Laboratorul 3

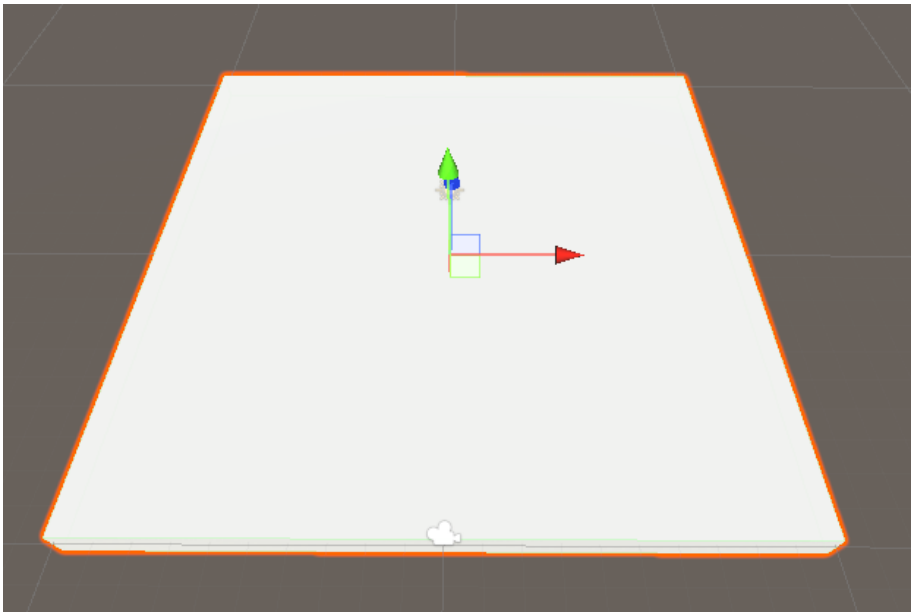
1 Crearea unei încăperi

Vom crea un proiect nou la fel ca cele create în laboratoarele anterioare ($3D\ Core\ Project + URP$).

Cu cunoștințele din laboratoarele precedente putem realiza o cameră (room) în interiorul căreia să se desfășoare jocul pe care îl implementăm.

1.1 Podea

În primul rând vom crea o podea folosind un cub pentru a reprezenta podeaua. Îi vom reseta componenta $Transfrm$, după care vom atribui field-ului $Scale$ valoarea $(20, 1, 20)$ pentru a obține o suprafță care seamănă cu o podea.



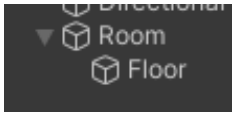
Vom crea un director nou, numit $Materials$ în interiorul căruia vom crea un nou material pe care îl vom folosi pentru podea. Vom numi materialul $Floor$, și îi vom atribui o culoare.

Vom atribui acest material cubului creat anterior.

1.2 Pereți

Următorul pas în realizarea încăperii este să adăugam pereții. Deoarece ne dorim ca încăperea să fie reprezentată de un singur obiect, este nevoie să existe o relație de înfrățire între elementele care alcătuiesc camera (pereți și podea). O soluție ar fi ca podeaua să reprezinte nodul părinte al încăperii, iar pereții să reprezinte nodurile copil. Această abordare este problematică în calculul *Scale*-ului, sau în cazul în care este nevoie de rotații asupra acestor obiecte.

Soluția pe care o vom folosi este următoarea: creăm un nou obiect gol, numit *Room*, căruia îi resetăm componenta Transform, iar apoi facem ca obiectul *Floor* să fie copil al obiectului *Room*.



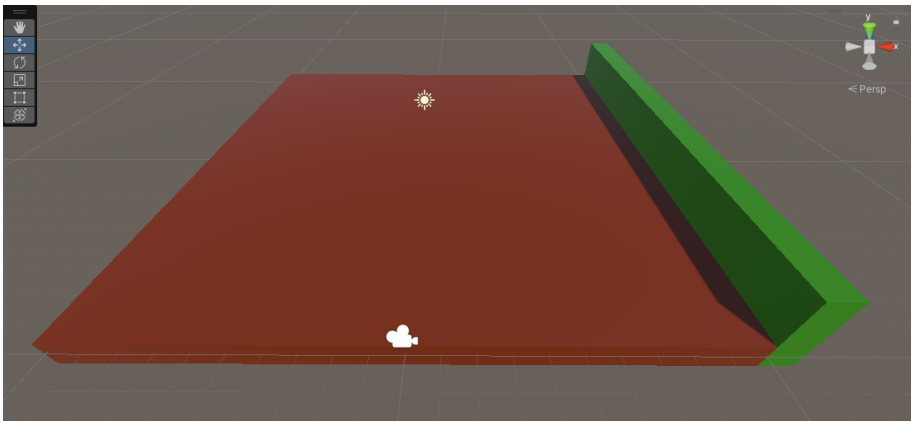
În acest moment putem realiza pereții. Vom crea un nou cub, copil al obiectului Room, căruia îi vom reseta componenta $Transform$. Vom numi acest obiect $Right\ Wall$.

Acest perete va reprezenta peretele din partea dreaptă a încăperii. Pentru a avea o lungime egală cu dimensiunea podelei, componenta Z a field-ului *Scale* va avea valoarea 20. Vom face ca pereții să aibă o înălțime egală cu 3 (valoarea Y a field-ului *Scale* al componentei transform). După ce am definit scara obiectului $(1, 3, 20)$, vom defini poziția acestuia. Pentru a se afla în partea dreaptă a podelei, valoarea componentei X a poziției trebuie să fie egală cu 10.5 (10 (distanța dintre centrul podelei și extremitatea din dreapta) + 5 (jumătate din grosimea peretelui)). Valoarea componentei Y a poziției va fi egală cu 1 (1.5 (jumătate din înălțimea peretelui) - 0.5 (jumătate din grosimea podelei)).

Așadar, componenta $Transform$ a obiectului perete are următoarele valori ale field-urilor:

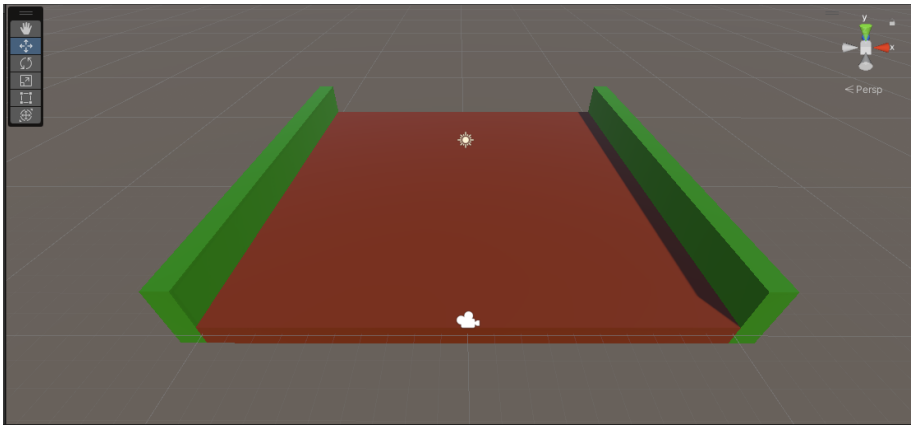
- $Position - (10.5, 1, 0)$
- $Rotation - (0, 0, 0)$
- $Scale - (1, 3, 20)$

Din nou, vom crea un nou material numit $Wall$ pe care îl vom folosi asupra acestui obiect. Vom selecta ce culoare dorim pentru acest material.



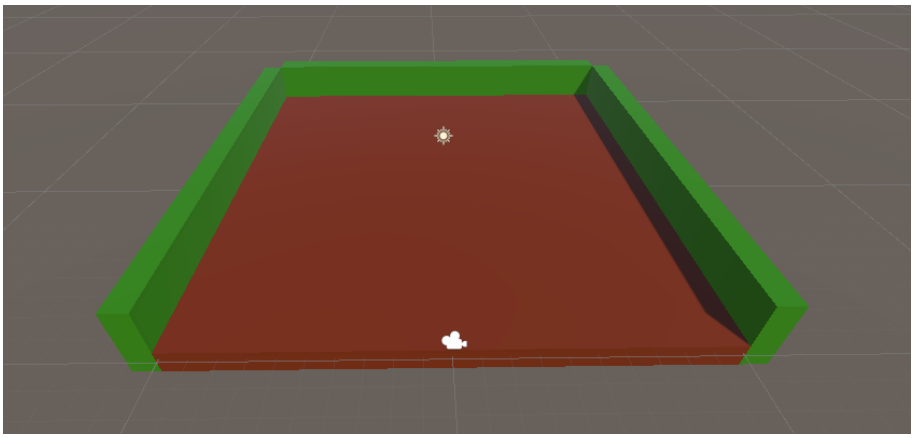
Vom continua cu ceilalți pereți. Pentru a nu mai incepe de la 0 cu crearea celorlalți pereți, ne putem folosi de peretele deja creat. Vom selecta acest obiect, după îl vom duplica (fie $Ctrl + D$, fie *click dreapta pe obiect in Hierarchy/ Duplicate*).

Vom numi obiectul duplicat $Left\ Wall$. Singura proprietate pe care o vom schimba va si poziția acestuia de-alungul axei X. În loc de valoarea 10.5 îi vom atribui valoarea -10.5 pentru a fi poziționat în partea stângă a podelei.

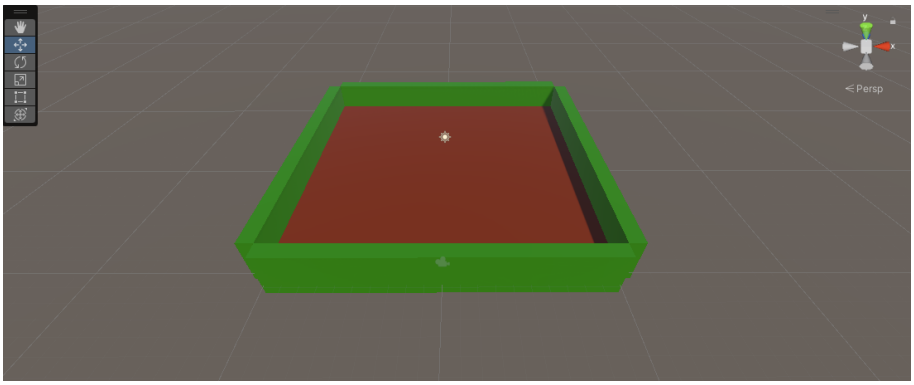


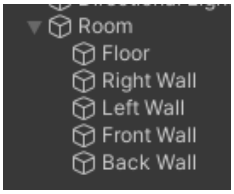
Similar vom proceda și în cazul peretelui din față.

Pentru a crea peretele din față vom duplica peretele din dreapta, iar obiectul nou îl vom redenumi $Front\ Wall$. În componenta $Transform$, vom interschimba valorile X și Z ale field-ului $Scale$. Vom face același lucru și pentru valorile X și Z alea field-ului $Position$.



Pentru peretele din spate, vom duplica peretele din față, vom denumi noul obiect *Back Wall*. Vom schimba valoarea Z a field-ului position din 10.5 în -10.5.





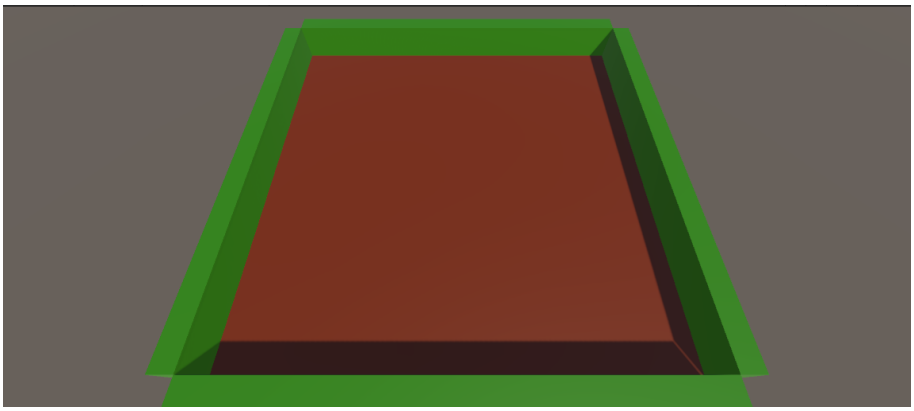
1.3 Camera

Dacă apăsăm pe butonul de *Play* vom observa ca unghiul de vizualizare al camerei principale nu este unul favorabil, ideal ar fi ca atunci când jocul rulează să putem vedea încăperea în întregime.

Putem experimenta cu diferite poziții și orientări ale camerei principale pentru a ajunge la o configurație favorabilă.

O configurație favorabilă pentru camera principală este următoarea:

- *Position* – $(0, 16, -11)$
- *Rotation* – $(60, 0, 0)$
- *Scale* – $(1, 1, 1)$



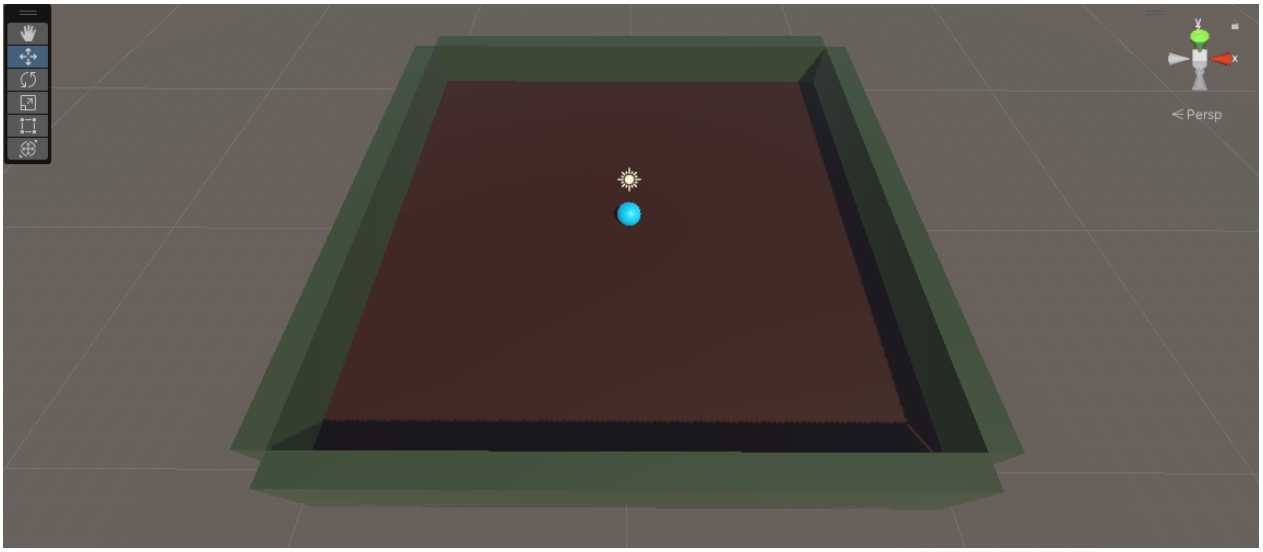
2 Controlarea unei sfere

2.1 Setarea scenei

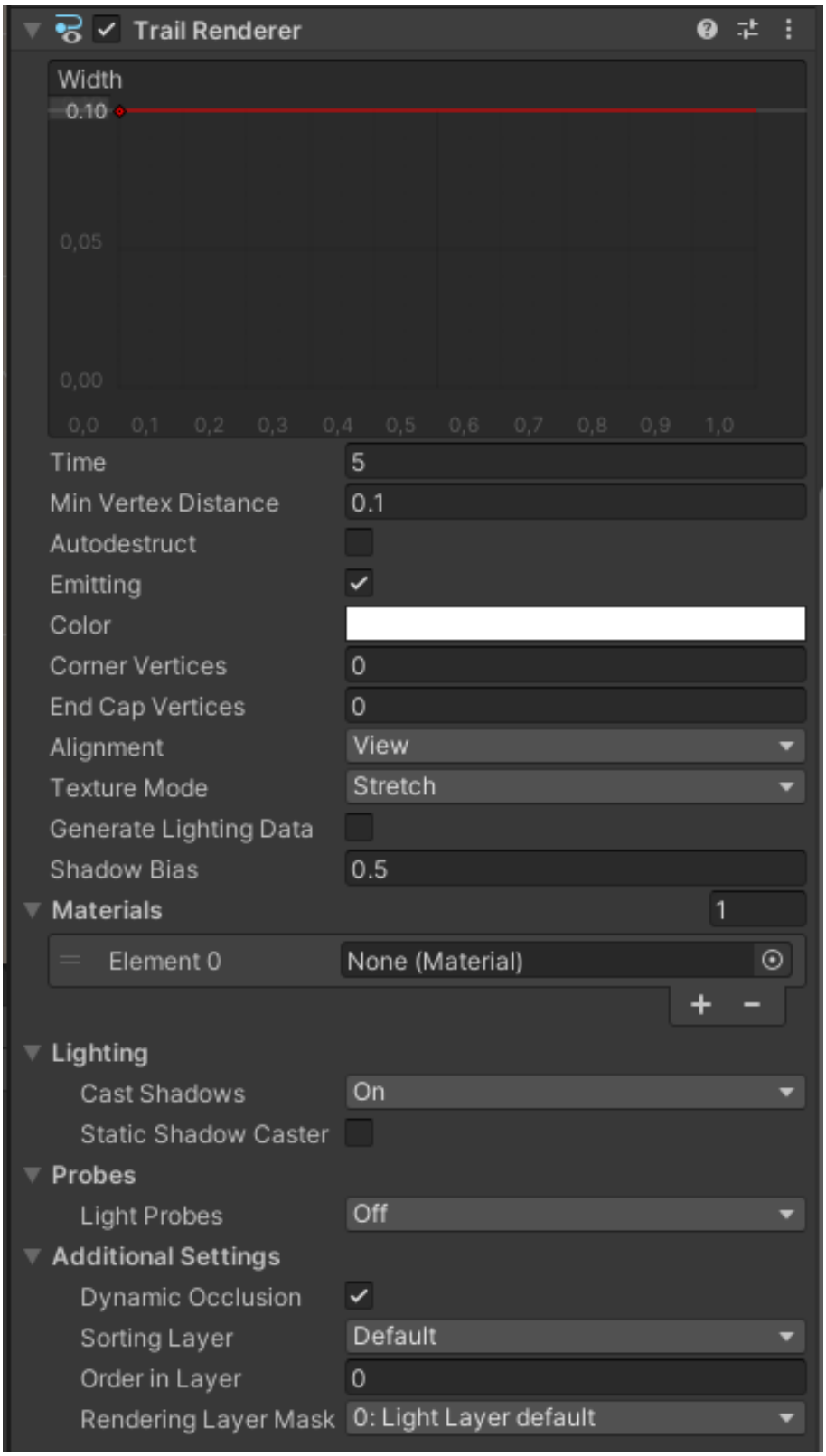
În continueare vom crea o sferă pe care jucătorul o poate controla.

Adăugăm în scenă o sferă nouă, căreia îi resetăm componenta $Transform$. Vom numi această sferă *Player*. O vom seta să fie la poziția $(0, 1, 0)$, adică în centrul încăperii create anterior ($1 = 0.5 + 0.5$, adică jumătate din grosimea podelei plus raza sferei).

Din nou, vom crea un material pentru această sferă. Îi vom atribui ce culoare dorim. Materialul îl vom numi *Player Material*.

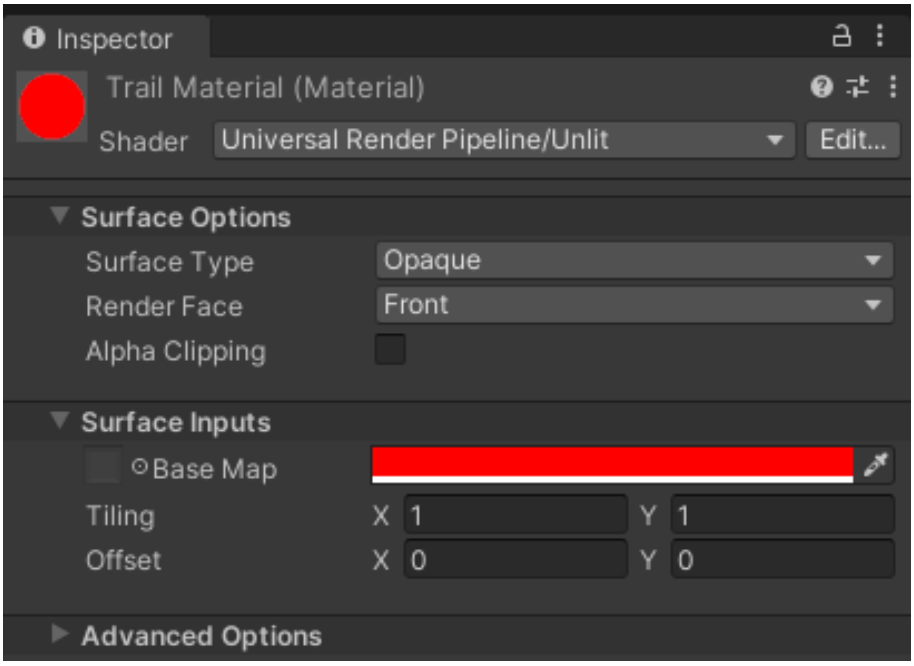


Pentru a ne fi mai ușor de urmărit modul în care bila se mișcă, îi vom adăuga o nouă componentă de tipul *Trail Renderer*. Această componentă desenează o coadă în spatele obiectlui care arată drumul pe care s-a deplasat obiectul. Acestei componente îi vom da un width egal cu 0.1 (implicit are valoarea 1.0).

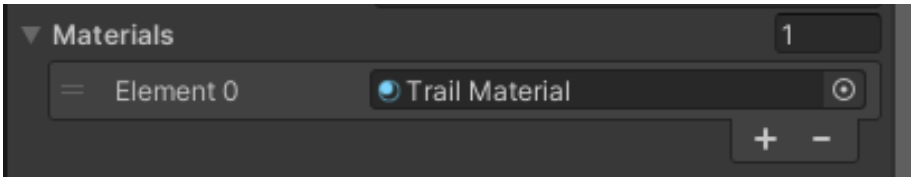


Componenta *Trail Renderer* are nevoie de un material cu care să deseneze coada. Vom crea un nou material numit *Trail Material*. Vom selecta materialul, iar din partea de sus a inspectorului vom modifica *shader*-ul folosit de material. Vom alege shader-ul *Universal Render Pipeline/ Unlit*, pentru ca această coadă să nu

fie afectată în niciun fel de iluminarea scenei. Îi putem atribui ce culoare dorim.



Acum că avem materialul, îl putem atribui componentei *Trail Renderer* (*Element* 0 al field-ului *Materials* al componentei *Trail Renderer*).



Deși nu putem controla bila încă, în scenă va apărea coada adăugată dacă mișcăm manual bila.

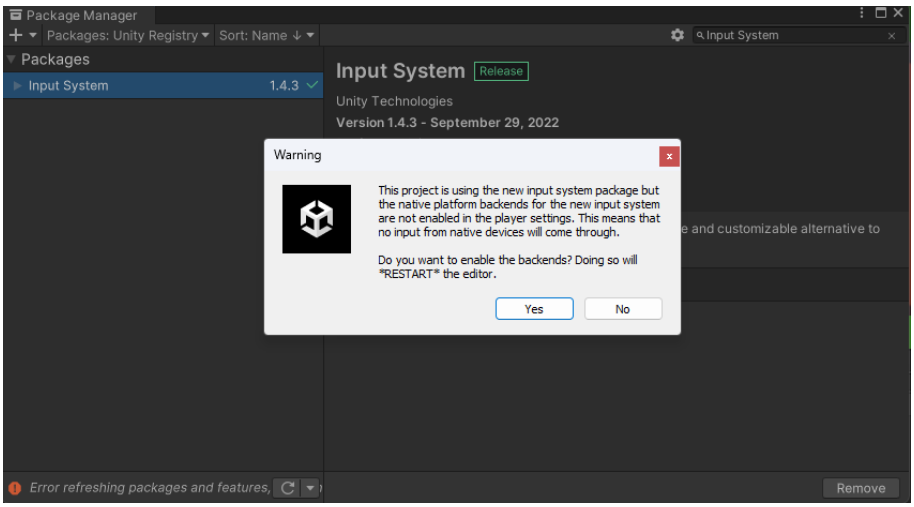
2.2 Input System

Pentru a controla această sferă este nevoie să prelucrăm *Input*-ul jucătorului într-un script. În *Unity* există mai multe moduri de a gestiona *Input*-ul jucătorului.

2.2.1 Instalare

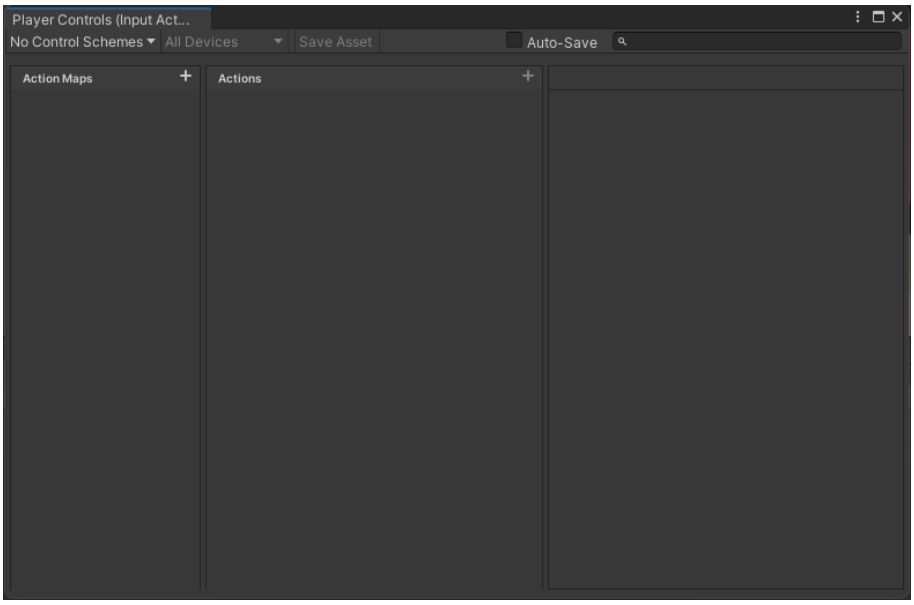
Din păcate, sistemele built-in pentru *Input* din *Unity* nu oferă suport pentru key rebinding. De asemenea, nici nu oferă prea mult suport pentru mai multe tipuri de controllere. De aceea, cei de la *Unity* au creat un nou sistem de *Input*, disponibil în *Package Manager*. În cadrul acestor laboratoare vom folosi noul sistem de *Input*.

Vom căuta *Input System* în *Package Manager*, iar apoi îl vom instala. După ce se va instala, Unity ne va avertiza ca proiectul folosește vechiul sistem de *Input* și ne întreabă dacă dorim să folosim noul sistem de *Input*. Vom da click pe *Yes* pentru a folosi noul sistem de *Input*. Acest lucru va face ca *Editorul Unity* să își dea restart.

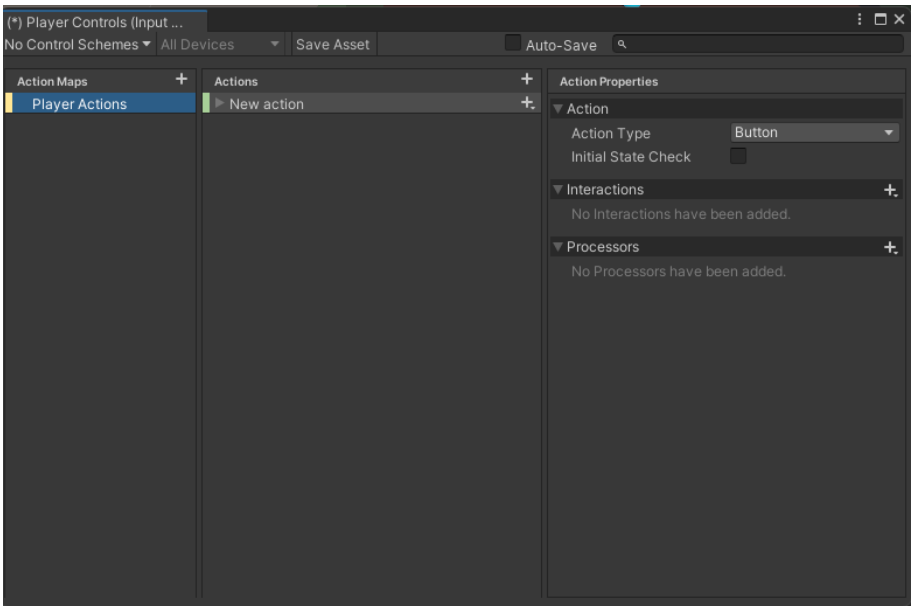


2.2.2 Definirea acțiunilor

După ce am terminat de instalat pachetul, îl putem folosi. Vom crea un nou director numit *Input Actions*. În interiorul acestui dirctor vom crea un fișier de tipul *Input Actions* (*click dreapta/ Create/ Input Actions*). Vom numi acest fișier *Player Controls*. Vom da dublu click pe fișierul creat și va apărea această fereastră:



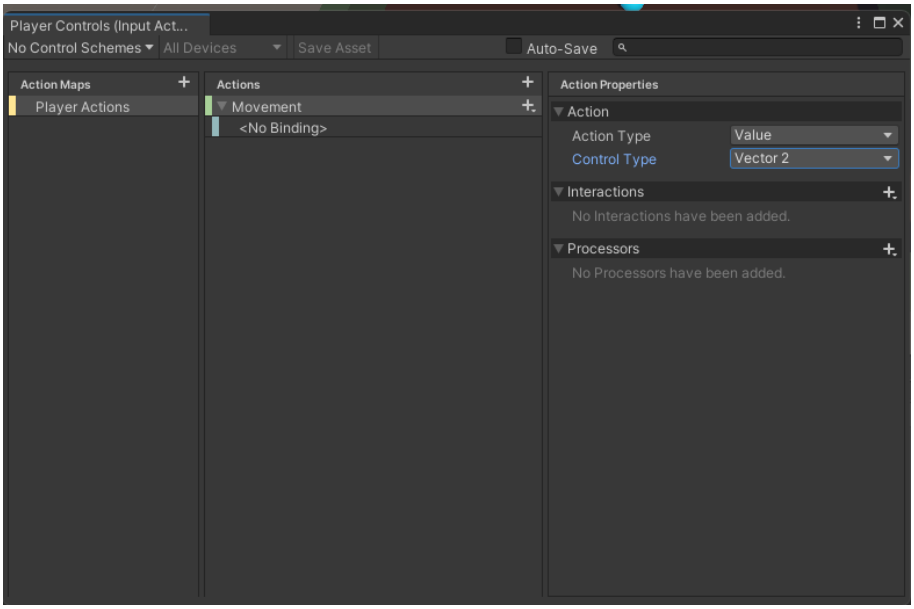
Primul lucru pe care trebuie săl facem aici este să creăm un action map. Apăsăm pe butonul + din dreptul *ActionMaps* și vom denumi acest action map *Player Actions*. Un action map poate fi vazut ca un mod de input pentru un anumit obiect. De exemplu: într-un joc open world, se poate folosi un action map când jucătorul conduce o mașină, și alt action map atunci când acesta merge pe jos.



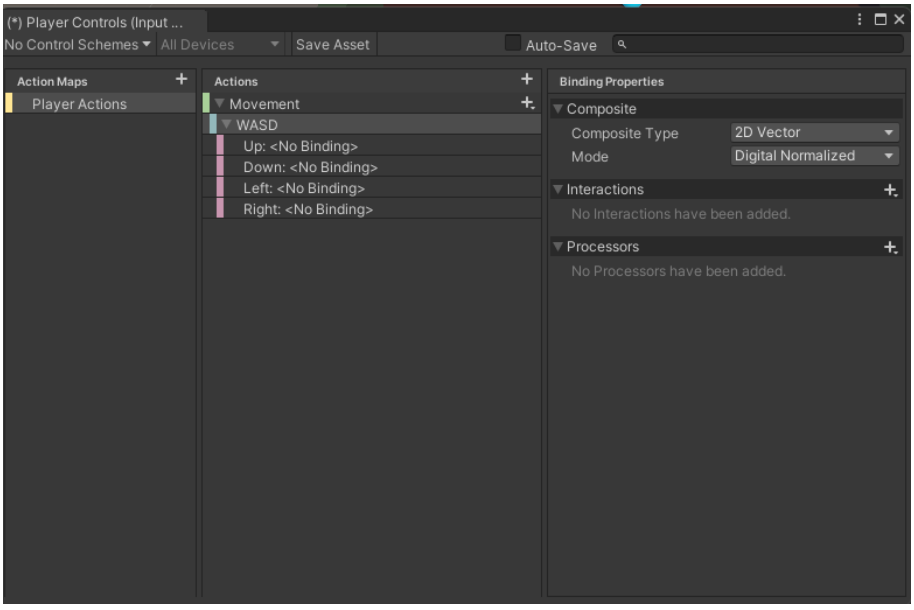
Următorul pas este să definim acțiunile specifice action map-ului. Acțiunile sunt combinații de input independente care împreună alcătuiesc un action map. De exemplu, action map-ul unui FPS are următoarele acțiuni: mișcarea caracterului, mișcarea camerei, sărit, shooting.

În cazul nostru, avem o singură acțiune, și anume mișcarea bilei. Când am creat action map-ul, s-a creat automat și o acțiune. Vom redenumi această acțiune *Movement*. În partea dreaptă a ferestrei, observăm field-ul *Action Type* care are valoarea *Button*. Vom schimba acea valoare în *Value*, deoarece vrem să primim ca input o valoare care să reprezinte direcția în care vrem ca bila să se miște. Va apărea un nou field numit *Control Type*, care are aloarea implicită *Button*. Vom schimba acea valoare în *Vector* 2, deoarece vrem ca

această acțiune să returneze un vector bidimensional care reprezintă direcția de deplasare a bilei în planul XZ.

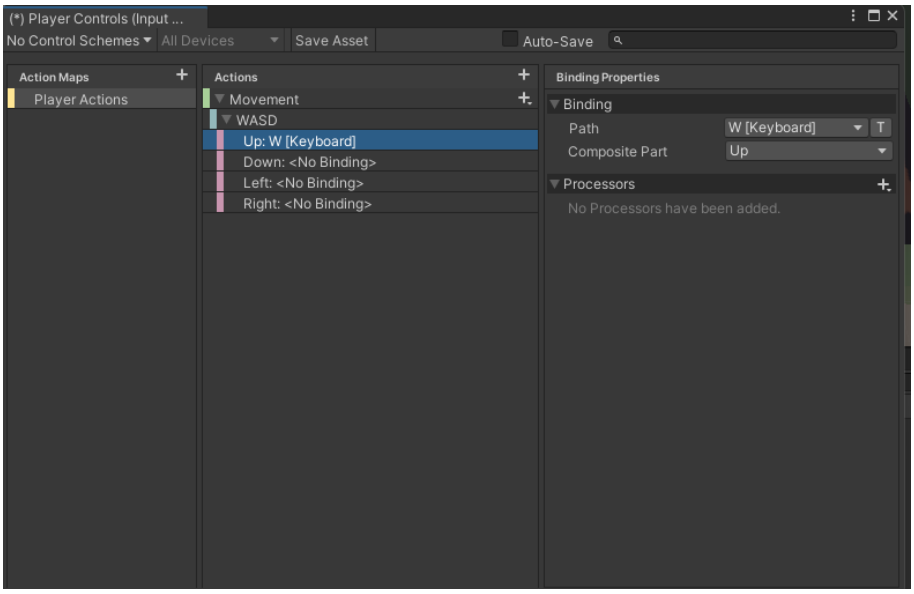


Pentru a finaliza definirea acestei acțiuni este necesar să definim și butoanele care compun această acțiune. Sub acțiune observăm că scrie $< No\ Binding >$. Acela este un binding adăugat automat de către *Unity*. Îl vom șterge, și ne vom crea propriul binding (*click dreapta/ Delete* pentru a-l șterge). Pentru a crea un nou binding apăsăm pe butonul + din dreptul acțiunii *Movement*. Acolo vom selecta *Add Up/Down/Left/Right Composite* pentru a ne crea un binding care să combine 4 taste într-o singură direcție bidirecțională (de regulă săgețile de pe tastatură sau WASD). Vom numi acest binding *WASD*.

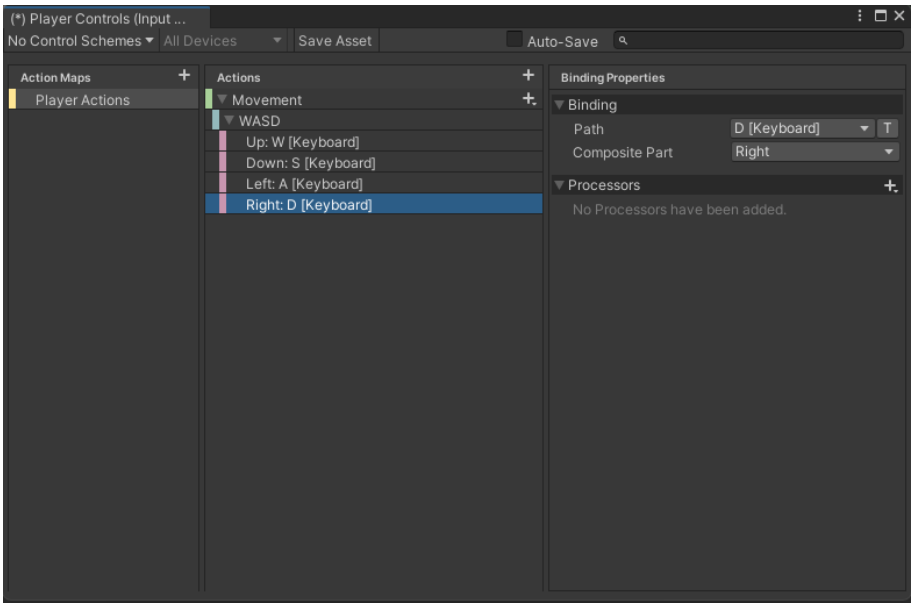


Tot ce rămâne de făcut este să atribuim fiecărei componente a acestui binding un buton de pe tastatura. Vom da click pe prima componentă a binding-ului ($Up < No\ Binding >$). În partea dreaptă vom da click

pe valoarea lui *Path*, iar de acolo putem fie să căutăm butonul dorit, fie să apăsăm pe butonul *Listen*, iar apoi să apăsăm pe butonul care vrem să reprezinte maparea. În cazul nostru, vom folosi butonul *W* pentru această mapare.



Smilar facem și cu celelalte componente ale acestui binding.

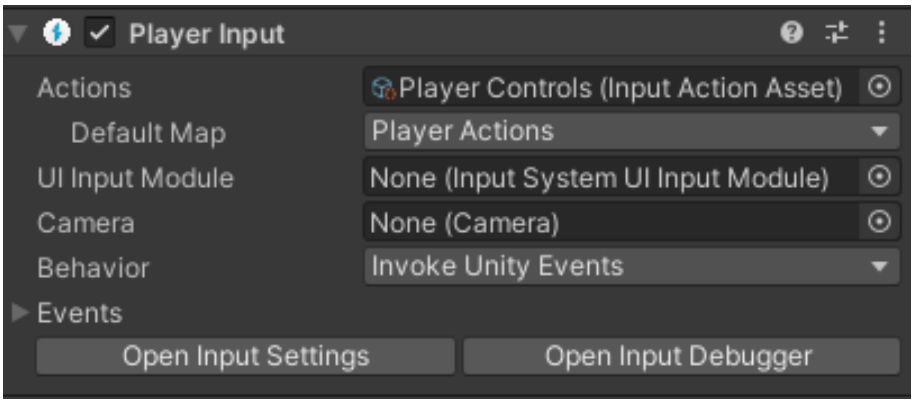


După ce terminăm de definit controalele, trebuie să apăsăm pe butonul *Save Asset* pentru a salva modificările făcute fișierului.

2.2.3 Folosirea acțiunilor de input

Revenind la obiectul nostru *Player*, îi vom adăuga o componentă de tipul *Player Input* pentru a face legătura între fișierul creat mai devreme și codul pe care îl vom scrie ulterior. Această componentă are un

field numit *Actions*. Vom da drag and drop fișierului creat anterior către acel field. De asemenea, vom schimba valoarea field-ului *Behaviour* din *Send Messages* în *Invoke Unity Events*. Acest *Behaviour* ne va permite să specificăm explicit funcția care vrem să se apeleze atunci când *Input*-ul este primit.



Acum suntem pregătiți să folosim input-ul în propriile scripturi. Vom crea un nou director numit *Scripts*, iar în interiorul lui vom crea un nou script numit *MovingSphere*. Vom șterge codul existent și vom rescrie codul necesar pentru o componentă fără funcționalități.

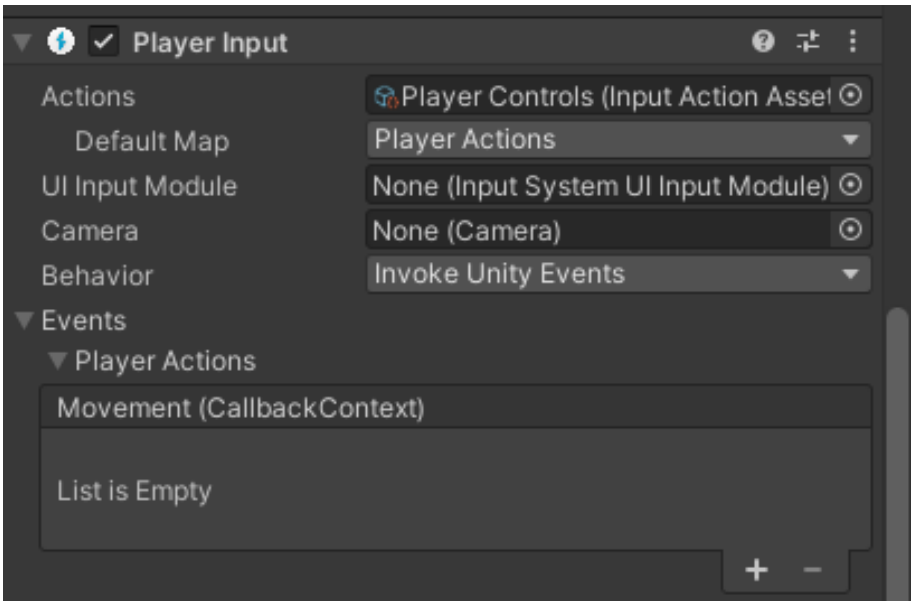
```
using UnityEngine;

public class MovingSphere : MonoBehaviour
{
}
```

Vom defini funcția care va fi apelată de către sistemul de input atunci când butoanele pentru mișcare sunt apăsate. Vom defini o nouă funcție publică numită *Movement* care primește ca argument un obiect de tipul *InputAction.CallbackContext* (se află în namespace-ul *UnityEngine.InputSystem*), care conține valoarea primită de la sistemul de *Input*.

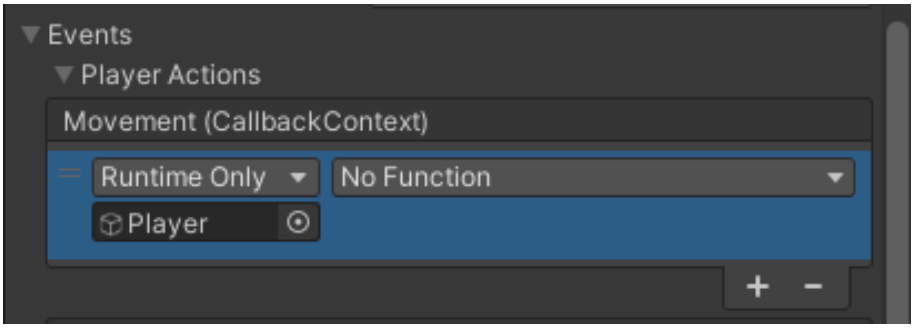
```
using UnityEngine;
using UnityEngine.InputSystem;

public class MovingSphere : MonoBehaviour
{
    public void Movement(InputAction.CallbackContext context)
    {
    }
}
```

Acum, vom atașa această componentă asupra obiectului *Player*.

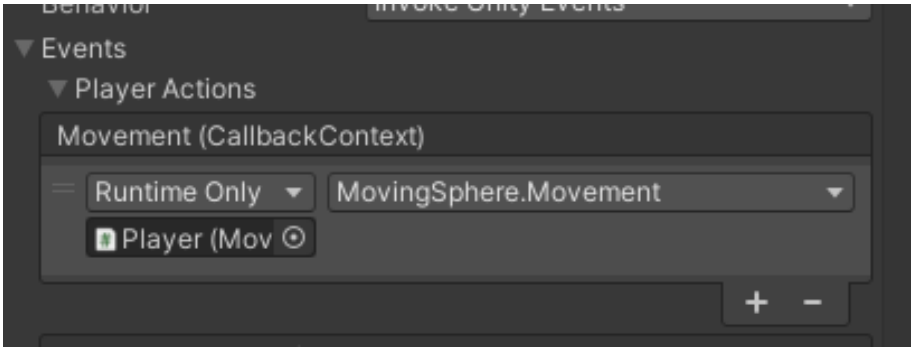
După ce scriptul anterior este atașat, este nevoie să îi indicăm componentei *Player Input* faptul că vrem să apeleze metoda *Movement* din componenta *MovingSphere* atunci când acțiunea *Movement* este folosită. Din componenta *Player Input*, extindem proprietatea *Events*, iar de acolo extindem proprietatea *Player Actions*. Acolo vom observa un field gol care face legătura între acțiunea Movement și o funcție.



Apăsăm pe butonul + pentru a selecta funcția definită anterior. Apare un field cu valoarea *None* (*Object*). Acolo trebuie să specificăm obiectul pe care se află componenta a cărei metode vrem s-o apelăm. Vom da drag and drop obiectului *Player* către acel field.



Acum trebuie specificată funcția folosind chenarul din dreapta. Dăm click pe *No Function*, alegem componenta în care se află metoda, în cazul nostru *MovingSphere*, iar de acolo selectăm metoda dorită, în cazul nostru *Movement*.



Pentru a ne asigura că merge, în interiorul funcției vom folosi *Debug.Log* pentru a ne asigura că valorile primite sunt corecte, iar metoda este apelată.

```
using UnityEngine;
using UnityEngine.InputSystem;

public class MovingSphere : MonoBehaviour
{
    public void Movement(InputAction.CallbackContext context)
    {
        Debug.Log("Movement: " + context.ReadValue<Vector2>());
    }
}
```

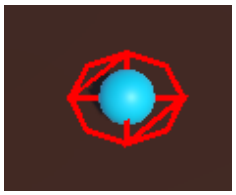
2.3 Mișcarea bilei în funcție de input

2.3.1 Poziție

Prin funcția *Movement* primim un $Vector2$ pe care il putem folosi pentru a mișca bila de-alungul planului XZ. Pentru a vizualiza acest vector primit, putem să folosim valorile sale pentru componentele X, respectiv Z ale obiectului.

```
using UnityEngine;
using UnityEngine.InputSystem;

public class MovingSphere : MonoBehaviour
{
    private Vector2 _movement = Vector2.zero;

    public void Movement(InputAction.CallbackContext context) =>
        _movement = context.ReadValue<Vector2>();

    private void Update()
    {
        transform.localPosition = new Vector3(_movement.x,
            transform.localPosition.y, _movement.y);
    }
}
```

Side note: *transform.localPosition* reprezinta valoarea poziției obiectului în coordonate locale, adică cele afișate în inspector. Coordonatele din lume se află folosind *transform.position*. În cazul nostru, în care obiectul nu are niciun părinte nu este nicio diferență între a folosi *transform.localPosition* și *transform.position*.

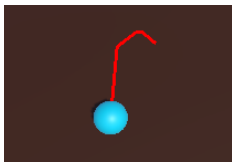


2.3.2 Viteză

Momentan, noi am mutat bila la poziția dată de vectorul de *Input*. Această mișcare nu este ce ne dorim în jocul pe care îl facem. Ce ne-am dori ar fi ca bila să aibă o viteză de deplasare, iar acea viteză de deplasare să fie infuențată de vectorul de input.

Am putea face similar cu laboratorul anterior, și să folosim input-ul pe post de viteză de deplasare (în cazul aceste, viteza este o valoare vectorială, deoarece reprezintă atât viteza pe aza X cât și cea pe aza Z, în engleză aceasta unitate vectorială se numește *velocity*).

```
using UnityEngine;
using UnityEngine.InputSystem;

public class MovingSphere : MonoBehaviour
{
    private Vector2 _movement = Vector2.zero;

    public void Movement(InputAction.CallbackContext context) =>
        _movement = context.ReadValue<Vector2>();

    private void Update()
    {
        var velocity = new Vector3(_movement.x, 0.0f, _movement.y);
        transform.localPosition += velocity * Time.deltaTime;
    }
}
```

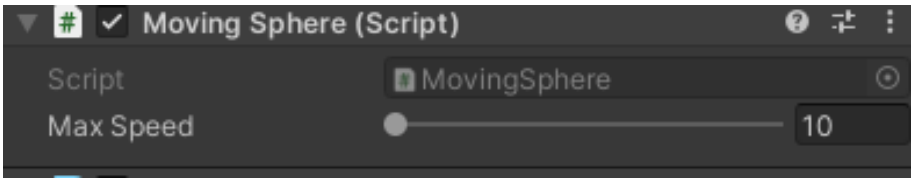


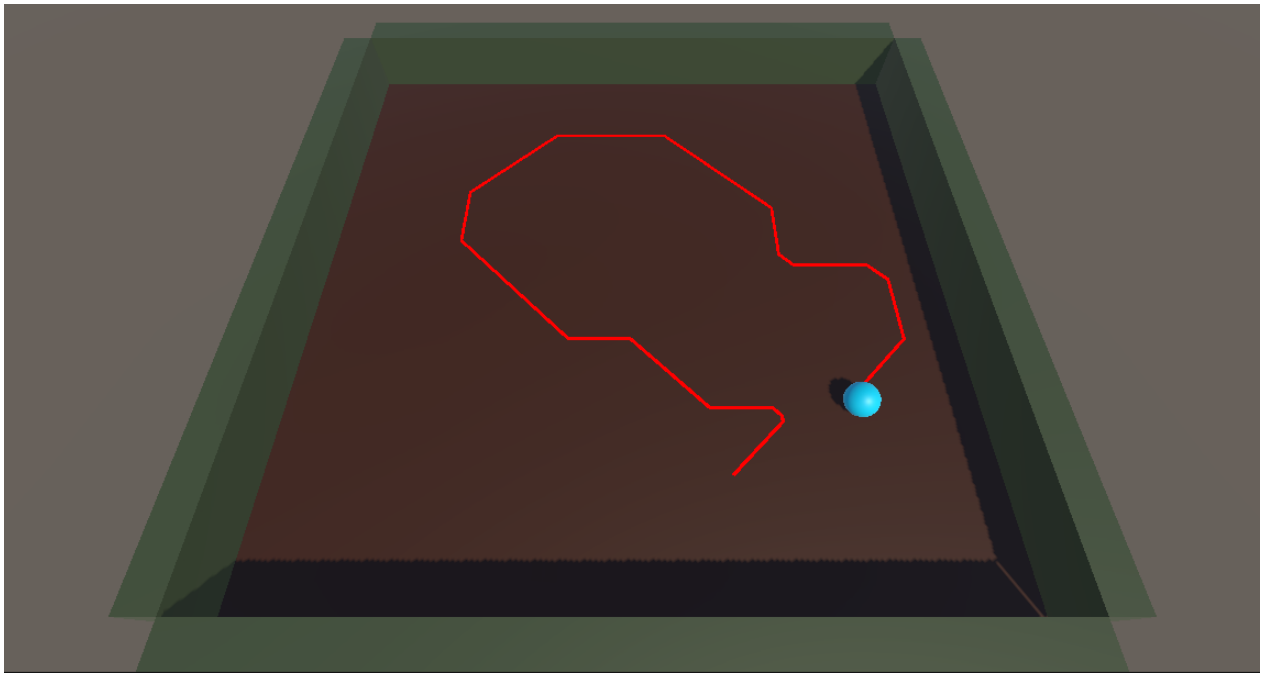
Acum mișcarea este puțin mai naturală, dar mult prea lentă. Pentru ca bila să se miște mai repede putem introduce o nouă variabilă serializabilă *_maxSpeed* care să fie înmulțită cu variabila *velocity*.

```
using UnityEngine;
using UnityEngine.InputSystem;

public class MovingSphere : MonoBehaviour
{
    [SerializeField, Range(10.0f, 100.0f)]
    private float   _maxSpeed = 10.0f;
```

```
    private Vector2 _movement = Vector2.zero;

    public void Movement(InputAction.CallbackContext context) =>
        _movement = context.ReadValue<Vector2>();

    private void Update()
    {
        var velocity = new Vector3(_movement.x, 0.0f, _movement.y) * _maxSpeed;
        transform.localPosition += velocity * Time.deltaTime;
    }
}
```

Se observă specificatorul *Range*$(10.0f, 100.0f)$. Acesta face ca din inspector să se poată atribui doar valori cuprinse între 10 și 100 acestei variabile. În inspector, se poate observa un slider cu valori între 10 și 100 pentru această variabilă.





Pentru a menține codul organizat vom introduce și o variabilă numită *displacement* care reprezintă mișcarea pentru un singur cadru.

```
var displacement = velocity * Time.deltaTime;
transform.localPosition += displacement;
```

2.3.3 Accelerație

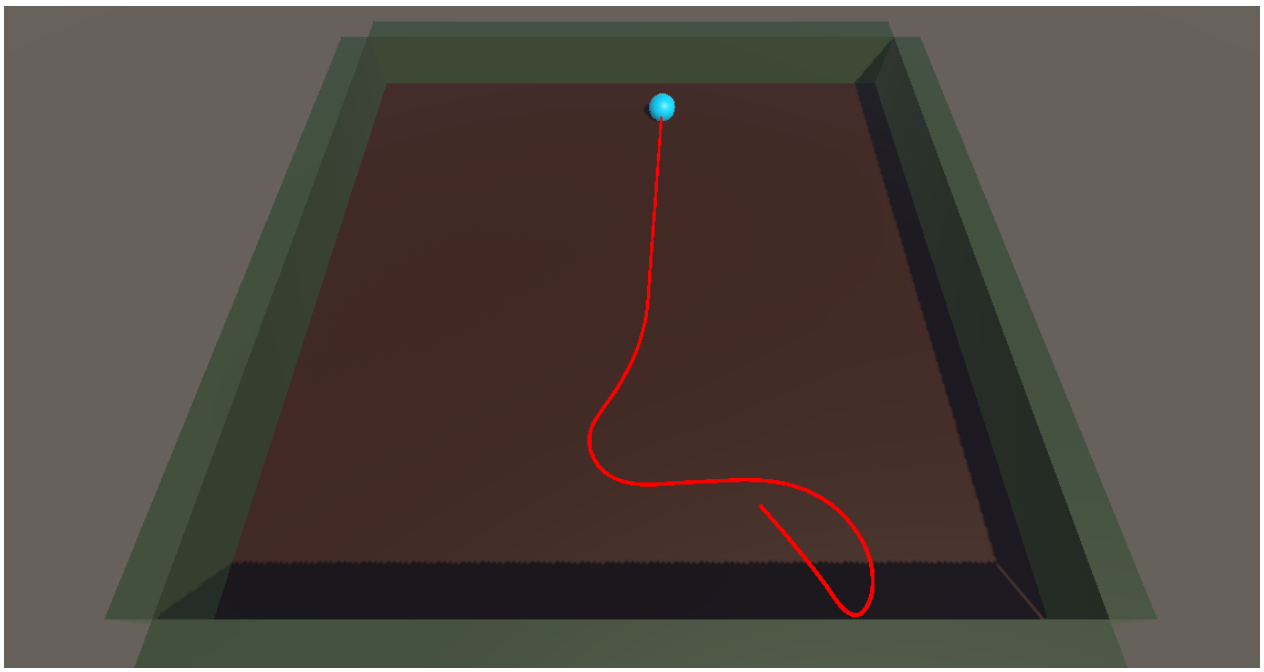
Mișcarea încă nu este naturală. bila răspunde instant la input. În realitate nu se întâmplă așa, acolo viteza crește și scade treptat. Vom introduce un nou concept, și anume accelerație. Accelerația o putem privi ca pe

o viteză a vitezei.

Dacă privim poziția unui obiect ca o funcție matematică, viteza este derivata de ordinul întâi al funcției, iar accelerația este derivata de ordinul 2 al funcției.

Ca mod de implementare, putem să o implementăm exact cum am implementat și viteza anterior, doar că acum o implementăm asupra variabilei *velocity*, ci nu *transform.localPosition*. Nu mai putem folosi o variabilă locală *velocity* deoarece avem nevoie de valoarea acesteia între frame-uri.

```
using UnityEngine;
using UnityEngine.InputSystem;

public class MovingSphere : MonoBehaviour
{
    [SerializeField, Range(10.0f, 100.0f)]
    private float    _maxSpeed = 10.0f;
    private Vector2 _movement = Vector2.zero;
    private Vector3 _velocity = Vector3.zero;

    public void Movement(InputAction.CallbackContext context) =>
        _movement = context.ReadValue<Vector2>();

    private void Update()
    {
        var acceleration = new Vector3(_movement.x, 0.0f, _movement.y) *
            _maxSpeed;
        _velocity += acceleration * Time.deltaTime;
        var displacement = _velocity * Time.deltaTime;
        transform.localPosition += displacement;
    }
}
```



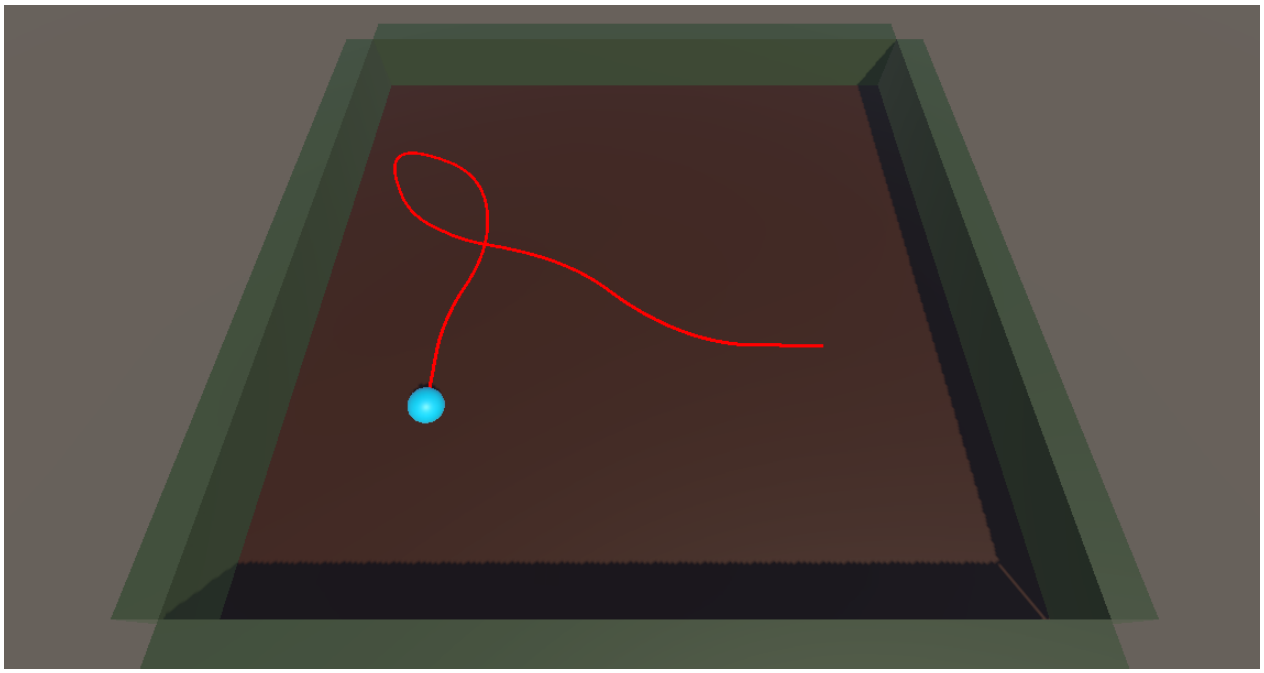
Acum mișcarea este mai fină și mai naturală, dar bila a devenit mult mai greu de controlat.

2.3.4 Viteza dorită

Putem combina metodele discutate anterior (modificarea vitezei direct accelerație). Astfel, vom avea o variabilă *desiredVelocity* care reprezintă valoarea pe care ne-o dorim să o avem ca viteză și care este calculată în funcție de *Input*. Asupra variabilei *velocity* aplicăm forțe de accelerație pentru a ajunge la valoarea *desiredVelocity*. În cazul acesta, bila va fi mult mai ușor de controlat deoarece aceasta va decelera automat când nu este niciun input, iar atunci când este input, aceasta va avea o limită de viteză (nu poate depăși *desiredVelocity*, variabilă care este calculată la fiecare frame).

```
using UnityEngine;
using UnityEngine.InputSystem;

public class MovingSphere : MonoBehaviour
{
    [SerializeField, Range(10.0f, 100.0f)]
    private float _maxSpeed = 10.0f;
    [SerializeField, Range(10.0f, 100.0f)]
    private float _maxAcceleration = 10.0f;

    private Vector2 _movement = Vector2.zero;
    private Vector3 _velocity = Vector3.zero;

    public void Movement(InputAction.CallbackContext context) =>
        _movement = context.ReadValue<Vector2>();

    private void Update()
    {
        var desiredVelocity = new Vector3(_movement.x, 0.0f, _movement.y) *
                              _maxSpeed;
        var maxSpeedChange = _maxAcceleration * Time.deltaTime;

        _velocity.x = Mathf.MoveTowards(_velocity.x, desiredVelocity.x,
            maxSpeedChange);
        _velocity.z = Mathf.MoveTowards(_velocity.z, desiredVelocity.z,
            maxSpeedChange);

        var displacement = _velocity * Time.deltaTime;
        transform.localPosition += displacement;
    }
}
```

Am introdus variabila *_maxAcceleration* care reprezintă viteza cu care variabila *_velocity* se îndreaptă spre variabila *desiredVelocity*. Am folosit funcția *Mathf.MoveTowards*. Această funcție primește ca argumente 3 numere (a, b și c) și returnează un număr d. Semnificația ei este următoarea: de la valoarea a mergi către valoarea b cu un pas de lungime maxim c și returnează valoarea la care ai ajuns. În cazul nostru, noi o folosim pentru a ajunge de la valoarea lui *_velocity* către valoarea lui *desiredVelocity*, cu pași de lungime egală cu *maxSpeedChange*.



2.4 Constrângerea poziției

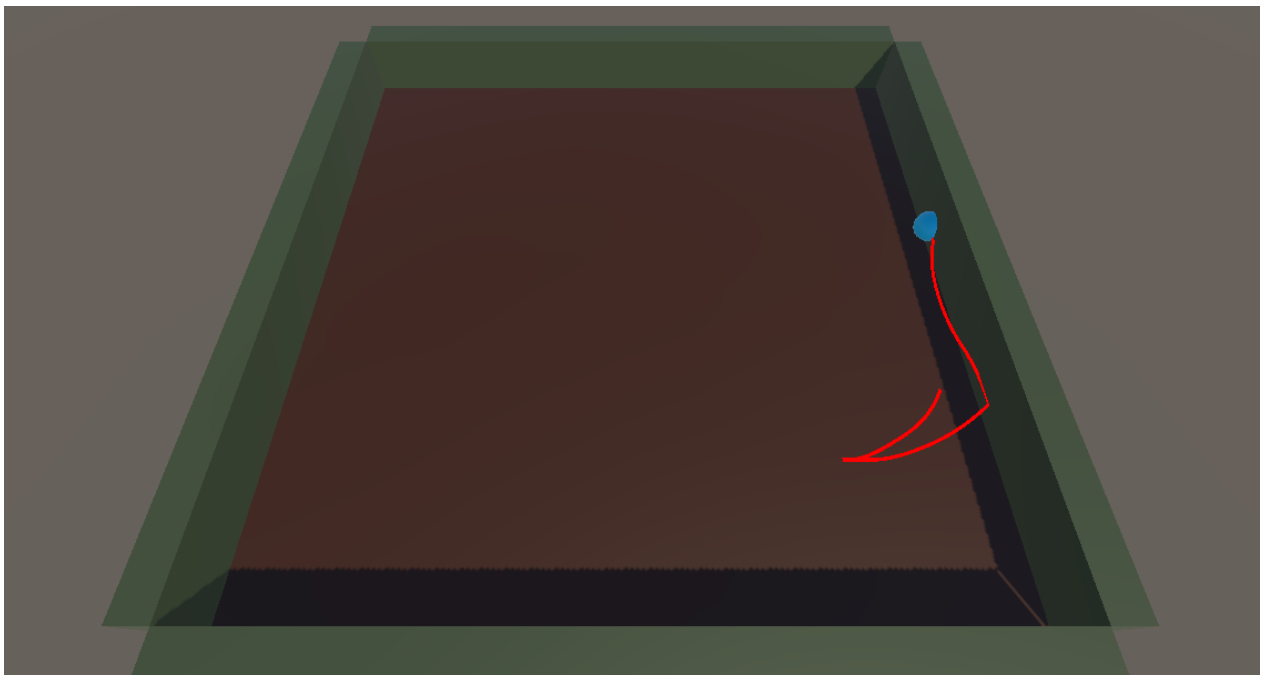
Momentan, sfera noastră poate trece prin pereți deoarece nu am implementat niciun fel de logică pentru detectarea coliziunilor.

O simplă implementare ar fi să detectăm dacă urmează ca bila să ajungă la o poziție care este în afara încăperii, iar in cazul acela să rămână la poziția anterioară. Vom introduce o variabilă de tip *Rect* care reprezintă zona în care bila se poate mișca, și încă o variabilă de tip $Vector3$ care reprezintă următoarea poziție.

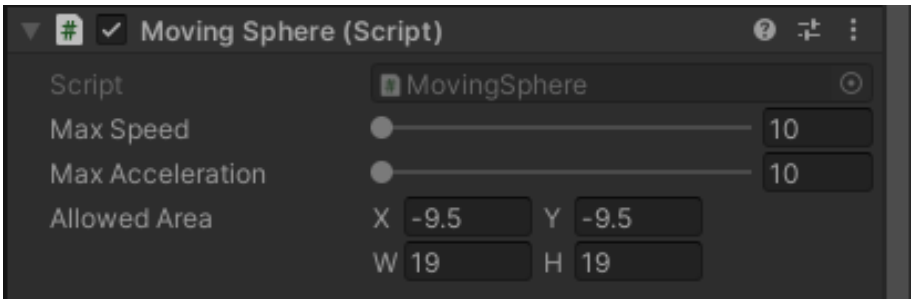
```
using UnityEngine;
using UnityEngine.InputSystem;

public class MovingSphere : MonoBehaviour
{
    [SerializeField, Range(10.0f, 100.0f)]
    private float _maxSpeed = 10.0f;
    [SerializeField, Range(10.0f, 100.0f)]
    private float _maxAcceleration = 10.0f;
    [SerializeField]
    private Rect _allowedArea = new(-10.0f, -10.0f, 20.0f, 20.0f);

    private Vector2 _movement = Vector2.zero;
    private Vector3 _velocity = Vector3.zero;

    public void Movement(InputAction.CallbackContext context) =>
        _movement = context.ReadValue<Vector2>();

    private void Update()
    {
        var desiredVelocity = new Vector3(_movement.x, 0.0f, _movement.y) *
```

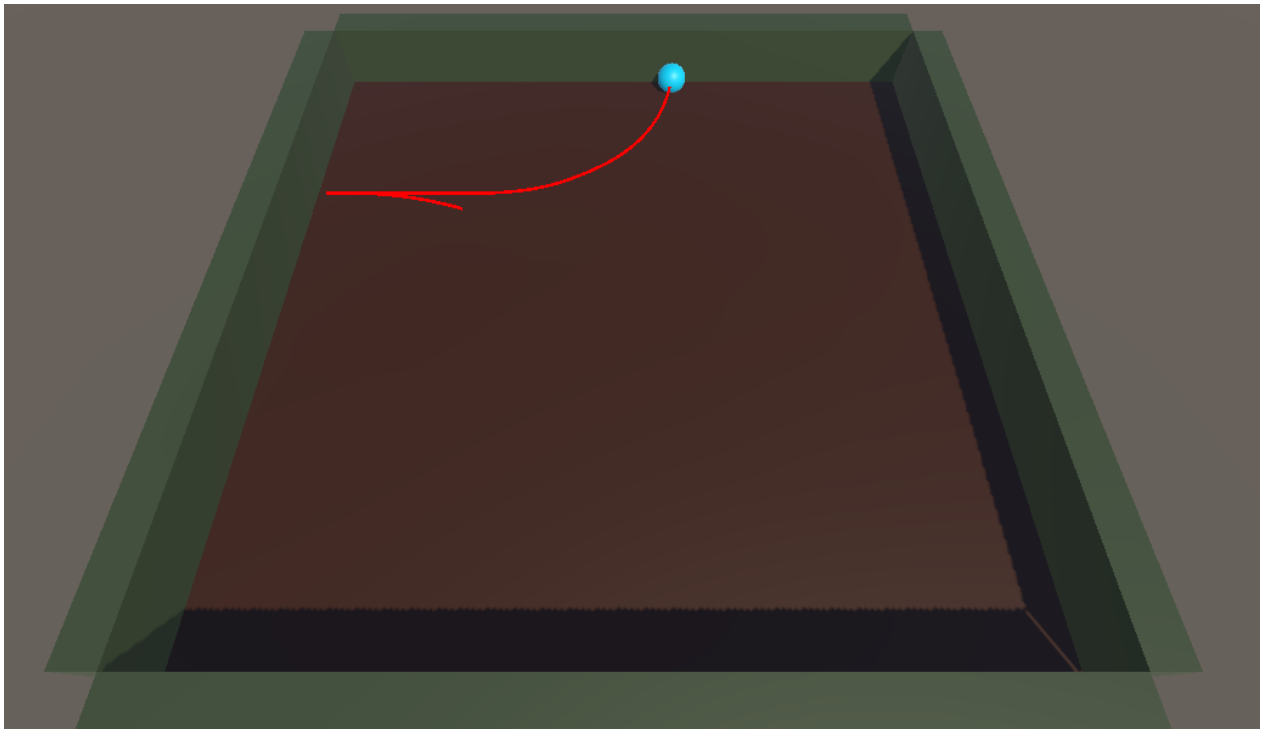
```
                                _maxSpeed;
        var maxSpeedChange = _maxAcceleration * Time.deltaTime;

        _velocity.x = Mathf.MoveTowards(_velocity.x, desiredVelocity.x,
            maxSpeedChange);
        _velocity.z = Mathf.MoveTowards(_velocity.z, desiredVelocity.z,
            maxSpeedChange);

        var displacement = _velocity * Time.deltaTime;
        var newPosition = transform.localPosition + displacement;

        if (!_allowedArea.Contains(new Vector2(newPosition.x, newPosition.z)))
            newPosition = transform.localPosition;

        transform.localPosition = newPosition;
    }
}
```

Side note: am folosit $newVector2(newPosition.x, newPosition.z)$ pentru metoda *Contains* și nu am folosit *newPosition* direct deoarece *newPosition* este un vector 3D, iar componenta sa Y nu se află în planul în care facem verificarea.



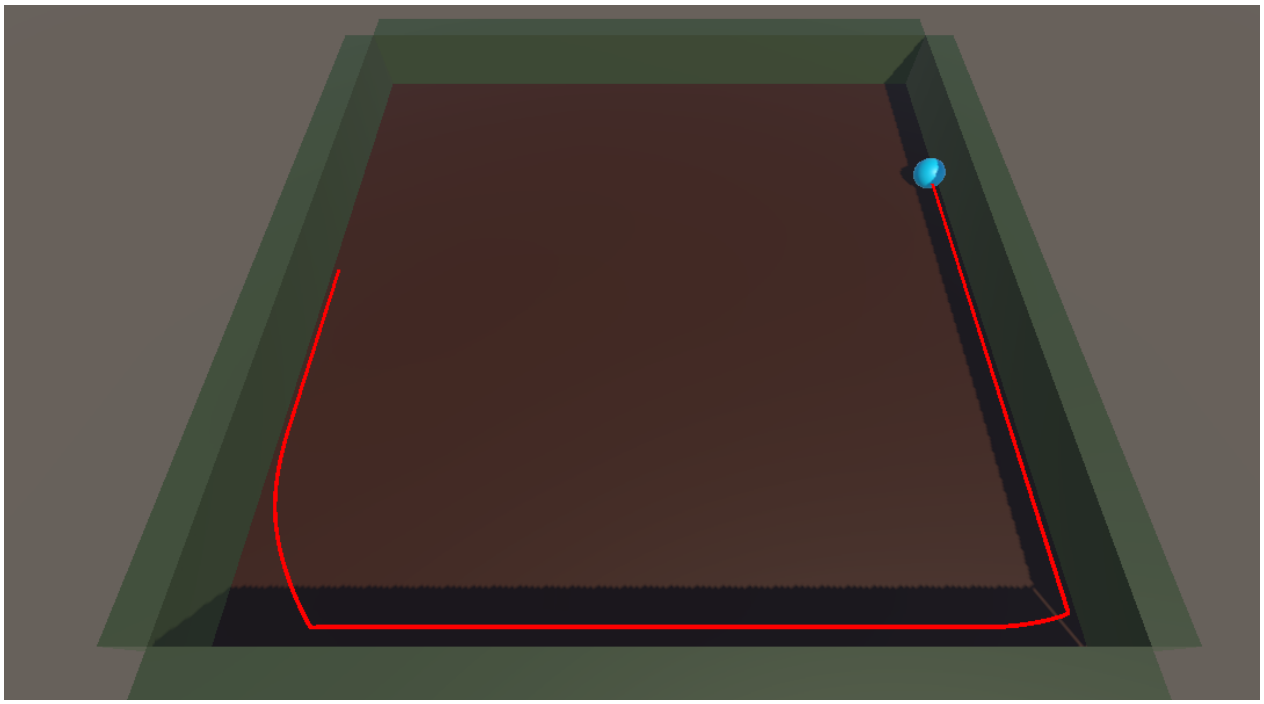
În primul rând se observă că bila încă trece prin pereți. Asta este deoarece noi nu luăm în calcul raza sferei (care este egală cu 0.5). Soluția simplă pentru această problemă este ca din inspector să adunăm 0.5 la coordonatele X și Y ale field-ului *Allowed Area*, iar din lungimea și înălțimea acestui filed să scădem 1.





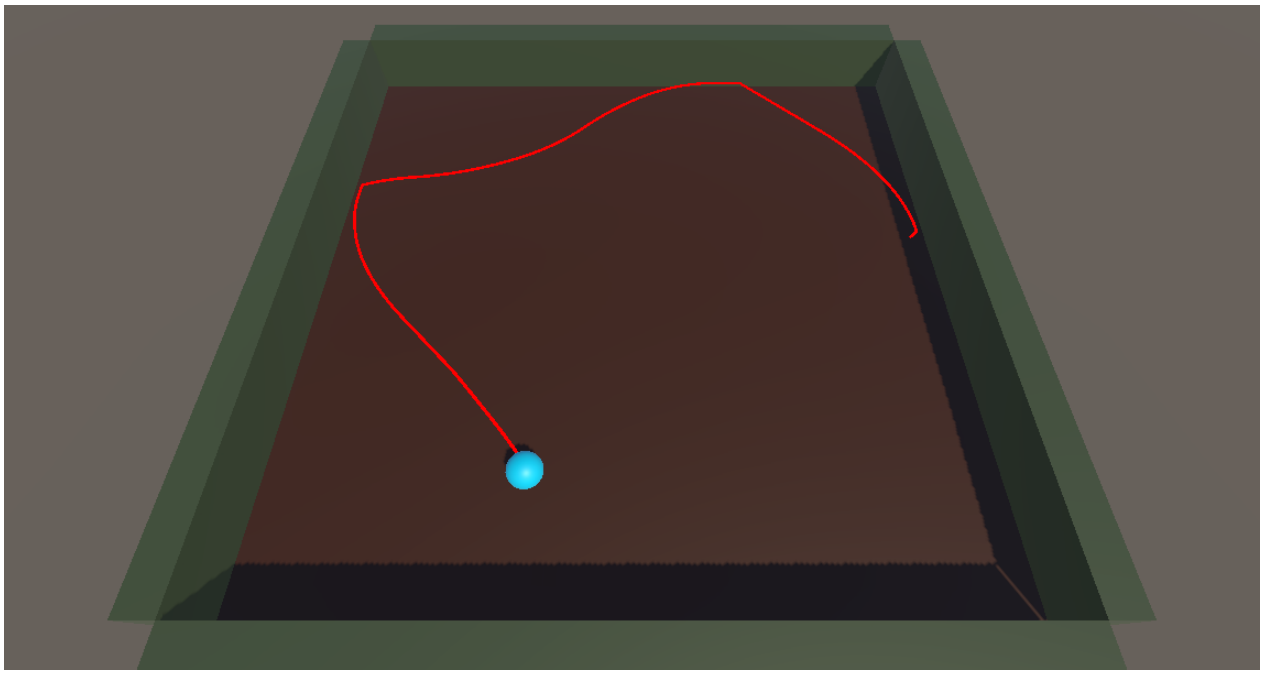
Mișcarea pe margini încă nu este perfectă deoarece păstrăm poziția veche a sferei. O soluție mai bună este să setăm manual poziția acesteia, în funcție de marginea pe care se află. Concret, dacă bila urmează să intre într-un perete, atunci îi vom modifica poziția astfel încât aceasta să fie fix la marginea sa admisă. Putem folosi funcția *Mathf.Clamp* pentru asta. Ea primește ca argument un număr, pe care îl returnează dacă se află în intervalul dat de celelalte două argumente. Dacă numărul depășește capetele intervalului, atunci cel mai apropiat capăt este returnat. În cazul acesta este utilă deoarece podeaua poate fi privită ca două intervale (unul pe axa X și altul pe axa Y, iar bila trebuie să rămână mereu în interiorul intervalelor, adică în interorul încăperii).

```
if (!_allowedArea.Contains(new Vector2(newPosition.x, newPosition.z)))
{
    newPosition.x = Mathf.Clamp(newPosition.x, _allowedArea.xMin,
        _allowedArea.xMax);
    newPosition.z = Mathf.Clamp(newPosition.z, _allowedArea.yMin,
        _allowedArea.yMax);
}
```



Acum bila se mișcă bine în dreptul marginilor, dar încă mai este o problemă. Bila pare că se lipește de margini pentru o perioadă. Asta este deoarece la coliziune, viteza sferei rămâne aceeași, pentru că noi nu o modificăm. O soluție ar fi ca atunci când sfera se lovește de un perete, viteza pe axa respectivă să devină 0. Pentru aceasta nu mai putem folosi if-ul anterior, pentru ca avem nevoie să știm exact și peretele de care s-a lovit bila. Vom scrie o serie de if-uri pentru asta:

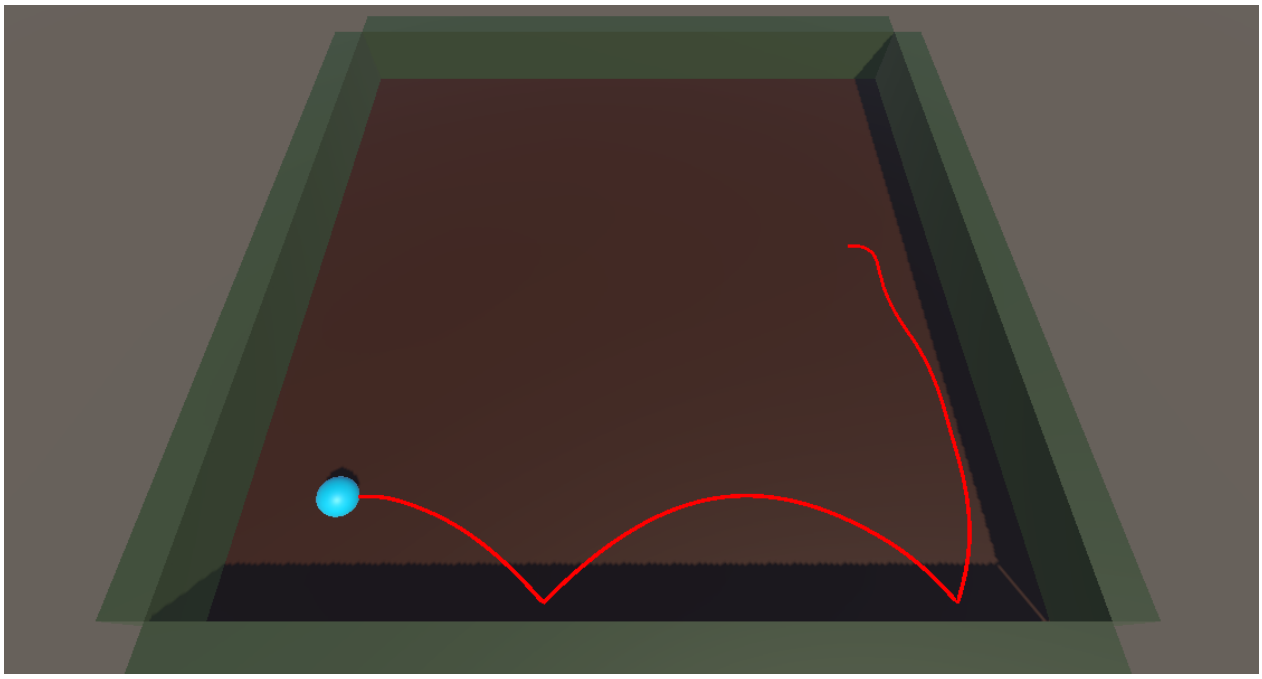
```
if (newPosition.x < _allowedArea.xMin)
{
    newPosition.x = _allowedArea.xMin;
    _velocity.x = 0.0f;
}

if (newPosition.x > _allowedArea.xMax)
{
    newPosition.x = _allowedArea.xMax;
    _velocity.x = 0.0f;
}

if (newPosition.z < _allowedArea.yMin)
{
    newPosition.z = _allowedArea.yMin;
    _velocity.z = 0.0f;
}

if (newPosition.z > _allowedArea.yMax)
{
    newPosition.z = _allowedArea.yMax;
    _velocity.z = 0.0f;
}
```



Acum bila nu se mai lipește de margini și mișcarea este destul de fină.

Mai putem face un lucru. Puem face ca atunci când sfera lovește un perete, aceasta să se întoarcă în direcția în care a venit, făcând-o să pară că se mișcă mai natural. Pentru asta, trebuie doar să inversăm viteza pe direcția peretelui de care ne-am lovit.

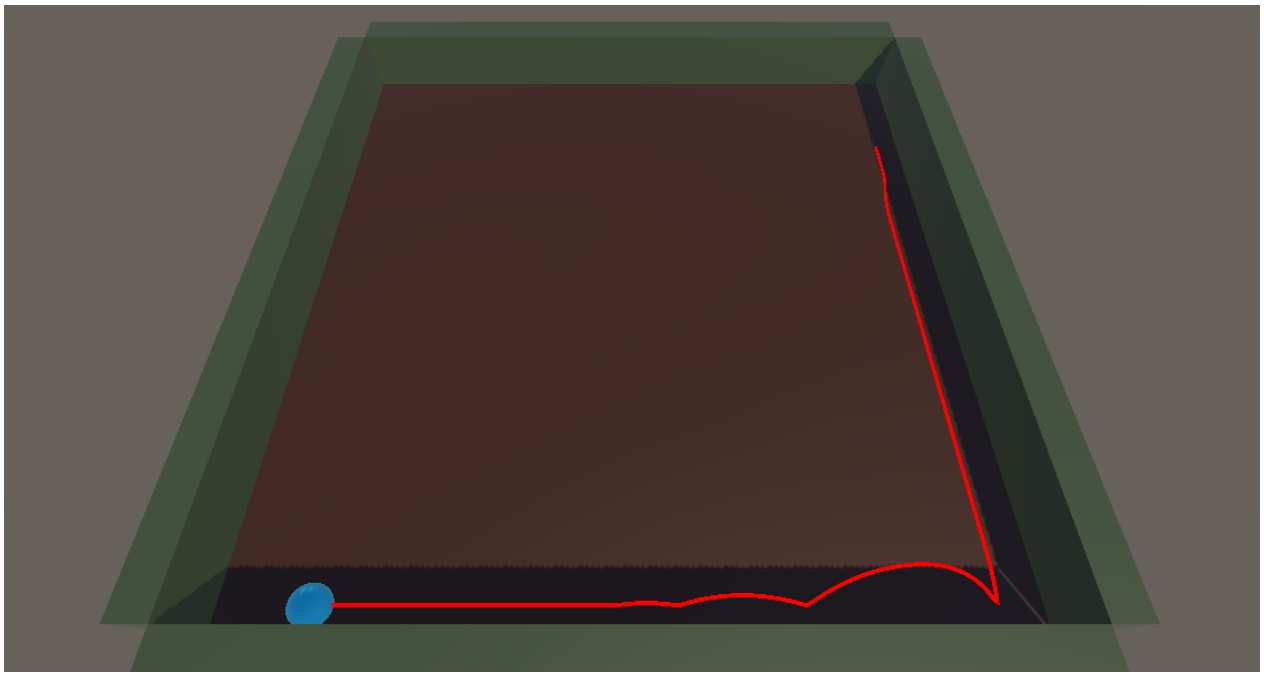
```
if (newPosition.x < _allowedArea.xMin)
{
    newPosition.x = _allowedArea.xMin;
    _velocity.x = -_velocity.x;
}

if (newPosition.x > _allowedArea.xMax)
{
    newPosition.x = _allowedArea.xMax;
    _velocity.x = -_velocity.x;
}

if (newPosition.z < _allowedArea.yMin)
{
    newPosition.z = _allowedArea.yMin;
    _velocity.z = -_velocity.z;
}

if (newPosition.z > _allowedArea.yMax)
{
    newPosition.z = _allowedArea.yMax;
    _velocity.z = -_velocity.z;
}
```



În cazul implementat aici, simulăm o situație ideală în care nu se pierde energie la coliziunea cu pereții. În viața reală s-ar pierde energie la aceste coliziuni, iar viteza de întoarcere ar fi puțin mai mică. Pentru a implementa asta, putem introduce o nouă variabilă numită *_bounciness* care reprezintă procentul de energie care se păstrează la coliziunea cu peretele. Această variabilă este un număr între 0 și 1, unde 0 înseamnă că niciun pic de energie nu se păastrează (*_velocity* devine 0 pe axa respectivă), iar 1 înseamnă că toată energia este păstrată (cazul de mai sus).

```
using UnityEngine;
using UnityEngine.InputSystem;

public class MovingSphere : MonoBehaviour
{
    [SerializeField, Range(10.0f, 100.0f)]
    private float _maxSpeed = 10.0f;
    [SerializeField, Range(10.0f, 100.0f)]
    private float _maxAcceleration = 10.0f;
    [SerializeField]
    private Rect _allowedArea = new(-10.0f, -10.0f, 20.0f, 20.0f);
    [SerializeField, Range(0.0f, 1.0f)]
    private float _bounciness = 0.5f;

    private Vector2 _movement = Vector2.zero;
    private Vector3 _velocity = Vector3.zero;

    public void Movement(InputAction.CallbackContext context) =>
        _movement = context.ReadValue<Vector2>();

    private void Update()
    {
        var desiredVelocity = new Vector3(_movement.x, 0.0f, _movement.y) *
                              _maxSpeed;
        var maxSpeedChange = _maxAcceleration * Time.deltaTime;

        _velocity.x = Mathf.MoveTowards(_velocity.x, desiredVelocity.x,
```

```
            maxSpeedChange);
        _velocity.z = Mathf.MoveTowards(_velocity.z, desiredVelocity.z,
            maxSpeedChange);

        var displacement = _velocity * Time.deltaTime;
        var newPosition = transform.localPosition + displacement;

        if (newPosition.x < _allowedArea.xMin)
        {
            newPosition.x = _allowedArea.xMin;
            _velocity.x = -_velocity.x * _bounciness;
        }

        if (newPosition.x > _allowedArea.xMax)
        {
            newPosition.x = _allowedArea.xMax;
            _velocity.x = -_velocity.x * _bounciness;
        }

        if (newPosition.z < _allowedArea.yMin)
        {
            newPosition.z = _allowedArea.yMin;
            _velocity.z = -_velocity.z * _bounciness;
        }

        if (newPosition.z > _allowedArea.yMax)
        {
            newPosition.z = _allowedArea.yMax;
            _velocity.z = -_velocity.z * _bounciness;
        }

        transform.localPosition = newPosition;
    }
}
```



3 Exerciții

3.1 Exercițiul 1

Modificați pereții din față, respectiv spate, astfel încât aceștia să se unească perfect cu pereții din stânga/dreapta (să nu mai existe golurile din colțuri).

3.2 Exercițiul 2

Adăugați binding-uri pentru a controla bila folosind și săgețile de pe tastatură. Dacă aveți la dispoziție un controller, adăugați binding-uri și pentru acesta.

3.3 Exercițiul 3

Adăugați o acțiune pentru sărit și implementați această mecanică.