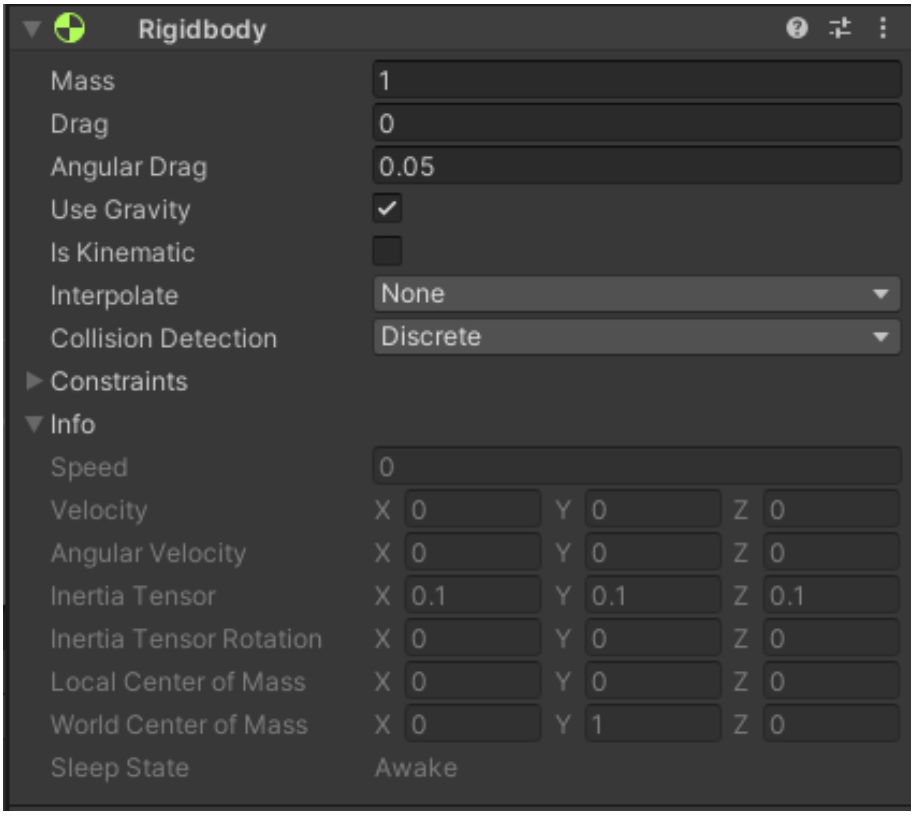
Laboratorul 4

1 Fizici

Până acum, am controlat manual mișcarea bilei și am simulat fizici de bază. În acest laborator ne vom folosi de funcționalitățile implementate în *Unity* pentru fizici. În exemplul acestui laborator, jocul cu bila, fizicile implementate în laboratorul anterior sunt suficiente. Totuși, pentru o aplicație în care este nevoie de interacțiuni fizice mai complexe, este de peferat să folosim sistemul de fizici din *Unity* (*NVIDIA PhysX*). Acest sistem de fizici este performant și are multe capabilități implementate. Vom continua să lucrăm în proiectul început în laboratorul anterior. Codul este disponibil aici.

1.1 Componenta *Rigidbody*

Componenta *Rigidbody* se ocupă cu simularea fizicilor obiectului asupra căreia este atașată. Am folosit această componentă și în primul laborator, dar nu am intrat în detalii. Vom adăuga o componentă de tip *Rigidbody* asupra sferei noastre.

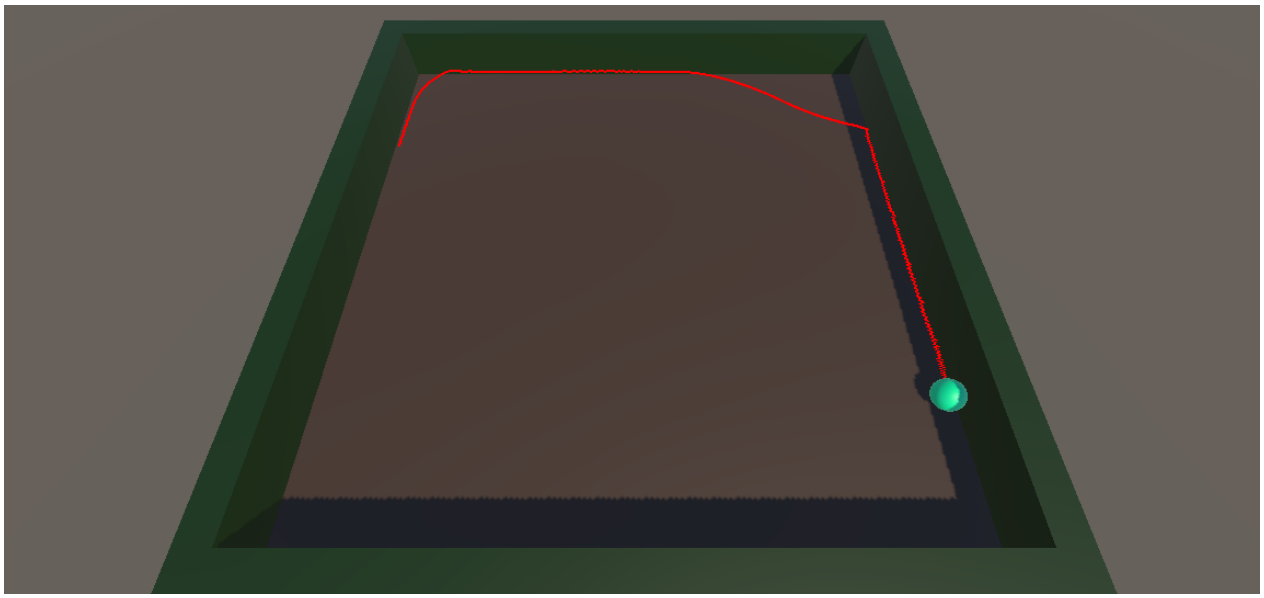


Această componentă se ocupă automat de detectarea coliziunilor, deci putem elimina codul scris în laboratorul precedent pentru verificarea coliziunilor cu cei 4 pereți.

```
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;
    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;

        transform.localPosition = newPosition;
    }
}
```

Dacă rulăm aplicația, vom observa că avem în continuare coliziuni cu pereții, dar acestea nu mai sunt la fel de bune ca înainte. Asta se datorează faptului că noi modificăm manual poziția sferei, în același timp în care aceasta este modificată de către *Rigidbody*, rezulatele fiind neplăcute. (La coliziune, sistemul de fizici împinge bile în afara peretelui, dar variabila noastră *_velocity* își păstrează valoarea)



1.2 Viteza *Rigidbody*-ului

Pentru nu ne lupta cu sistemul de fizic din Unity, trebuie să nu mai modificăm direct poziția sferec, ci să folosim componenta *Rigidbody* pentru a interacționa cu sfera. În cod, putem obține o referință către componenta de tip *Rigidbody* a obiectului folosind metoda *GeComponent* <>. Această metodă este una de tip *template*, care primește ca tip de date tipul componentei căutate.

```
private Rigidbody _rigidbody;
...
private void Awake() =>
    _rigidbody = GetComponent<Rigidbody>();
```

Dacă obiectul nu conține o componentă de tip *Rigidbody*, *GetComponent* < *Rigidbody* > () va returna *null*. Pentru a ne asigura că obiectul asupra căruia scriptul este atașat conține o componentă de tip *Rigidbody* vom folosi [*RequireComponent*] deasupra declarației clasei.

```
[RequireComponent(typeof(Rigidbody))]
public class MovingSphere : MonoBehaviour
```

În loc de a modifica direct poziția sferei, vom modifica proprietatea *velocity* a componentei *Rigidbody*. Intern, compoenta *Rigidbody* va face calculele pentru modificarea poziției folosind această viteză setată.

```
//var displacement = _velocity * Time.deltaTime;
//var newPosition = transform.localPosition + displacement;
//transform.localPosition = newPosition;
_rigidbody.velocity = _velocity;
```

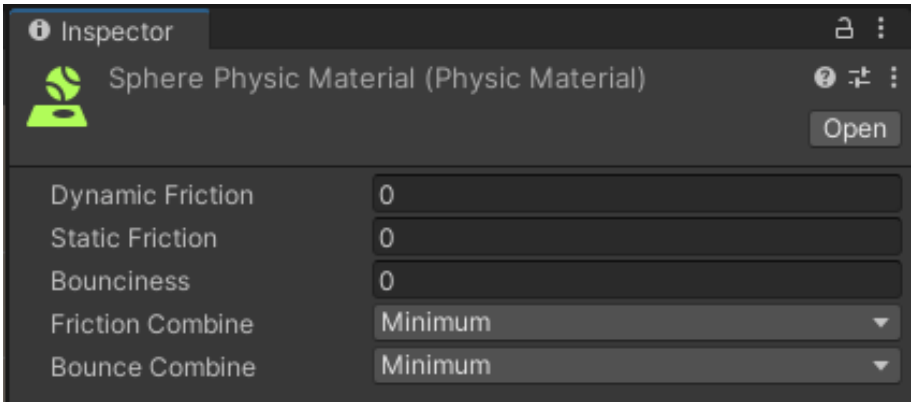
Această soluție ar merge dacă interacțiunile fizice nu ar afecta *velocity*-ul componentei *Rigidbody*. Totuși, acesta este influențat de mediu, iar aceste schimbări nu sunt reflectate și în variabila *_velocity*. Pentru a repara acest lucru, trebuie ca înainte de a calcula noul *velocity*, să se pornească de la valoarea *velocity* a componentei *Rigidbody*.

```
private void Update()
{
    _velocity = _rigidbody.velocity;
    ...
```

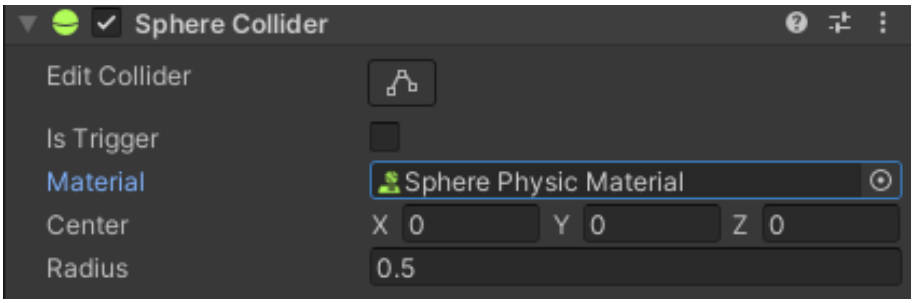
În exemplul nostru, modificăm valoarea proprietății *velocity* a componentei *Rigidbody* direct. De regulă este recomandat să se folosească metode precum *AddForce*. În cazul nostru, valoarea nu este modificată brusc, ci doar se alterează puțin valoarea existentă. Cât timp proprietatea *velocity* este alterată într-un mod controlat și natural, este în regulă să fie modificată direct.

1.3 Mișcare fără frecare

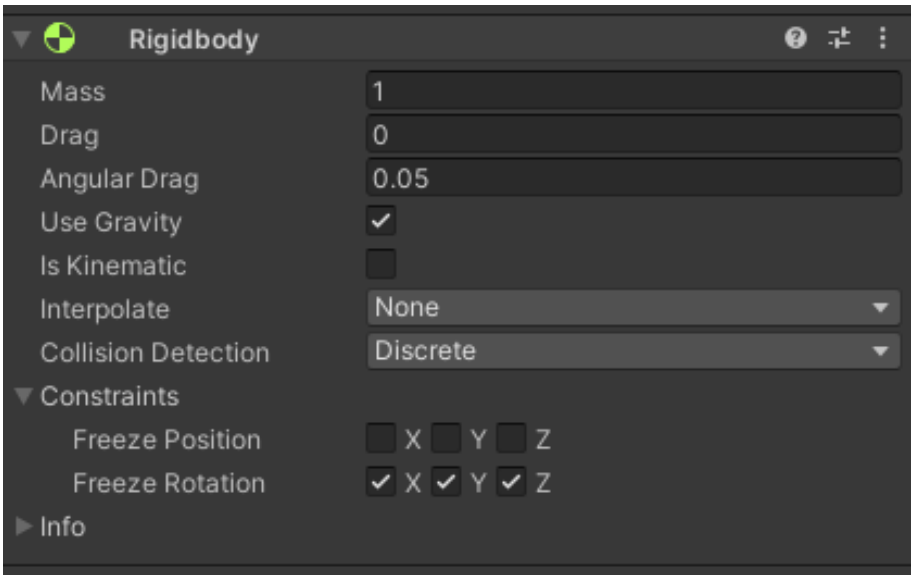
Acum că folosim sistemul de fizici din Unity, rezultatele par similare cu ce am obținut în laboratorul trecut. Totuși, acum bila nu mai ajunge la viteza maximă. Asta se întâmplă deoarece sistemul de fizici din Unity adaugă o forță de frecare atunci când bila se mișcă, fapt care o face să încetinească. Pentru a rezolva această problemă este nevoie să creăm un *Physic Material*. Vom crea un director nou, numit *Physics*. În interiorul acestui director vom crea un nou *Physic Material* (*click dreapta/Create/Physic Material*) numit *Sphere Physic Material*. În inspector, vom seta toate valorile acestui material cu 0, iar valorile modurilor de combinare le vom seta pe *Minimum*.



Pentru a atribui acest material sferei noastre, acesa trebuie setat în field-ul *Material* al componentei *Sphere Collider* atașat sferei.



Acum, dacă dăm *Play*, sfera va ajunge la viteza maximă deoarece forța de frecare a fost eliminată datorită materialului creat anterior. Deși am atribuit materialului valoarea 0 pentru *Bounciness*, sfera încă are un mic bounce la atingerea cu pereții. Asta se datorează sistemului de fizici care mișcă bila puțin în afara coliziunii după ce aceasta a fost detectată. Modificând manual viteza sferei, este posibil ca în unele cazuri, sfera să se miște prea mult, trecând de un obstacol, fără ca acea coliziune să fie detectată. Modul de tratare al acestui caz poate fi setat folosind field-ul *Collision Detection* al componentei *Rigidbody*. În exemplul nostru nu ne interesează acest aspect. Folosind acest material, observăm că bila nu se mai rotește atunci când se mișcă. Pentru a ne asigura că rotația nu va fi deloc alterată de sistemul de fizici, putem bifa căsuțele X, Y și Z ale proprietății *Freeze Rotation* a componentei *Rigidbody*.



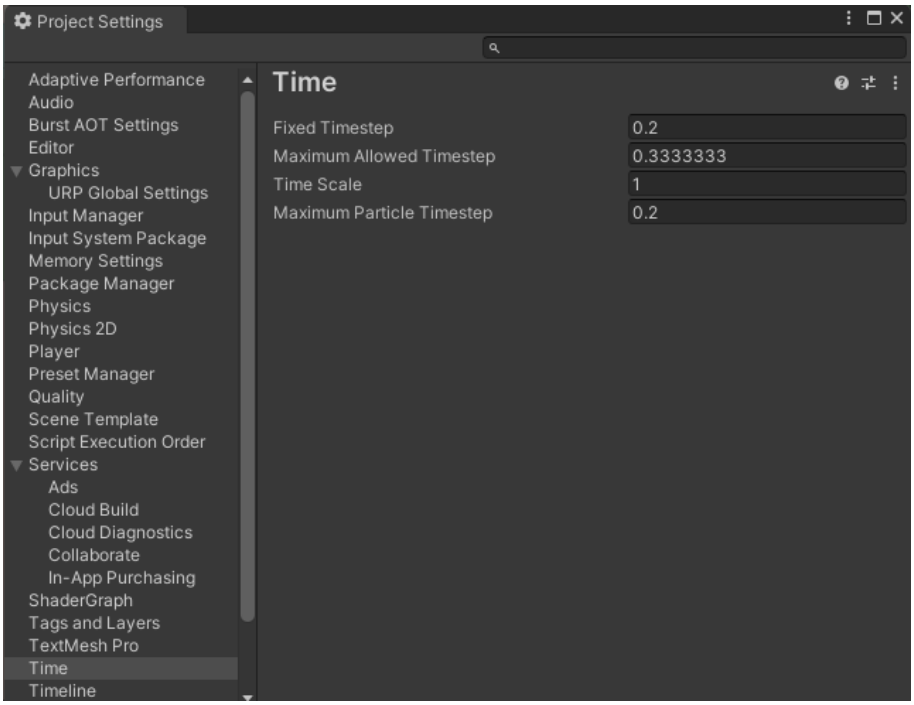
1.4 Fixed Update

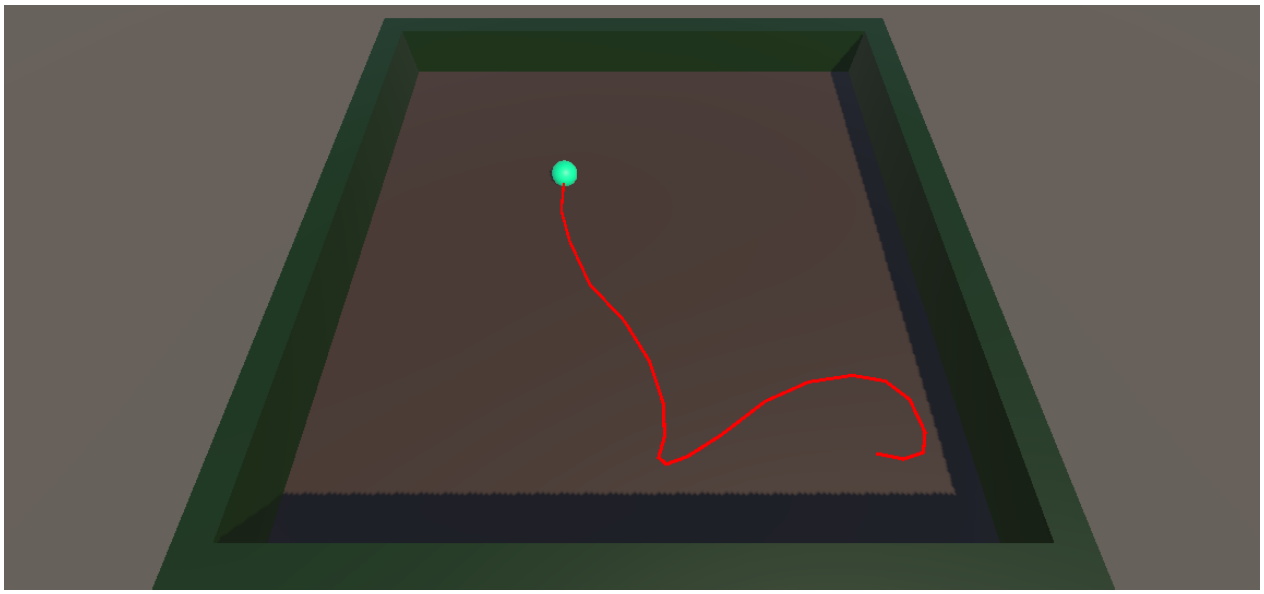
Sistemul de fizici efectuează calculele o dată la un interval fix de secunde (implicit 0.02 secunde). Sistemul de fizici este independent de framerate-ul aplicației. Pentru a obține cele mai bune rezultate ale mișcării, trebuie ca atunci când interacționăm cu componenta *Rigidbody* să ne sincronizăm cu sistemul de fizici. În *Unity* există o metodă specială numită *FixedUpdate* care se execută fix înaintea unui pas de simulare a fizicilor. În update vom păstra doar codul care calculează valoarea dorită pentru *velocity*, iar în *FixedUpdate* vom modifica valoarea proprietății *velocity* a componentei *Rigidbody*.

```
using UnityEngine;
using UnityEngine.InputSystem;

[RequireComponent(typeof(Rigidbody))]
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;
    private Vector3    _velocity        = Vector3.zero,
                       _desiredVelocity = Vector3.zero;
    private Rigidbody  _rigidbody;

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

    private void Awake() =>
        _rigidbody = GetComponent<Rigidbody>();

    private void Update() =>
        _desiredVelocity = new Vector3(_movement.x, 0.0f, _movement.y) *
                           _maxSpeed;

    private void FixedUpdate()
```

```
    {
        _velocity = _rigidbody.velocity;

        var maxSpeedChange = _maxAcceleration * Time.deltaTime;

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

        _rigidbody.velocity = _velocity;
    }
}
```

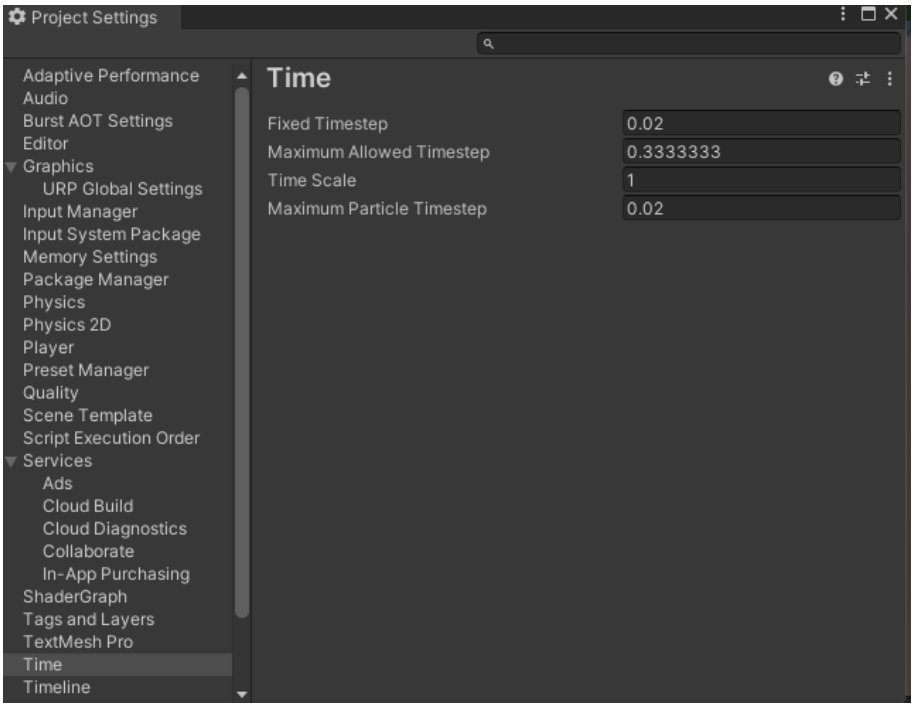
Se observă că în interiorul funcției *FixedUpdate* am folosit *Time.deltaTime* pentru mișcare, deși funcția *FixedUpdate* se execută independent față de *Update*, iar timpul scurs între executările acestei metode este diferit de timpul scurs între execuțiile metodei *Update*. În Unity există proprietatea $Time.fixedDeltaTime$ care are valoarea egală cu timpul setat pentru simularea fizicilor. Acea proprietate este menită să fie folosită în interiorul metodei *FixedUpdate*. În interiorul metodei *FixedUpdate*, *Time.deltaTime* este echivalentă cu $Time.fixedDeltaTime$, deci codul nu trebuie modificat. În funcție de framerate-ul aplicației, *FixedUpdate* se poate executa de zero, o dată sau mai multe ori per executarea metodei *Update*. Prima dată se invocă apelurile metodei *FixedUpdate*, apoi *Update*, iar apoi frame-ul este desenat pe ecran. Dacă funcția *FixedUpdate* se execută mult prea rar, atunci este posibil ca mișcarea să pară sacadată, deși aplicația rulează bine. Vom exemplifica asta setând valoarea *Fixed Timestep*-ului să fie egală cu 0.2 (*Edit/Projec Settings/Time/Fixed Timestep*).





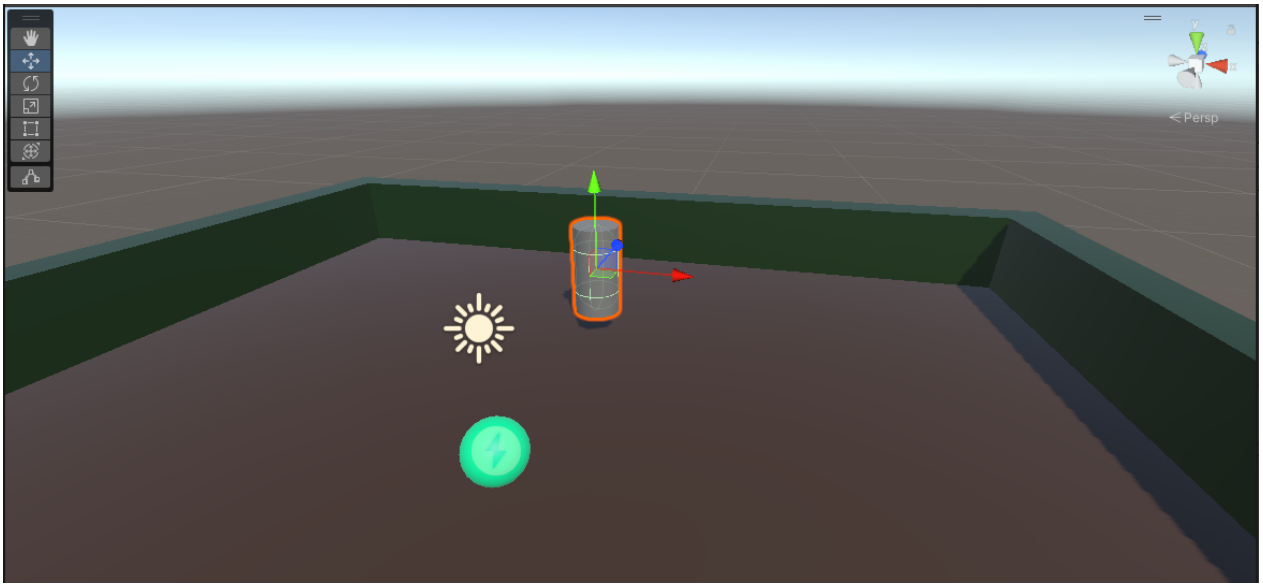
În acest moment, mișcarea este mult mai sacadată, chiar dacă framerate-ul aplicației este unul bun. Pentru a face mișcarea să fie fină și în acest caz, putem schimba valoarea field-ului *Interpolate* al componentei *Rigidbody* din *None* în *Interpolate*. Asta va face ca între execuțiile lui *FixedUpdate* sfera poziția sferei să fie interpolată între poziția anterioară și cea curentă. Mișcarea va fi fină, dar poziția bilei va fi puțin ăn urma poziției la care trebui să fie cu adevărat.

Mai există o valoare posibilă pentru field-ul *Interpolate* și anume *Extrapolate*, care va face ca bila să se miște în continuare folosind viteza *velocity* între apelări ale metodei *FixedUpdate*. Această metodă este bună pentru obiectele a cărot valoare pentru *velocity* este aproape constantă. Nu este cazul exemplului nostru, deci vom lăsa valoarea setată pe *Interpolate*. Vom seta din nou *Fixed Timestep* la valoarea sa anterioară (și *Maximum Particle Timestep*, a carei valoare s-a modificat automat la schimbarea valorii *Fixed Timestep*).

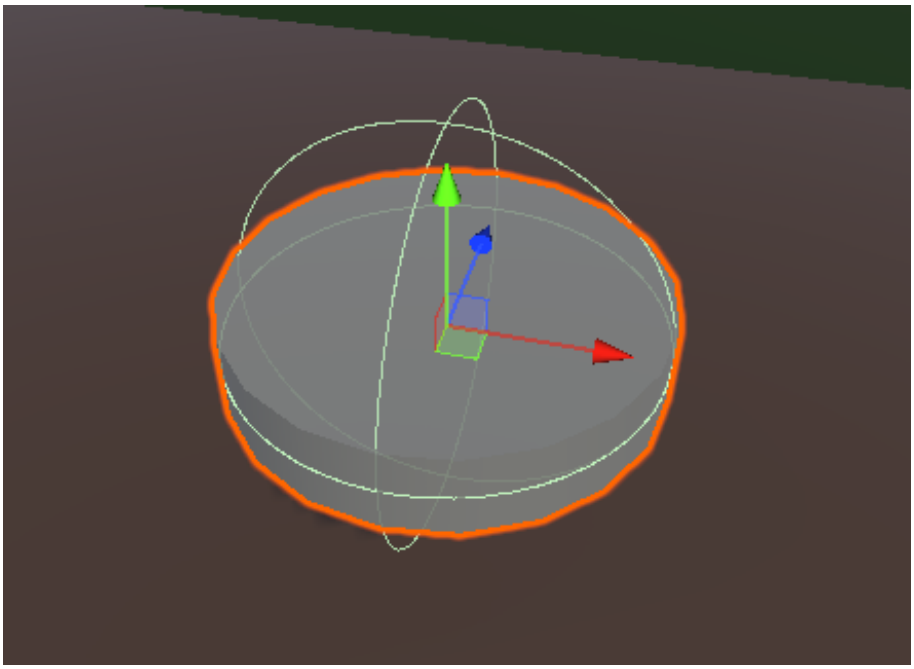


2 *Pickup*-uri

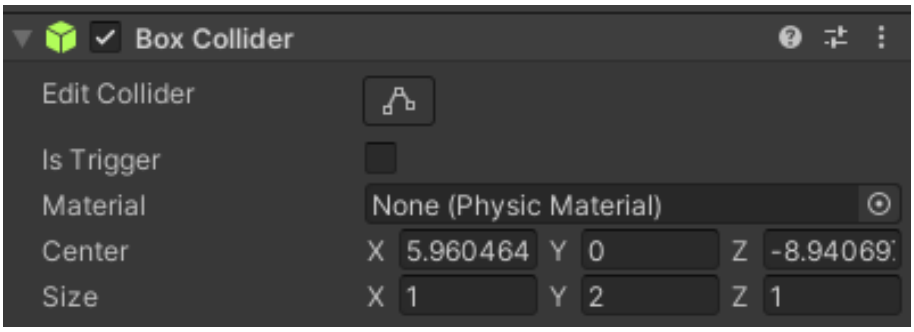
În acest joc, jucătorul va trebui să adune cât mai mulți bănuți într-in numit interval de timp. Acești bănuți vor apărea în lume sub formă de *Pickup*-uri. Atunci când jucătorul atinge un *Pickup* se întâmplă o acțiune determinată de tipul *Pickup*-ului atins. Vom crea un Pickup care să reprezinte un bănuț. Pentru asta vom crea un cilindru (*Create*/*3D Object*/*Cylinder*) pe care îl vom numi *Coin*. Pentru a ne fi ușor să-l vedem, îl vom muta la poziția $(0, 1, 5)$.

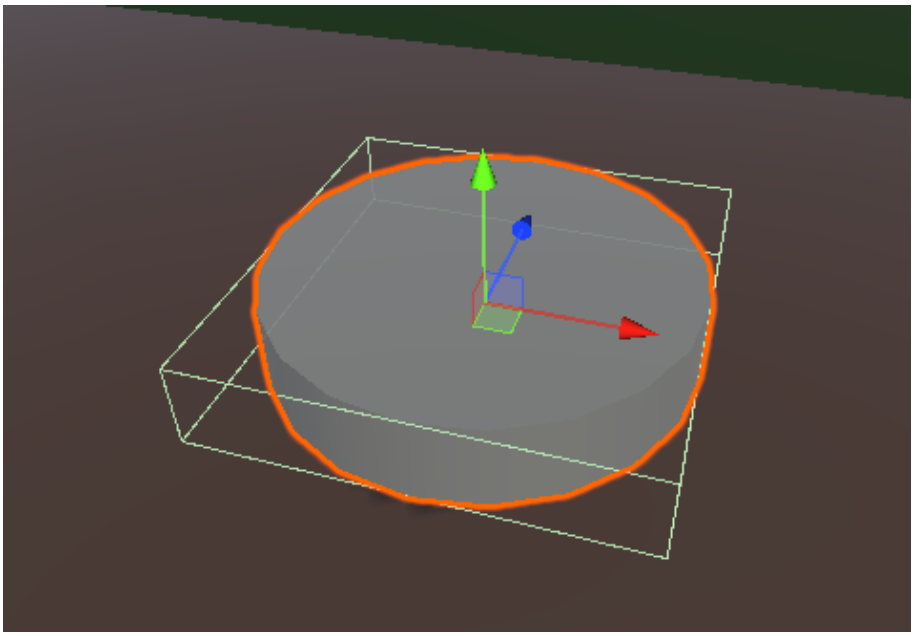


Pentru a-l face să aibă formă de bănuț, acesta trebuie "turtit". Putem face acest lucru prin setarea valorii *Scale* pe axa Y cu 0.1.

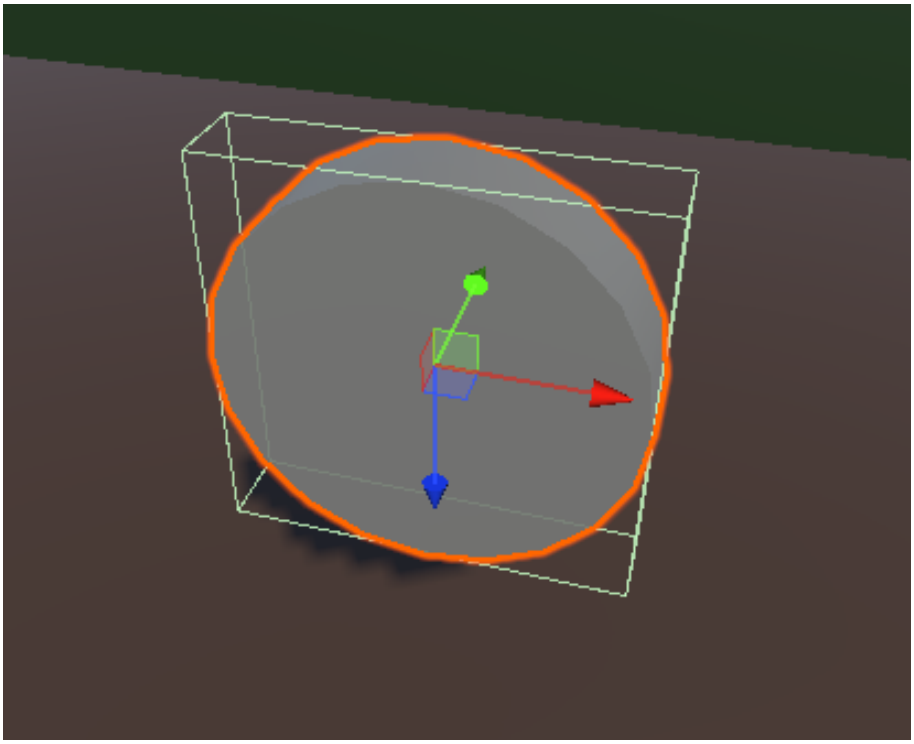


Se observă ca acum collider-ul obiectului acoperă mai mult decât întregul bănuț. Asta se datorează formei lui *Capsule Collider* care conține sfere în capetele capsulei. Pentru a estima mai bine forma bănuțului vom înlocui componenta *Capsule Collider* cu o componentă *Box Collider*.

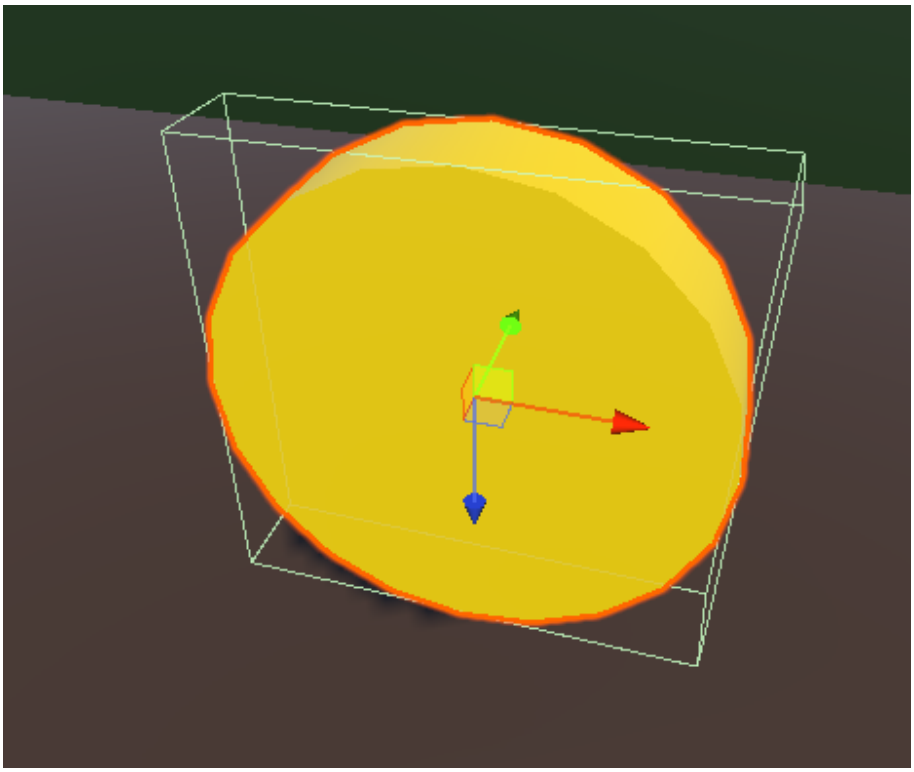




Am dori ca bănuțul să "stea în picioare". Pentru asta, îi vom aplica o rotație de 90 de grade de-alungul axei X.



Îi vom atribui și un material nou, cum am făcut în laboratoarele anterioare, pentru a îi seta o altă culoare. Materialul se va numi *Coin Material*.



Dacă rulăm aplicația, vom observa că obiectul se comportă ca un obstacol prin care bila nu poate trece. Acesta este comportamentul pe care îl dorim momentan. Vom crea un script nou, numit *Pickup* în interiorul directorului *Scripts*. Momentan, acest script se va ocupa doar cu rotirea obiectului creat de-alungul axei Z cu o viteză care poate fi setată în inspector (cum am văzut în laboratorul 2):

```
using UnityEngine;

public class Pickup : MonoBehaviour
{
    [SerializeField]
    private float _rotationSpeed = 100.0f;

    private void Update() =>
        transform.Rotate(0.0f, 0.0f, -_rotationSpeed * Time.deltaTime);
}
```

Obiectul este rotit în jurul axei Z, deoarece aceasta este o rotație relativă la orientarea curentă a obiectului. Deoarece i-am setat o rotație de 90 de grade de-alungul axei X, axa $forward$ (care corespunde lui Z) are orientarea în jos, paralelă cu Y din world space. Rotația este ngativă deoarece direcția lui $forward$ este "în jos", deci direcția de rotație este inversată (vezi Laboratorul 2). Vom atașa acest script asupra obiectului *Coin*, iar acesta se va roti constant în timp ce rulează jocul.

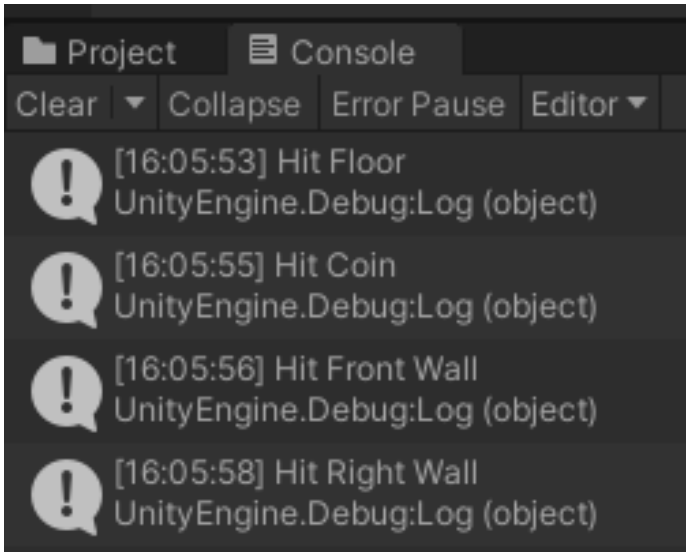
2.1 Detectarea coliziunilor

Vom dori să detectăm momentul în care bila atinge bănuțul. Pentru acest eveniment există o metodă *OnCollisionEnter*, care este apelată de fiecare dată când un obiect care conține componenta *Rigidbody* atinge un obiect.

```
private void OnCollisionEnter() =>
    Debug.Log("Hit something.");
```

De cele mai multe ori, dorim să aflăm informații despre obiectul cu care s-a făcut coliziunea. Pentru a afla aceste informații, este nevoie să declarăm un argument de tip *Collision* acestei metode. În acest argument vom primi detalii despre coliziune:

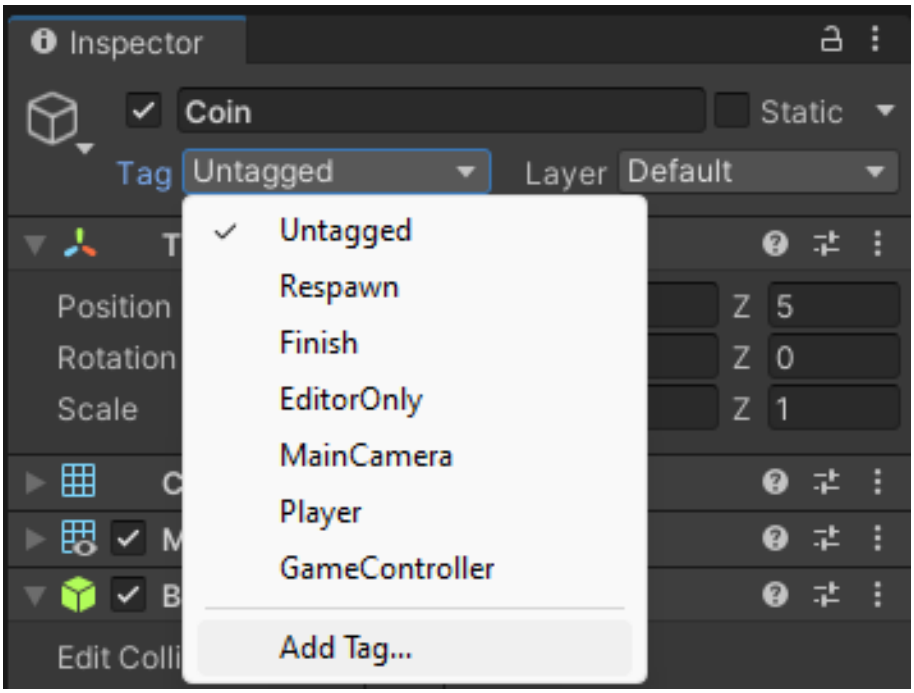
```
private void OnCollisionEnter(Collision collision) =>
    Debug.Log("Hit " + collision.gameObject.name);
```



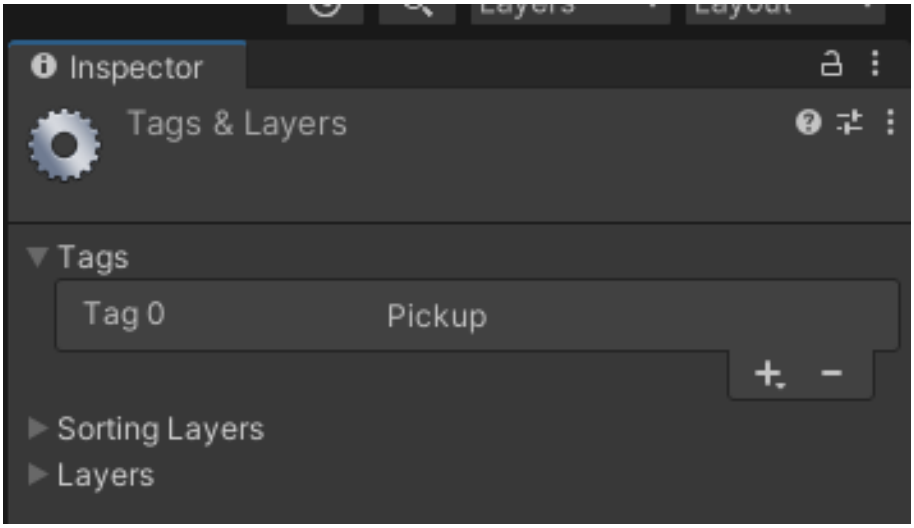
Pentru a implementa sistemul de *Pickup*-uri descris mai sus, ne dorim ca atunci când se realizează o coliziune să detectăm dacă celălalt obiect a fost un *Pickup*, iar în acel caz să apelăm o metodă specifică acelui *Pickup*. Vom defini o metodă publică numită *Collect* în interiorul clasei *Pickup*.

```
public void Collect() =>
    Debug.Log("Pickup collected.");
```

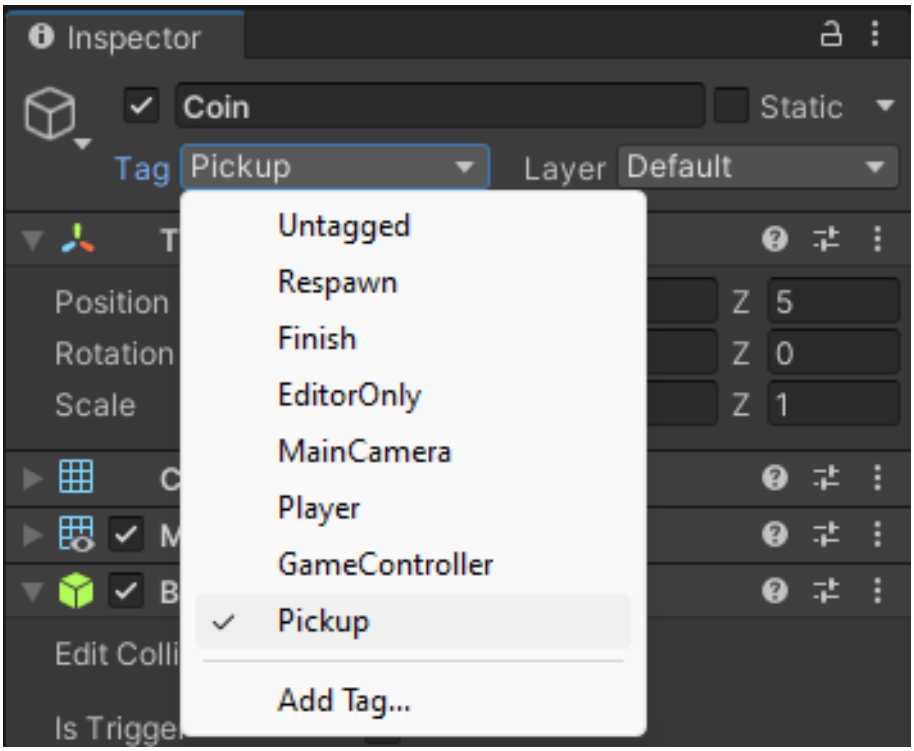
Pentru a detecta dacă obiectul atins este un *Pickup* ne vom folosi de *Tag*-uri. Un *Tag* este un șir de caractere care determină categoria de obiecte din care face parte un obiect. Implicit, obiectele nu au definite un *Tag*. Vom defini un nou *Tag* numit *Pickup*. Pentru a adăuga un nou *Tag*, vom selecta obiectul *Coin* în inspector, iar în dreptul proprietății *Tag* vom selecta *Add Tag*.



În meniul care se deschide vom găsi o listă numită *Tags*. În această listă vom adăuga un element numit *Pickup* și apoi vom apăsa pe butonul *Save*.



După ce a fost creat, *Tag*-ul trebuie atribuit obiectului *Coin*. Vom selecta acest obiect, iar în dreptul proprietății *Tag* vom selecta *Pickup*.



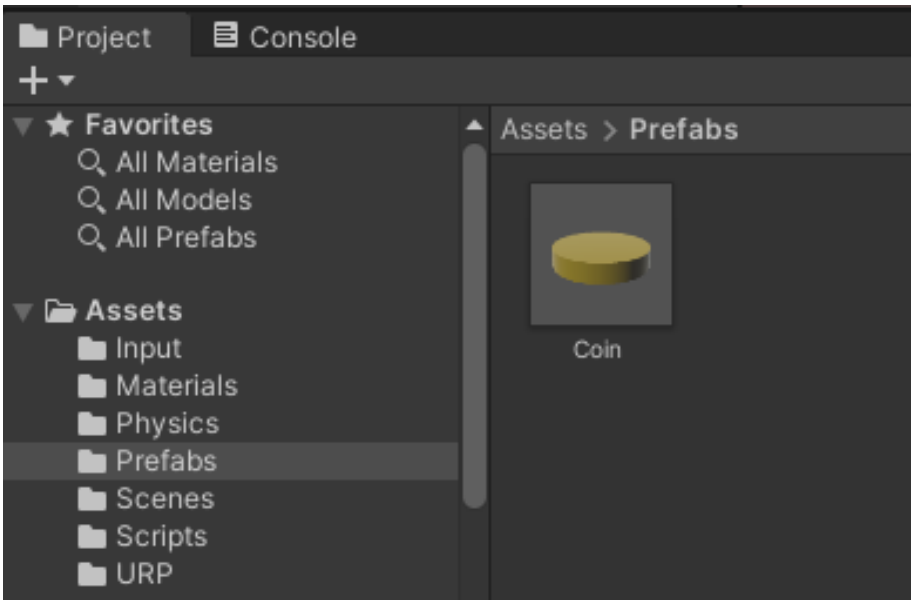
Acum, în interiorul metodei *OnCollisionEnter* putem detecta dacă un *Pickup* a fost atins. Pentru a verifica *Tag*-ul unui obiect putem folosi metoda *CompareTag* din clasa *GameObject*.

```
private void OnCollisionEnter(Collision collision)
{
    var other = collision.gameObject;

    if (other.CompareTag("Pickup"))
    {
        var pickup = other.GetComponent<Pickup>();
        pickup.Collect();
        Destroy(other);
    }
}
```

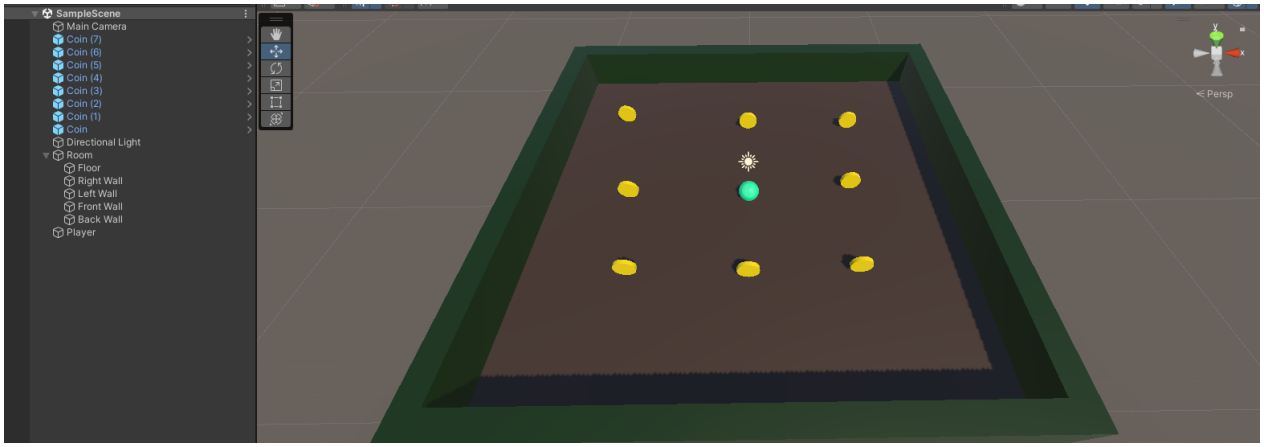
După ce este detectată coliziunea cu un *Pickup* se apelează metoda de colectare a acestuia, iar obiectul este șters din scenă folosind metoda *Destroy*.

2.2 *Prefab*-uri

În acest joc dorim să plasăm în lume *Pickup*-urile folosind script-uri, nu manual. Pentru a plasa un obiect în lume folosind un script este nevoie să avem o reprezentare care definește proprietățile acestuia. Această reprezentare se numește *Prefab*. Vom crea un *Prefab* pentru obiectul *Coin*. În primul rând vom reseta poziția acestuia $(0, 0, 0)$. Vom crea un director nou numit *Prefabs*. Pentru a crea *Prefab*-ul, vom selecta obiectul *Coin* în fereastra *Hierarchy* și îi vom da drag and drop în directorul *Prefabs*.



Vom șterge obiectul *Coin* din scenă. Acest *Prefab* poate fi folosit pentru a adăuga obiecte ca cel anterior în scenă (cu drag and drop în fereastra *Hierarchy* sau în fereastra *Scene*).



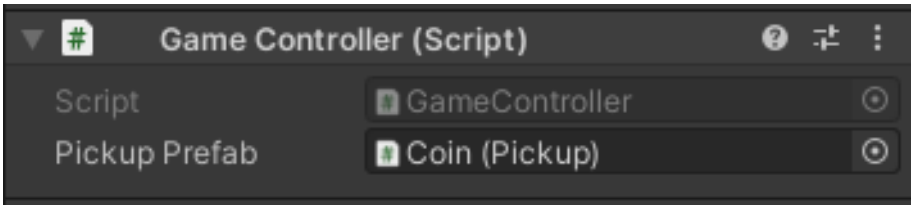
Vom șterge toate monedele instanțiate.

2.3 Instanțiere

Avem nevoie de un script care să plaseze în lume *Pickup*-urile. Vom crea un obiect gol (*clickdreapta/Create Empty*), pe care îl vom numi *GameController*, căruia îi vom reseta componenta *Transform*. Vom crea un nou scrit numit *GameController* și îl vom atașa obiectului creat anterior. În acest script vom declara o referință către *Prefab*-ul pe care dorim să-l plasăm în scenă. Această referință poate fi de tipul *GameObject* sau tipul oricărei componente pe care *Prefab*-ul o conține. În cazul nostru, referința va avea tipul *Pickup*.

```
using UnityEngine;

public class GameController : MonoBehaviour
{
    [SerializeField]
    private Pickup _pickupPrefab = default;
}
```

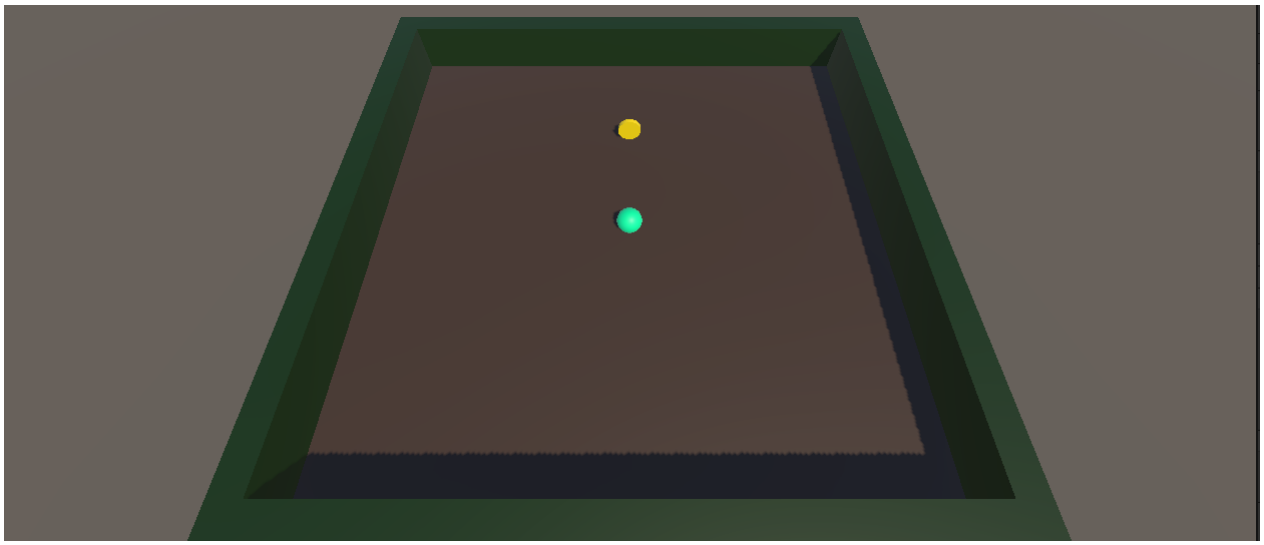
În inspector vom folosi drag and drop pentru a atribui *Prefab*-ul creat anterior field-ului *Pickup Prefab* al componentei *GameController*.



Pentru a adăuga o instanță a unui Prefab în scenă se folosește metoda *Instantiate*, care primește ca argument *Prefab*-ul pe care dorim s-l instanțiem și returnează o referință către obiectul adăugat în scenă. Această metodă plasează obiectul în scenă la poziția specificată de către *Prefab* (în cazul nostru $(0, 0, 0)$). După ce obiectul este instanțiat, putem folosi referința acestuia cum dorim. Vom folosi această referință pentru a muta obiectul la poziția $(0, 1, 5)$.

```
using UnityEngine;

public class GameController : MonoBehaviour
{
    [SerializeField]
    private Pickup _pickupPrefab = default;

    private void Awake()
    {
        var pickup = Instantiate(_pickupPrefab);
        pickup.transform.position = new Vector3(0, 1, 5);
    }
}
```

Când pornim jocul, rezultatul este același ca data trecută, doar că acum obiectul *Coin* este adăugat în scenă folosind un script.



În interiorul acestui script vom lua o referință către obiectul *Floor*. Această referință va fi utilă pentru a determina pozițiile în care se pot plasa Pickup-uri în încăpere. O referință pentru un obiect se declară în același mod ca o referință pentru un *Prefab*.

```
[SerializeField]
private BoxCollider _floor = default;
```

În inspector vom seta valoarea acestei referințe folosind drag and drop (obiectului *Floor* îi dăm drag and drop în field-ul respeciv).

Vom definini o nouă metodă *SpawnPickup* care să plaseze un obiect de tip *Pickup* deasupra podelei la o poziție aleatoare. Această metodă va primi ca argument Prefab-ul și va întoarce obiectul adăugat în scenă.

```
private Pickup SpawnPickup(Pickup prefab)
{
    var pickup = Instantiate(prefab);
    return pickup;
}
```

Pentru a determina dimensiunile podelei ne putem folosi de proprietatea *bounds* a obiectului *_floor*.

```
private Pickup SpawnPickup(Pickup prefab)
{
    var pickup = Instantiate(prefab);
    var bounds = _floor.bounds;
    return pickup;
}
```

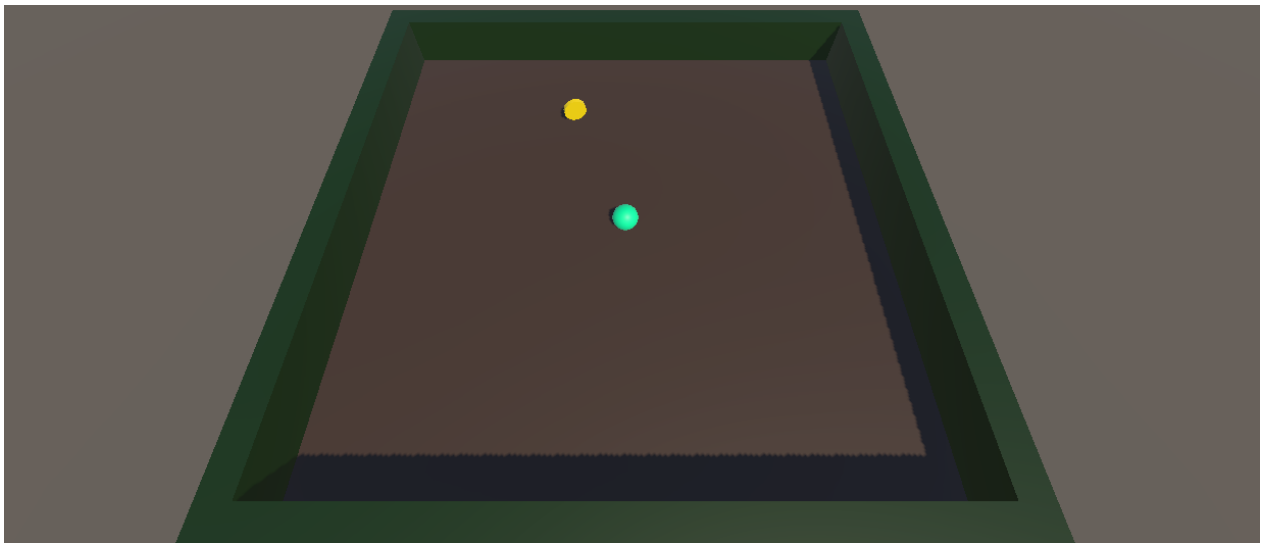
Vom dori ca poziția obiectului să fie una la întâmplare deasupra podelei. În *Unity* putem folosi *Random.value* pentru a obține un număr aleator cuprins între 0 și 1. Știm că poziția de-alungul axei X trebuie să fie un număr cuprins între *bounds.min.x* și *bounds.max.x* (analog pentru axa Z). Funcția $Mathf.Lerp(a, b, t)$ ne este de folos în acest caz. Această funcție returnează un număr cuprins între a și b în funcție de argumentul t, care este un număr cuprins între 0 și 1. Formula din spatele acestei funcții este: $a + t(b - a)$. Deci, atunci când t are valoarea 0, funcția returnează a, iar atunci când t are valoarea 1 returnează b.

```
private Pickup SpawnPickup(Pickup prefab)
{
    var pickup = Instantiate(prefab);
    var bounds = _floor.bounds;

    var position = new Vector3(
        Mathf.Lerp(bounds.min.x, bounds.max.x, Random.value),
        1.0f,
        Mathf.Lerp(bounds.min.z, bounds.max.z, Random.value)
    );

    pickup.transform.position = position;

    return pickup;
}
```

Vom apela această funcție în *Awake*, în locul codului scris anterior.

```
private void Awake() => SpawnPickup(_pickupPrefab);
```

Acum obiectul este plasat la o poziție aleatoare din scenă.



Putem înlocui valoarea hard-codată $1.0f$ a poziției pe axa Y folosind o nouă variabilă.

```
[SerializeField]
private float _pickupRadius = 0.5f;
...
var position = new Vector3(
    Mathf.Lerp(bounds.min.x, bounds.max.x, Random.value),
    bounds.max.y + _pickupRadius,
    Mathf.Lerp(bounds.min.z, bounds.max.z, Random.value)
);
```

De asemenea, este posibil ca *Pickup*-urile să fie plasate în poziții care aring pereții. Pentru a repara acest lucru trebuie să modificăm variabila *bounds*.

```
bounds.size -= new Vector3(_pickupRadius, 0.0f, _pickupRadius) * 2.0f;
```

După ce un *Pickup* este luat și dispare din scenă am dori să apară altul în locul lui. Pentru acest lucru vom folosi o listă statică în interiorul clasei *GameController* care să țină evidența *Pickup*-urilor din scenă. De metoda *Awake* nu mai este nevoie.

```
private static List<Pickup> InstantiatedPickups = new();
```

În interiorul metodei *Update* vom verifica dacă această listă este goală, iar daca nu, atunci vom instanția un nou *Pickup* pe care îl vom adăuga listei.

```
private void Update()
{
    if (InstantiatedPickups.Count < 1)
        InstantiatedPickups.Add(SpawnPickup(_pickupPrefab));
}
```

Atunci când *Pickup*-urile sunt distruse, referințele lor vor rămâne în lista *InstantiatedPickups*, deci nu se vor adăuga *Pickup*-uri noi în scenă. Pentru a rezolva asta, trebuie ca atunci când obiectele sunt distruse, acestea să fie eliminate din listă. Vom defini o metodă statică *UnregisterPickup* care va scoate un obiect din această listă.

```
public static void UnregisterPickup(Pickup pickup)
{
    Debug.Assert(
        InstantiatedPickups.Contains(pickup),
        "Unregistration of unknown pickup!", pickup);
```

```
    InstantiatedPickups.Remove(pickup);
}
```

Acestă funcție folosește *Debug.Assert* pentru a afișa un mesaj de eroare în cazul în care se dorește eliminarea din listă a unui element care nu se află în listă. În script-ul *Pickup* vom defini metoda *OnDisable* care este apelată de *Unity* atunci când componenta este dezactivată, sau când obiectul este distrus. În această metodă vom apela metoda *UnregisterPickup* definită anterior pentru a elimina referința obiectului din lista *InstantiatedPickups*.

```
private void OnDisable() =>
    GameController.UnregisterPickup(this);
```

Analog, putem defini o metodă *RegisterPickup* care să fie apelată atunci când obiectul este creat.

```
public static void RegisterPickup(Pickup pickup)
{
    Debug.Assert(!InstantiatedPickups.Contains(pickup),
        "Duplicate registration of pickup!", pickup);

    InstantiatedPickups.Add(pickup);
}
```

Nu vom mai adăuga manual obiectele în listă în *Update* și vom face ca funcția *SpawnPickup* să nu returneze nimic.

```
private void Update()
{
    if (InstantiatedPickups.Count < 1)
        SpawnPickup(_pickupPrefab);
}

private void SpawnPickup(Pickup prefab)
{
    ...
    //return pickup;
}
```

În clasa *Pickup* vom defini metoda *OnEnable* care este apelată atunci când obiectul este creat sau când componenta respectivă este activată. Această metodă va apela metoda *RegisterPickup* definită anterior.

```
private void OnEnable() =>
    GameController.RegisterPickup(this);
```

3 Exerciții

3.1 Exercițiul 1

Modificați codul astfel încât sa existe mereu în scenă 3 *Pickup*-uri.

3.2 Exercițiul 2

Încercați să faceți ca bila să nu se oprească după ce atinge un *Pickup* (vezi *Triggers*).