functie gebruiken omdat we geen forces toepassen op het GameObject.

Open het script en voeg volgende code toe:

```
using UnityEngine;

public class Rotator : MonoBehaviour
{
    void Update()
    {
        transform.Rotate(new Vector3(15, 30, 45) * Time.deltaTime);
    }
}
```

### ROTAREN

Om te beginnen kunnen we via de .Rotate() methode de transform property van het GameObject aanpassen. Met behulp van een Vector3 geven we mee in welke mate er geroteerd dient te worden langs elke as. Merk op dat die rotatie frame per frame gebeurt. Elk frame gebeurt er dus een rotatie van x: 15, y: 30 en z: 45. Het volgend frame heeft de kubus een nieuwe rotatie waarna dezelfde waardes weer worden toegepast om verder te roteren.

### DELTATIME

De framerate op elke pc is echter anders én afhankelijk van het moment. Indien je pc plots veel werk te voorduren krijgt, dan bestaat de kans dat de framerate zal minderen. Dat brengt echter een ongelukkig probleem met zich mee: de Update() event function wordt frame per frame uitgevoerd. Hierdoor bestaat de kans dat de animatie schokkerig zal verlopen. Gezien de tijdsperiode tussen twee frames variabele is, is het niet interessant om de framerate als houvast te gebruiken. De helperklasse Time biedt hier een oplossing: aan deze klasse kunnen we de deltaTime opvragen. Dit is de tijd verstreken tussen twee frames. Door deze te gebruiken (en te vermenigvuldigen met de Vector3 waardes), zorgen we ervoor dat – afhankelijk van de tijdsduur tussen de frames – de animatie groter of kleiner wordt. Dat zorgt voor een vlotte transitie tussen frames.

**Vermenigvuldigingen met deltaTime zorgt er dus in essentie voor voor dat de rotatie per seconde gebeurt in plaats van per frame. Frames zijn niet onderhevig aan een vast interval; tijd wel.**



### 1.11.1.3 HET COLLECTABLE ITEM HERBRUIKBAAR MAKEN

#### PREFABS

Een prefab is een herbruikbaar asset in Unity. Unity's Prefab systeem laat toe om GameObjects te maken, configureren en ze uiteindelijk te bewaren. Alle components, properties en child-GameObjects worden bewaard. Het is als het ware een blauwdruk van een specifiek GameObject.

Het collectable item is een mooi voorbeeld. Het heeft o.a. het volgende gekoppeld:

- Een eigen Material
- Het rotatiescript
- ...

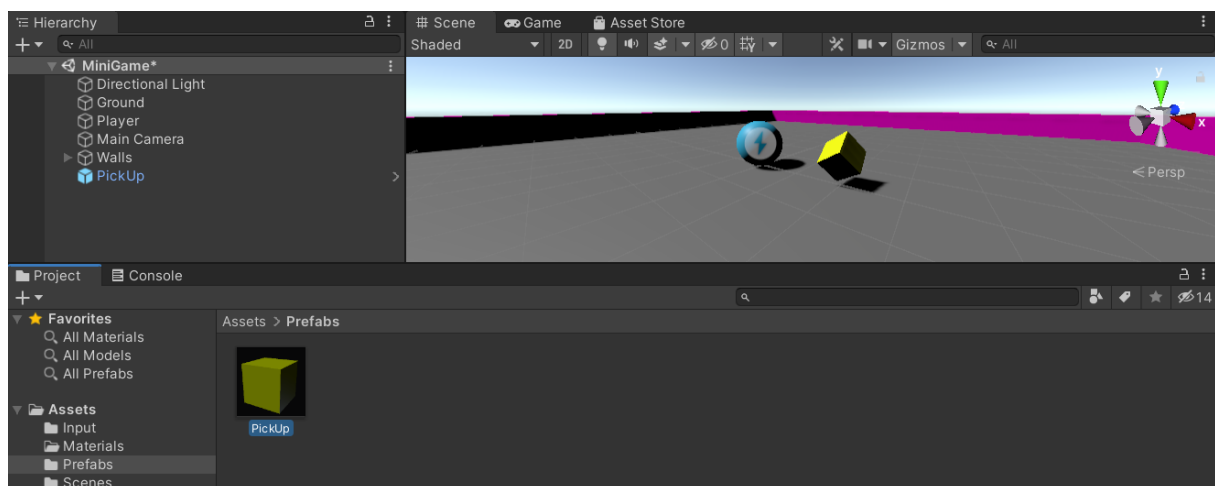
Het gebruiken heeft zo zijn voordelen. Door het collectable item om te bouwen naar een Prefab, kunnen we meerdere collectable items in de Scene (of eender welke andere Scene in het project) plaatsen. Daarnaast wordt het mogelijk om alle instanties aan te passen door het prefab te wijzigen. Neem bijvoorbeeld dat we de Pickup items willen veranderen van kleur. Indien we geen prefabs gebruiken, dan moeten we GameObject per GameObject de kleur aanpassen. Gebruiken we echter een prefab, dan moeten we enkel de kleur van het prefab wijzigen om alle instanties van kleur te veranderen.

Prefabs worden bovendien ook gebruikt wanneer GameObjects run-time geïnstantieerd worden. Voorbeelden zijn:

- Een collectable item die pas tevoorschijn komt onder bepaalde voorwaarden
- Een kanonskogel die afgevuurd wordt vanuit een piratenschip
- ...

#### EEN PREFAB MAKEN

Maak in de Assets map een nieuwe map met de naam *Prefabs*. Sleep nu vanuit het *Hierarchy* venster de het Pickup GameObject naar deze map. Unity maakt automatisch het prefab aan. Merk op dat het Pickup GameObject nu blauw gemarkeerd wordt in het *Hierarchy* venster.

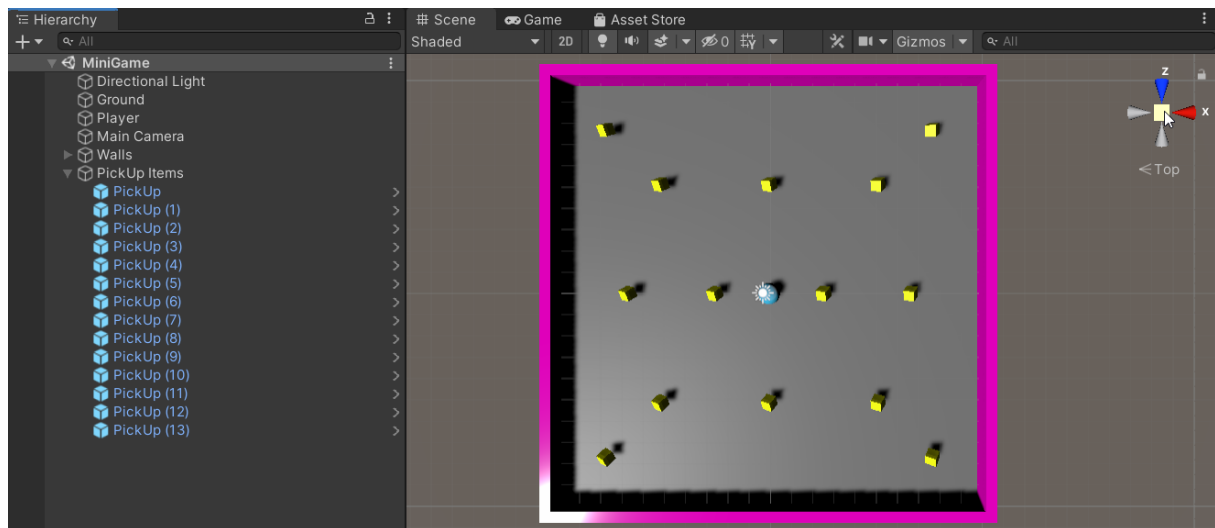


Je kan het prefab individueel bekijken in de Scene view door op het pijltje te klikken (>).

## EEN PREFAB (HER)GEBRUIKEN

Nu we de prefab gedefinieerd hebben kunnen we het eenvoudigweg kopiëren. Tik *RMB > Create Empty* in het *Hierarchy* venster en noem het 'PickUp Items'. Hierin zullen we alle collectable PickUps onderbrengen. 'Reset' eerst de transform eigenschappen ervan en sleep dan het eerder gemaakte PickUp prefab erin zodat het een child van het GameObject wordt. Dupliceer vervolgens het Prefab en aantal keer en plaats ze in het rond op het speelveld door de transform properties van elk prefab te wijzigen.

Je dit eventueel doen vanuit een top view. Doe dit door rechtsboven op het y-as te klikken in het *Scene* venster.



Test de game even uit. We kunnen nu rond de items rollen, maar we botsen er gewoon tegen.

### 1.11.2 COLLISION (BOTSING) DETECTEREN

#### 1.11.2.1 COLLISION DETECTEREN

Wanneer de bal tegen een PickUp item rolt, dan willen we dat we het item kunnen oppikken om punten te verdienen. We moeten dus te weten komen wanneer de bal botst met een PickUp item.

Voeg de volgende code toe aan de PlayerController:

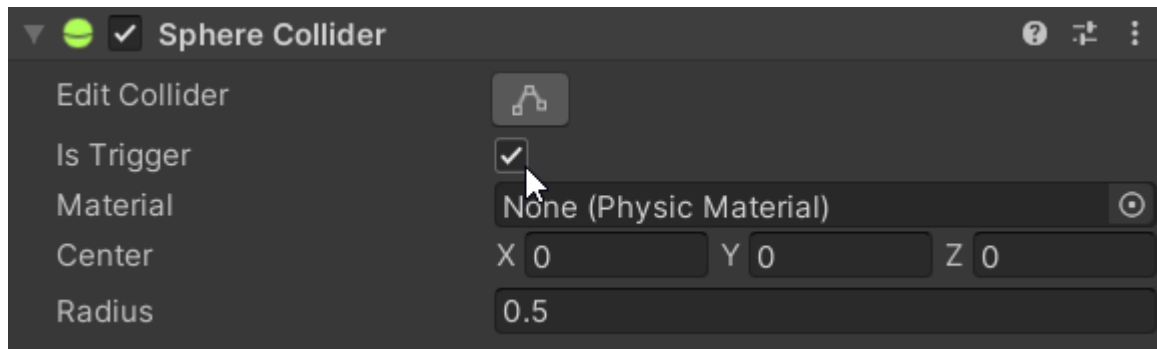
```
public class PlayerController : MonoBehaviour
{
    ...

    void OnTriggerEnter(Collider other)
    {
        other.gameObject.SetActive(false);
    }
}
```

Herinner je je nog de Sphere Collider van onze speler? Deze was verantwoordelijk om te detecteren wanneer het botst met iets. Net zoals de bal hebben ook de PickUp GameObjects een Collider, maar dan een Box Collider.

Wanneer de bal ergens tegen botst dan wordt `OnTriggerEnter(Collider other)` gecalled. De Collider van het GameObject waarmee we botsen, wordt ontvangen. Met bovenstaande code kunnen we het andere GameObject, na botsing, verbergen.

Echter missen we nog één stap. We moeten aan de andere GameObjects nog vertellen dat ze fungeren als 'Trigger' om deze code te laten uitvoeren. Ga naar de bal en stel in de inspector in dat de Sphere Collider dient als Trigger:



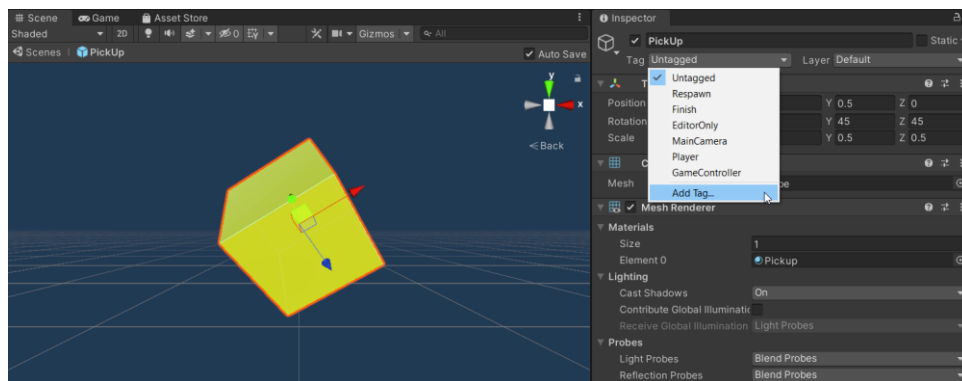
Test het spel nog eens uit. We komen in de problemen! Eender welk object waarmee we botsen wordt nu onzichtbaar gemaakt door de code die we toevoegden. Dat is uiteraard niet wat we wilden! We moeten een manier vinden om enkel de Pickup items onzichtbaar (oppikbaar) te maken ...

**Vink 'Is Trigger' alvast weer uit!**

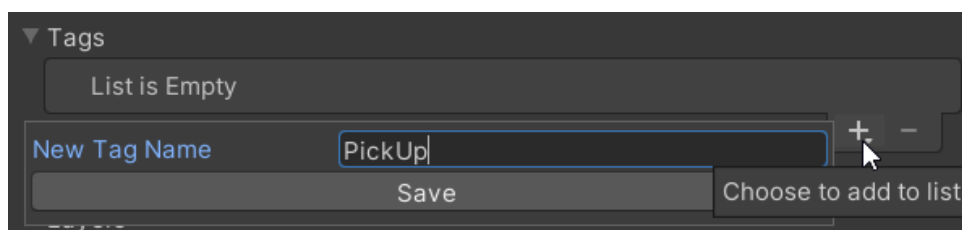
#### 1.11.2.2 UNITY TAG SYSTEM

Het Unity Tag system laat toe om bovenstaande probleem te vermijden. We kunnen aan de prefab een tag koppelen (een string), zodat we kunnen checken of we effectief de code willen uitvoeren bij de botsing. Enkel wanneer het de tag van de prefab heeft, zullen we dit doen.

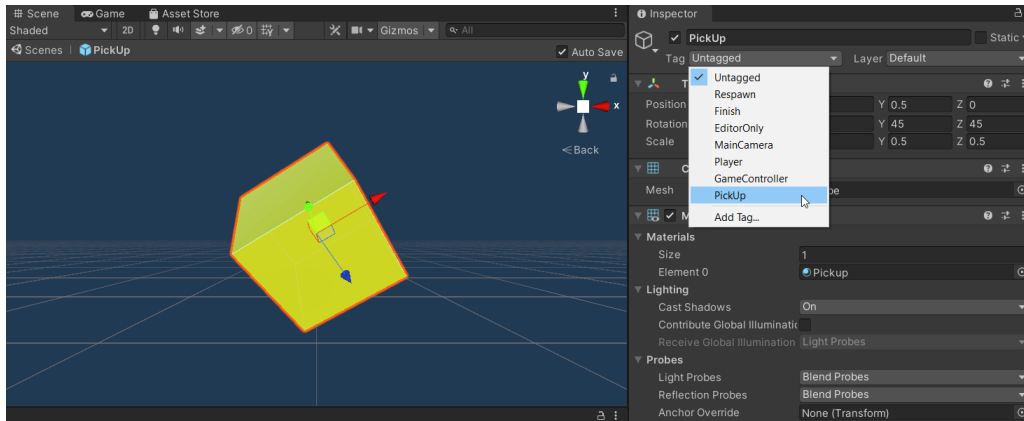
Navigeer in de mappen naar de Pickup Prefab en dubbelklik erop. Voeg in de inspector een tag toe:



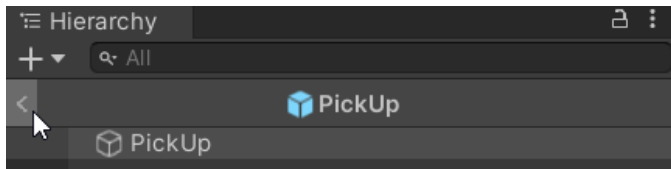
Voeg vervolgens een tag name ("PickUp") toe:



Nu moeten we de zonet aangemaakte tag nog effectief koppelen aan de prefab. Terug in de inspector stel je deze nu in:



Verlaat de Prefab Editor door op < te klikken in het *hierarchy* venster:



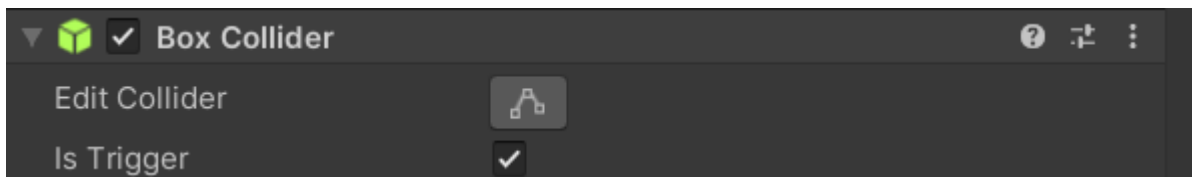
Merk op dat, omdat we een prefab gebruiken, nu alle PickUp GameObjects deze tag gekregen hebben.

In code kunnen we de tag nu als volgt opvragen:

```
public class PlayerController : MonoBehaviour
{
    ...

    void OnTriggerEnter(Collider other)
    {
        if(other.gameObject.CompareTag("PickUp"))
        {
            other.gameObject.SetActive(false);
        }
    }
}
```

Daarnet kozen we ervoor om de bal een trigger te maken. Dat is op zich niet zo'n slim idee: de bal krijgt te veel voor het zeggen. Logischer is dat de andere GameObjects een trigger zijn. Deze keer doen we het dus goed: ga terug naar de Prefab van het PickUp item en stel de Box Collider in als Trigger.



Test het spel nogmaals uit. Je kan nu de PickUp items verzamelen!

### 1.11.2.3 EFFICIËNTE COLLISIE MET EEN RIGIDBODY

Unity maakt een onderscheid tussen statische en dynamische componenten.

Het project mag dan tot nu toe makken werkt dan wel, maar efficiënt is de niet. Het probleem schuilt zich in het feit dat de Pickup items bewegen. Hierdoor moet Unity de Box Collider voor elk item telkens opnieuw berekenen.

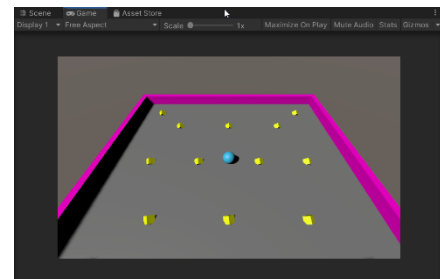
Wanneer een GameObject een Rigidbody heeft, dan wordt het door Unity beschouwd als een **dynamisch** component. Unity behandelt het dan als een object dat gemaakt is om te bewegen. **Statische** componenten zijn dan weer gemaakt om net niet te bewegen.

Momenteel zijn de Pickup objecten als static gedefinieerd, maar ze roteren. Dit werkt wel, maar Unity doet er langer over om telkens de nieuwe posities van de GameObjects te berekenen. Deze (langere) berekening gebeurt trouwens elk frame.

De oplossing is dus eenvoudig: de Pickup items moeten een Rigidbody component krijgen.

Ga in het *Project* venster naar de Pickup prefab en klik erop. Voeg vervolgens een Rigidbody Component toe in de inspector. Sluit het venster previewvenster van de Pickup af (het <-knopje in het *hierarchy* venster) en sleep je spel nog eens. De Pickup items vallen nu los door de grond.

Dit viel eigenlijk te verwachten. Enerzijds zweven de prefabs in de lucht, waardoor ze wel moesten vallen wanneer we een Rigidbody toevoegden (die, herinner je, net instaat voor het toepassen van de wetten van de fysica). Dat ze door de grond vallen is misschien een verrassing, maar ook dat is een probleem dat we al eerder aankaarten. De prefabs fungeren als trigger, waardoor ze niets botsen met het speelveld.

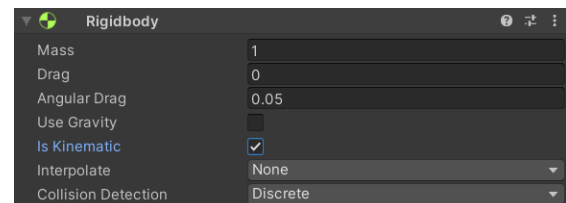


STATISCHE COLLIDER	DYNAMISCHE COLLIDER
beweegt niet	beweegt
heeft een Rigidbody	heeft geen Rigidbody

STANDAARD RIGIDBODY	KINEMATIC RIGIDBODY
Beweegt via forces (cfr. de .AddForce() die we reeds gebruikten)	Beweegt via de transform properties

Ga terug naar de Rigidbody component van het prefab. Een mogelijke oplossing is om de 'Use Gravity' checkbox van het Rigidbody af te vinken, maar dan reageert het nog steeds op fysieke krachten (bijvoorbeeld opzij geduwd worden). Door het prefab 'Kinematic' te maken, zal het niet meer reageren op de fysieke krachten.



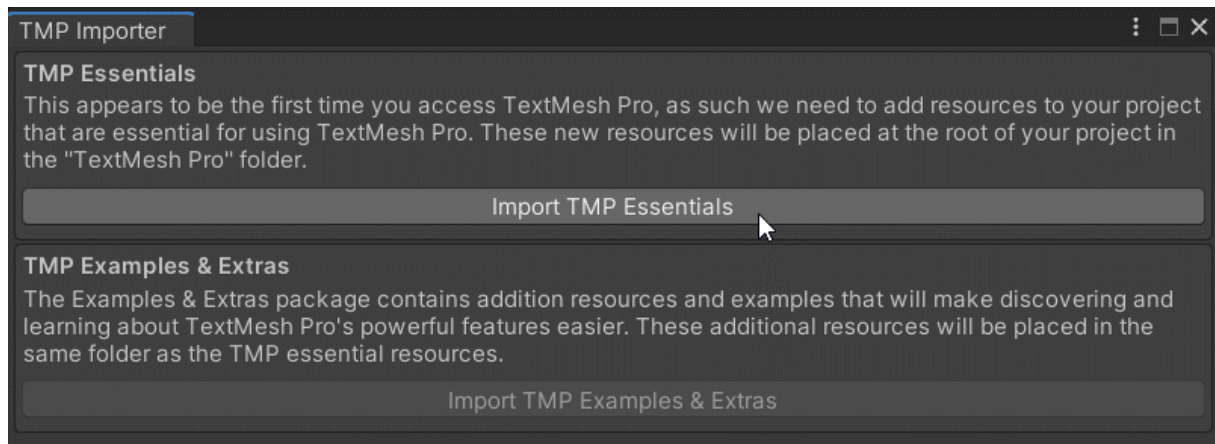
Test je spel nogmaals uit. Het doet exact hetzelfde als voordien, maar op een efficiëntere manier.

## 1.11.3 GUI MAKEN

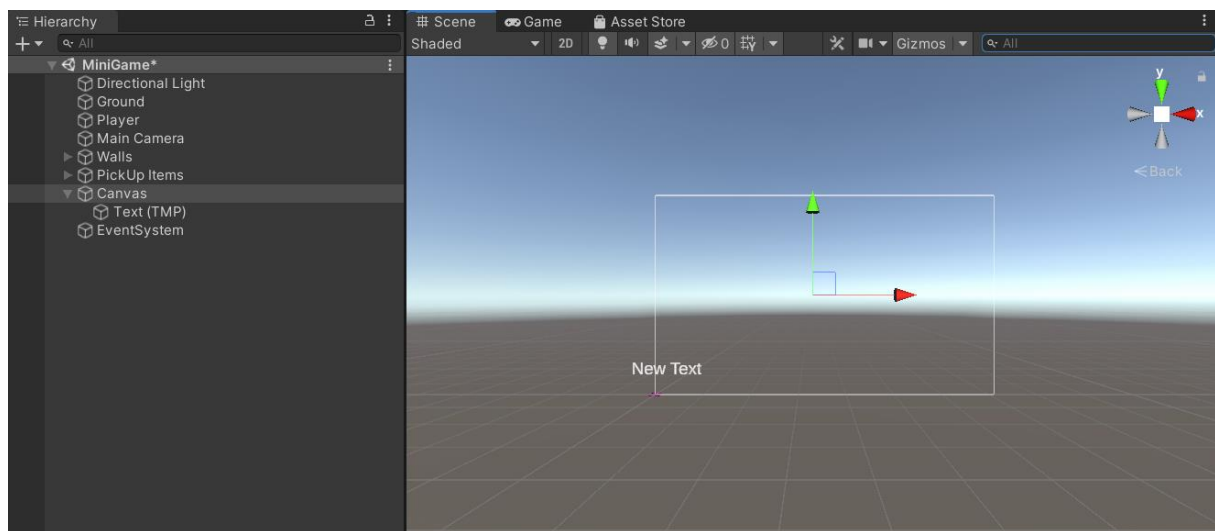
### 1.11.3.1 GUI MAKEN

Tot slot voegen we tekst toe om aan te geven hoeveel PickUp items we verzameld hebben.

Klik *RMB* > *UI* > *Text – Text Mesh Pro* in het *hierarchy* venster. Een dialoogvenster verschijnt. Klik op 'Import TMP Essentials'.

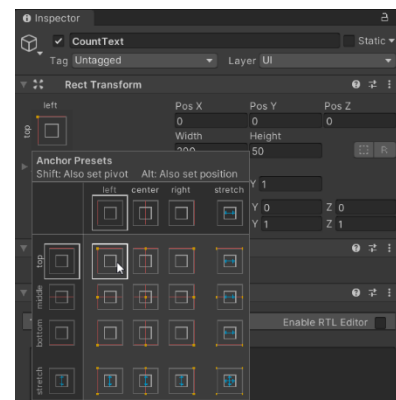


Selecteer het Canvas GameObject en druk F in het Scenevenster. Klik vervolgens in de rechterbovenhoek op de assen om een front-view te krijgen:



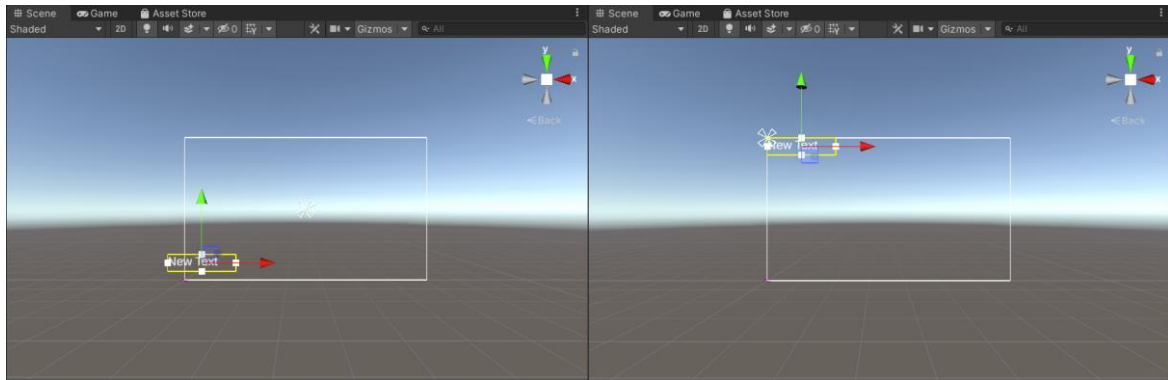
Selecteer vervolgens het *Text (TMP)* object en geef het de naam 'CountText'. In de inspector kunnen we de positie aanpassen door op het volgende icoon hiernaast afbeeld te tikken. Houd echter ook ALT en SHIFT ingedrukt. Dit wijzigt het anchor-punt van het canvas zelf ook.

De tekst wordt namelijk relatief t.o.v. anchorpunt het canvas geplaatst.

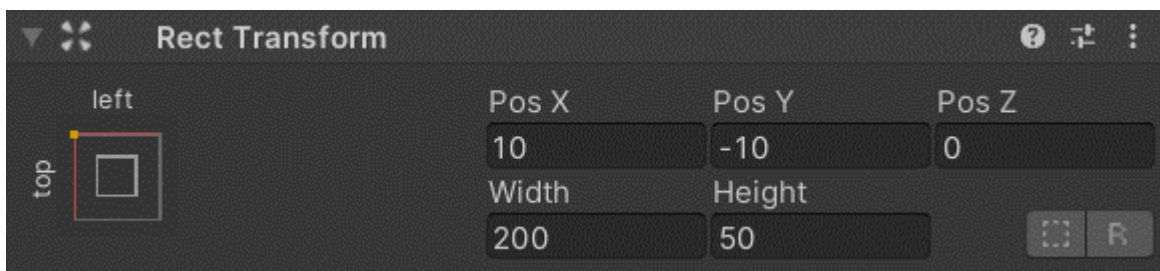




Voor en na (ler erop hoe het anchorpunt van het canvas verplaatst werd naar de linkerbovenhoek):



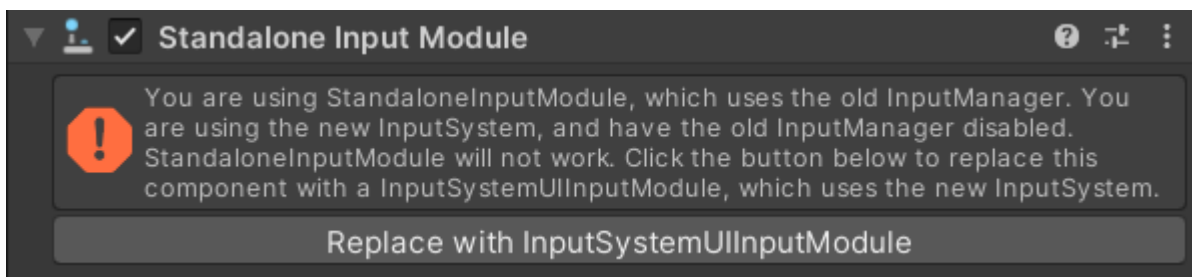
Voorzie nu wat meer marge link en boven om wat ademruimte te voorzien door de Pos X en Pos Y in te stellen:



Voorzie ook een standaard tekst, 'Count Text':



Het Input System werd vernieuwd in Unity. Momenteel gebruikt onze code nog een oude manier van werken. Er rest ons nu nog enkel een extra stap om te bevestigen dat we het nieuwe input system willen gebruiken. Selecteer het EventSystem GameObject en klik op 'Replace with InputSystemUIInputModule' in de inspector:



### 1.11.3.2 DE TEKST AANPASSEN VANUIT HET SCRIPT

Voeg in het PlayerController script de volgende code toe:

```
using TMPro;
using UnityEngine;
using UnityEngine.InputSystem;

public class PlayerController : MonoBehaviour
{
    public float speed = 0;
    public TextMeshProUGUI countText;
    private Rigidbody playerBall;
    private float movementX;
    private float movementY;
    private int count;

    void Start()
    {
        playerBall = GetComponent<Rigidbody>();
        count = 0;
        SetCountText();
    }

    void OnMove(InputValue movementValue)
    {
        Vector2 movementVector = movementValue.Get<Vector2>();
        movementX = movementVector.x;
        movementY = movementVector.y;
    }

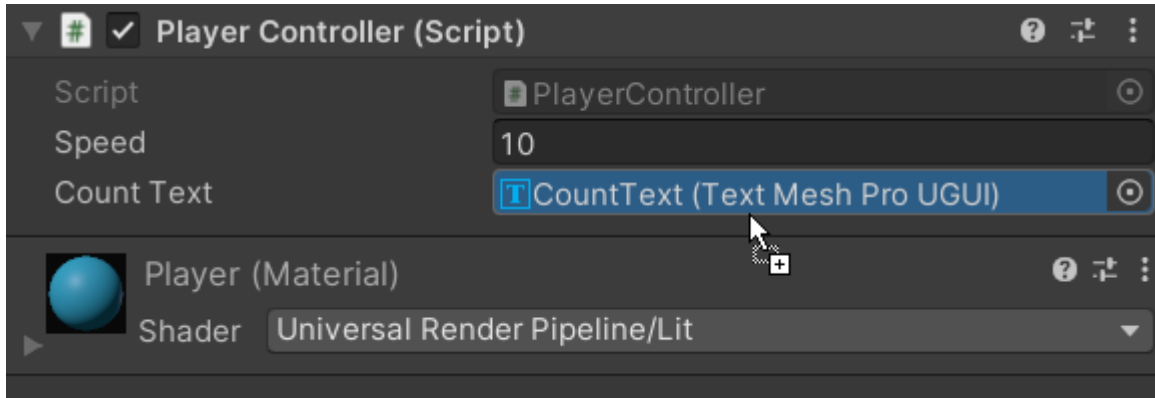
    void FixedUpdate()
    {
        Vector3 movement = new Vector3(movementX, 0.0f, movementY);
        playerBall.AddForce(movement * speed);
    }

    void OnTriggerEnter(Collider other)
    {
        if(other.gameObject.CompareTag("PickUp"))
        {
            other.gameObject.SetActive(false);
            count++;
            SetCountText();
        }
    }

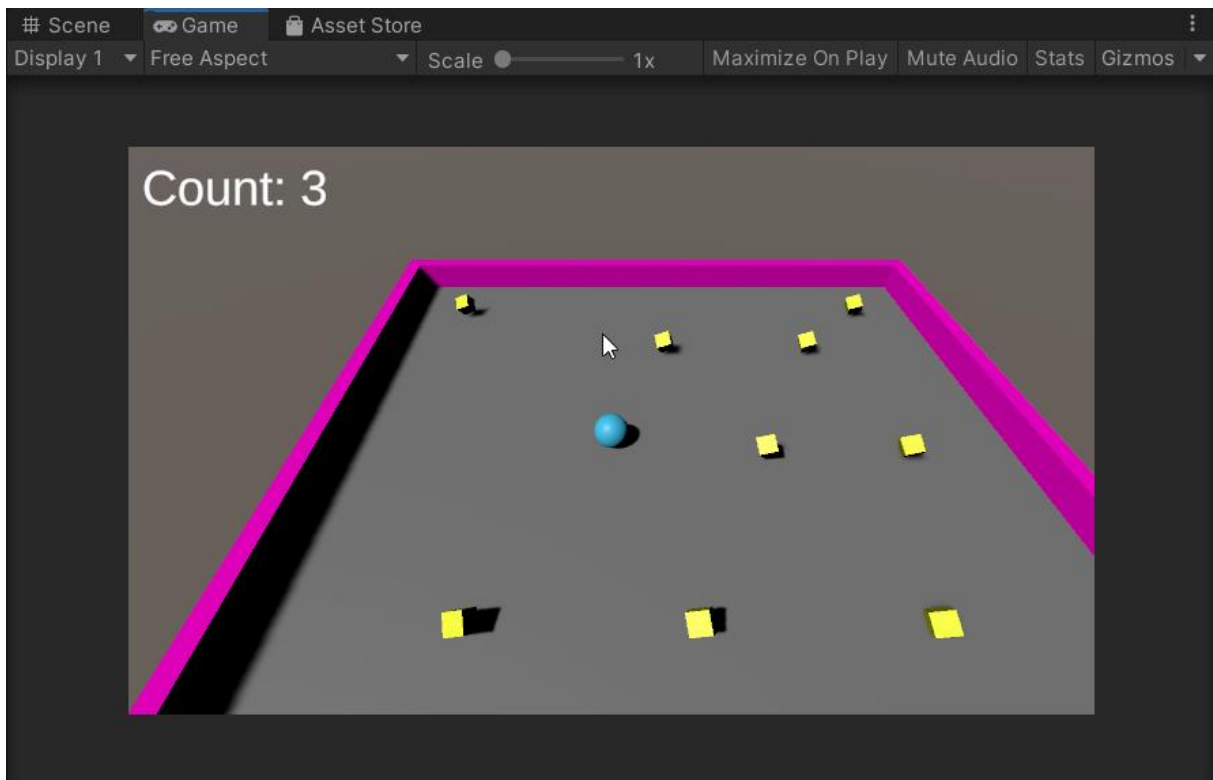
    private void SetCountText()
    {
        countText.text = "Count: " + count;
    }
}
```

In bovenstaande code maakten we een public variabele aan voor de tekst. Deze moeten we nog enkel, zoals eerder, koppelen in de inspector.

Voeg in de Inspector een referentie toe naar het CountText GameObject door CountText uit de *hierarchy* naar het desbetreffende slot te slepen:



Test je game nog eens uit. Er wordt nu bijgehouden hoeveel items je al verzamelde:



#### 1.11.4 OEFENING – WIN MESSAGE

Ga nu zelf aan de slag om een win message te tonen. Hou in het playerscript een referentie bij naar een GameObject. Dit is het GameObject dat de tekst “You win!” dient te tonen. Schrijf code om op een gepast moment deze tekst (on)zichtbaar te maken.

