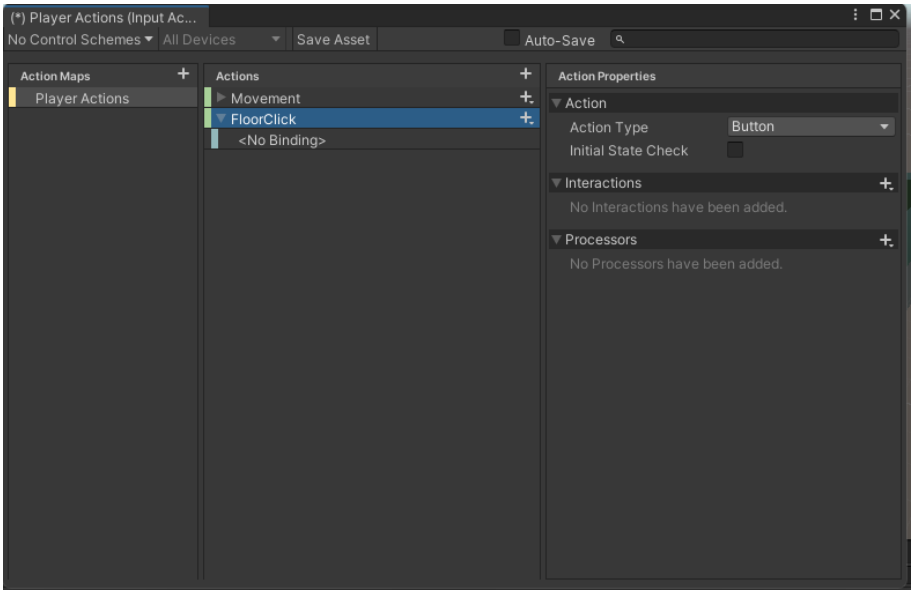
Laboratorul 6

În acest laborator vom continua jocul început în laboratoarele trecute. Codul este disponibil aici.

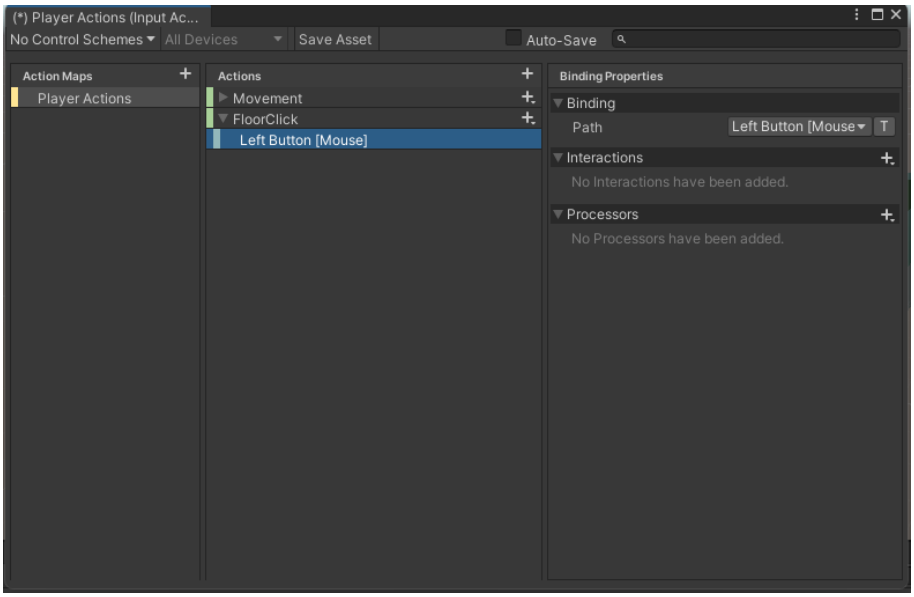
Vom adăuga o nouă modalitate de input pentru joc: atunci când jucătorul apasă click pe o poziție, bila se va deplasa către poziția respectivă.

1 *Click Action*

Vom adăuga o nouă acțiune de input pentru a detecta dacă utilizatorul a apăsat click. Vom numi această acțiune *FloorClick*. Tipul acestei acțiuni va fi *Button* (valoarea implicită), deoarece prin această acțiune dorim doar să detectăm apăsarea *click*-ului.



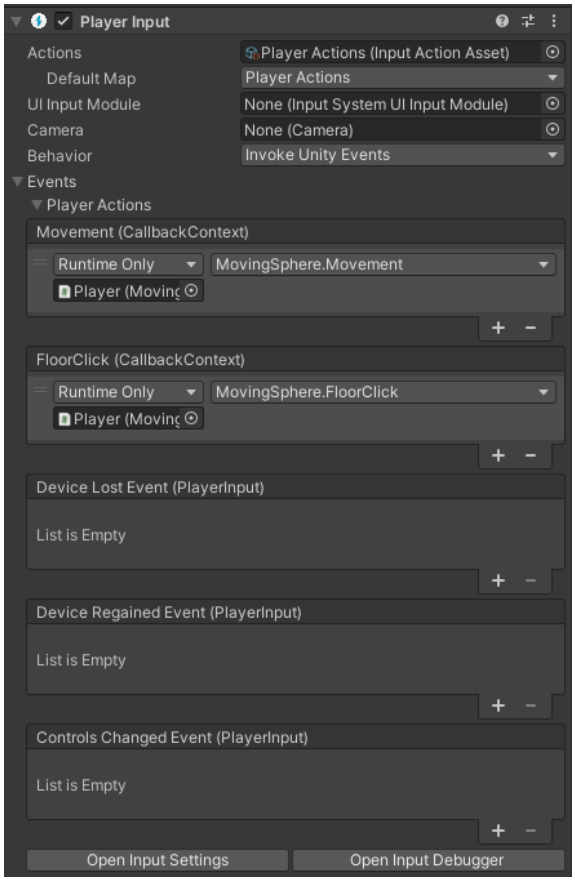
Pentru această acțiune vom adăuga un singur *binding* care detectează apăsarea butonului *click stanga* de pe mouse.



În scriptul *MovingSphere* vom declara evenimentul care dorim să fie apelat atunci când *click*-ul este apăsat.

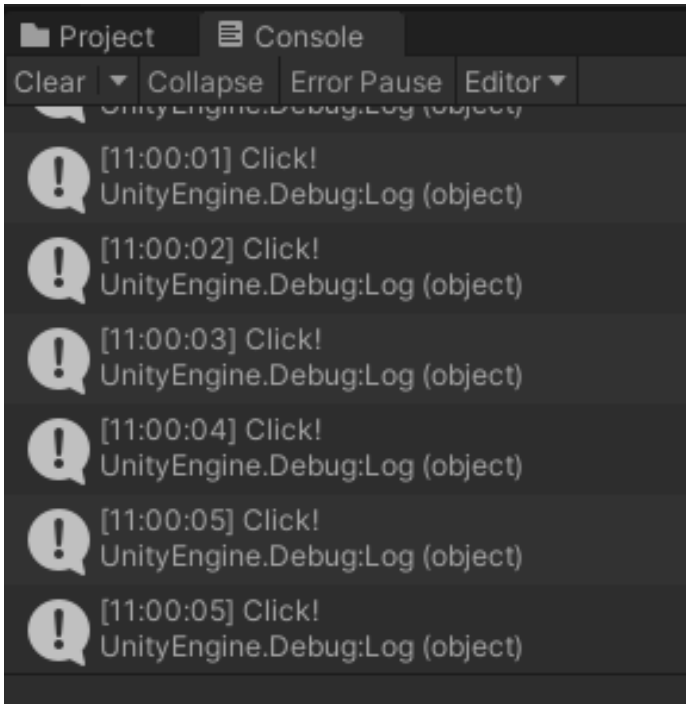
```
public void FloorClick(InputAction.CallbackContext context)
{
}
```

În componenta *Player Input* atașată sferei vom referenția această metodă, cum am procedat și în cazul metodei *Movement*.



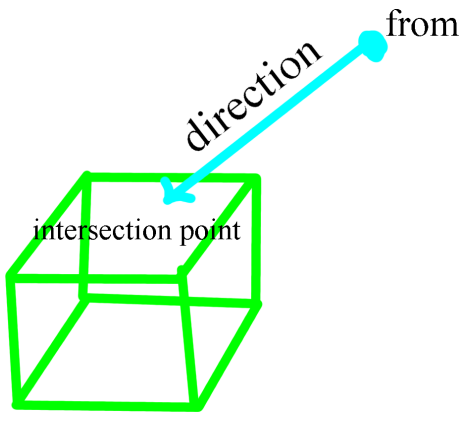
Metoda *FloorClick* este apelată de mai multe ori pe parcursul unui *click* în funcție de stările prin care trece butonul (apăsat, eliberat, etc..). Pentru a detecta doar momentul în care click-ul a fost apăsat, ne putem folosi de proprietatea *started* a variabilei *context*.

```
public void FloorClick(InputAction.CallbackContext context)
{
    if (!context.started)
        return;

    Debug.Log("Click!");
}
```



2 *Raycasting*

Raycasting reprezintă găsirea intersecției dintre o rază și obiectele din lume. Dacă raza se intersectează cu unul sau mai multe obiecte, atunci se consideră doar prima intersecție, și sunt returnate informații despre acea intersecție.

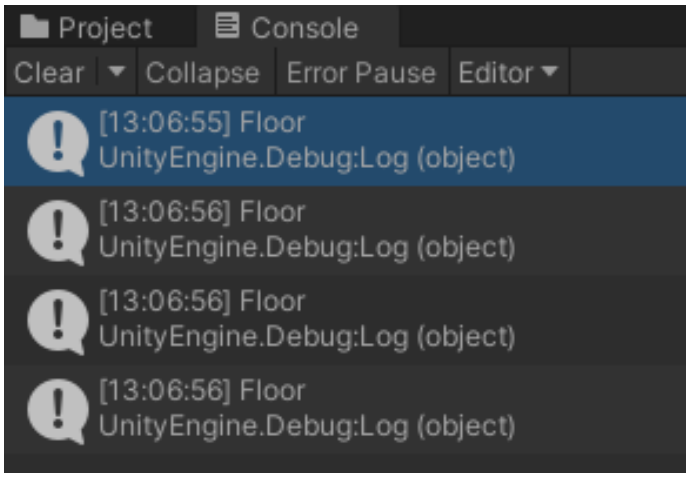


Unity face *raycasting* bazându-se pe colliderele obiectelor, prin urmare, intersecțiile nu respectă neapărat *mesh*-ul obiectului, ci doar *collider*-ul (așa cum se întâmplă și în cazul fizicilor). Pentru a face *Raycasting* se folosește metoda *Physics.Raycast*.

Atunci când apăsăm *click* vom face un *Raycast* dinspre cameră spre lume, și vom afișa pe ecran numele obiectului intersectat.

```
public void FloorClick(InputAction.CallbackContext context)
{
    if (!context.started)
        return;

    var mainCamera = Camera.main;
    var ray = new Ray(mainCamera.transform.position,
                mainCamera.transform.forward);

    if (Physics.Raycast(ray, out var hitInfo))
        Debug.Log(hitInfo.transform.name);
}
```

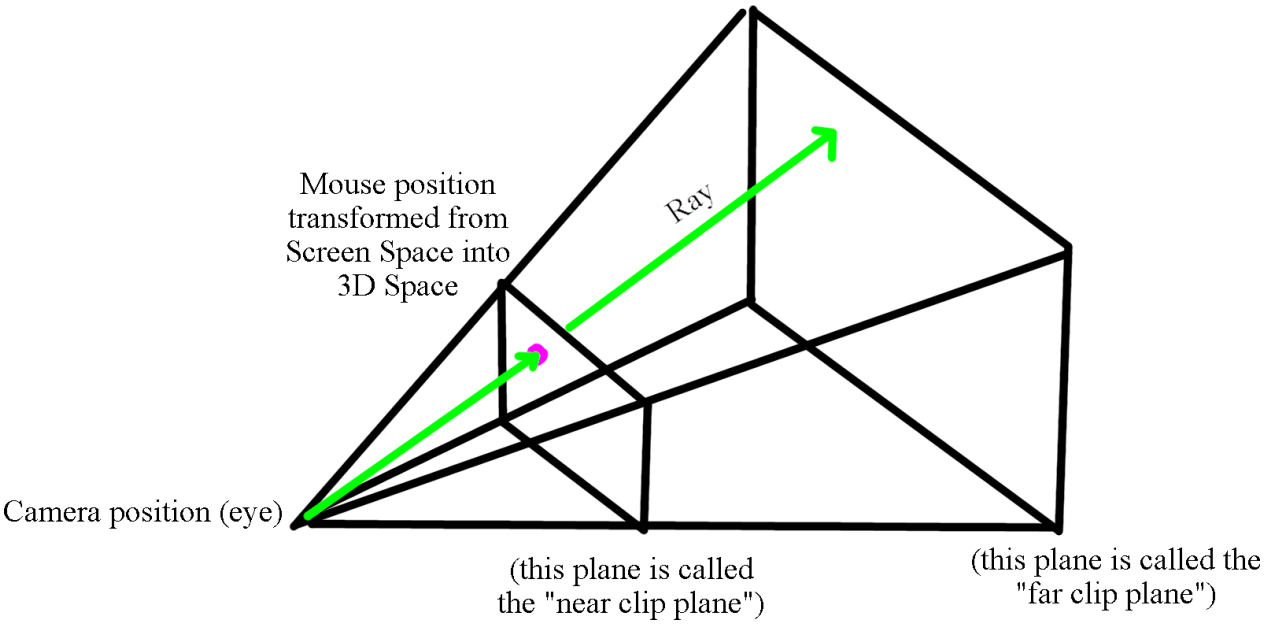
Acum, raza are mereu aceeași direcție, indiferent de poziția mouse-ului pe ecran.



Putem obține poziția mouse-ului în interiorul ferestrei. Această poziție poate fi transformată într-o poziție în spațiul 3D al lumii folosind metodele *Camera.ScreenToWorldPoint* sau *Camera.ViewportToWorldPoint* (în acest caz este nevoie de calcule adiționale), sau poate fi transformată direct într-o rază care pornește dinspre camera înspre lume, trecând prin poziția coresopndentă mouse-ului în spațiul 3D folosind metoda *Camera.ScreenPointToRay*. Vom folosi metoda *Camera.ScreenPointToRay* pentru a determina raza. Pentru a afla poziția mouse-ului folosind noul sistem de *Input* putem folosi *Mouse.current.position*.

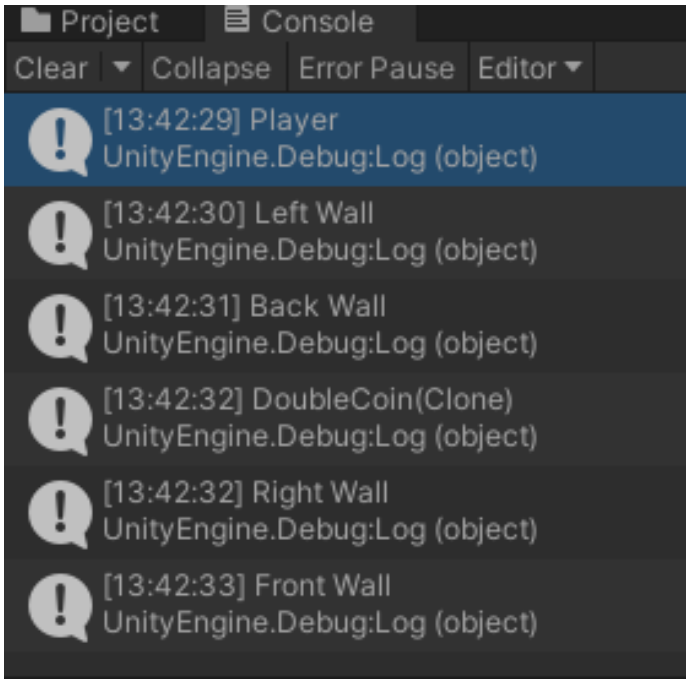
```
...
Vector3 mousePosition = Mouse.current.position.ReadValue();
var ray = mainCamera.ScreenPointToRay(mousePosition);
//var ray = new Ray(mainCamera.transform.position,
//    mainCamera.transform.forward);
...
```

Următoarea figură evidențiază matematica din spatele acestor linii de cod:



Trunchiul de piramidă determinat de planurile *near clip plane* și *far clip plane* reprezintă tot ce este vizibil pentru cameră. *eye* reprezintă poziția camerei, iar *near clip plane* reprezintă cel mai apropiat plan din lume, care este vizibil pentru cameră. Transformată în 3D, poziția mouse-ului se va afla pe acest plan.

Acum, când apăsăm click în lume, în funcție de poziția mouse-ului, în consolă vor apărea numele diferitor obiecte.

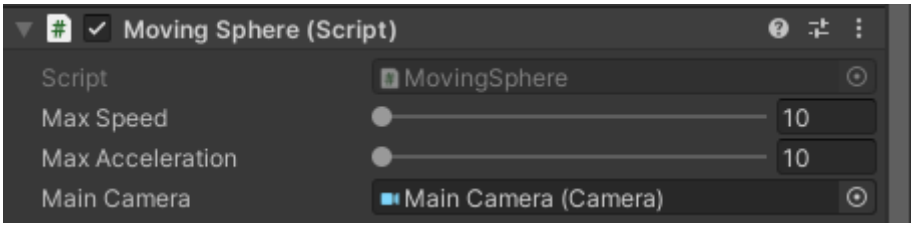


Pentru a obține o referință către cameră, am folosit proprietatea *Camera.main*. Această metodă de a obține o referință către cameră este una ineficientă, deoarece intern, *Unity* face o căutare a obiectelor care au *tag*-ul *MainCamera*. În loc să folosim *Camera.main*, vom adăuga manual o referință în clasa *MovingSphere*.

```
...
[SerializeField]
```

```
private Camera _mainCamera = default;
...
```

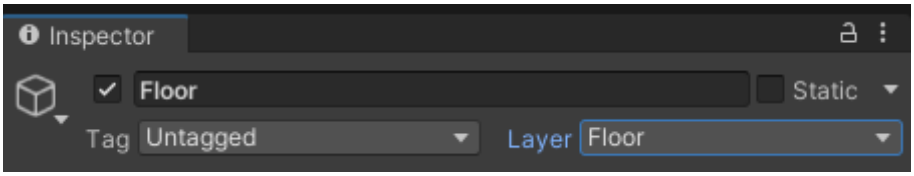
```
...
//var mainCamera = Camera.main;
Vector3 mousePosition = Mouse.current.position.ReadValue();
var ray = _mainCamera.ScreenPointToRay(mousePosition);
...
```



Momentan, *Raycast*-ul de mai sus verifică intersecția dintre rază și toate obiectele din scenă. Pentru acest joc am vrea să aflăm doar dacă raza se intersectează cu podeaua și punctul de intersecție.

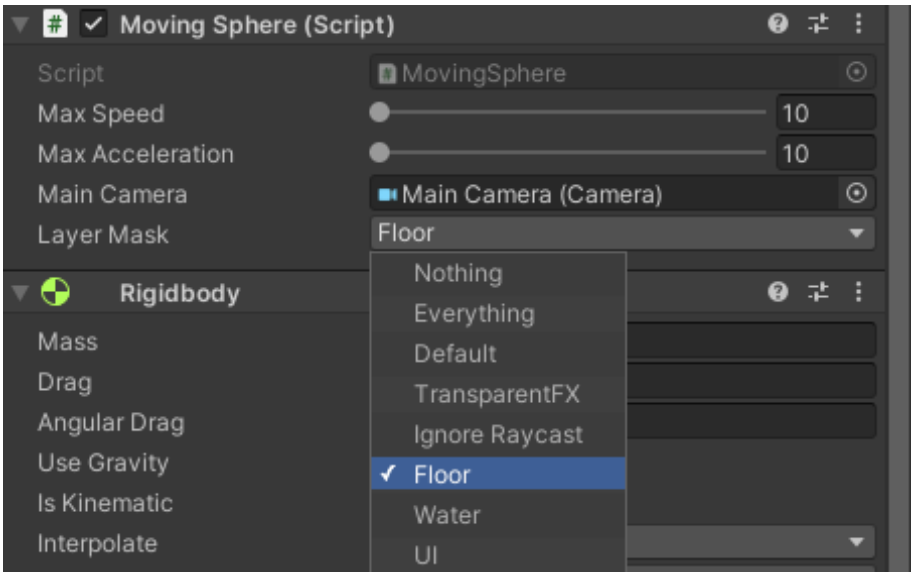
Putem folosi *LayerMask* pentru a specifica metodei *Physics.Raycast* ce obiecte să ignore în verifcarea intersecțiilor. Un obiect de tip *LayerMask* este pur și simplu un *int*, în care fiecare bit reprezintă un *Layer* pe care se află obiecte. Implicit, obiectele au *Layer*-ul *Default*.

Vom crea un nou layer numit *Floor* și îl vom atribui obiectului *Floor* din scenă. Un *Layer* se adaugă similar cu un *Tag* (procesul de adăugare este aproape identic, diferă doar proprietatea *Layers* și faptul că lista de layere nu este dinamică, ci are 32 de elemente).



În scriptul *MovingSphere* vom adăuga o nouă variabilă de tip *LayerMask*, iar din inspector vom face ca acea variabilă să ia în considerare doar obiectele care se află pe *Layer*-ul *Floor*.

```
...
[SerializeField]
private LayerMask _layerMask = -1; // -1 means that all bits are set
                                   // (see Computer Architecture Course)
...
```



Această variabilă o putem transmite acum ca argument metodei *Physics.Raycast*.

```
if (Physics.Raycast(ray, out var hitInfo, (int)_layerMask))
    Debug.Log(hitInfo.transform.name);
```

Dacă rulăm, comportamentul nu va fi cel dorit atunci când apăsăm click, deoarece această variantă supraîncărcată a metodei *Raycast* va face va considera ca al 3-lea argument este distanta maxima de *Raycast*. Vom folosi altă variantă supraîncărcată a metodei pentru a specifica *Layer Mask*-ul.

```
...
private const float _maxDistance = 100.0f;
...
```

```
...
if (Physics.Raycast(ray, out var hitInfo, _maxDistance, _layerMask))
    Debug.Log(hitInfo.transform.name);
...
```

Vom observa că acum doar podeaua este luată în calcul în stabilirea acestei intersecții.

Pentru a face ca bila să se îndrepte către poziția respectivă, vom crea o corutină care se va ocupa de asta.

```
using System.Collections;
...
```

```
...
private IEnumerator GoToPosition(Vector3 targetPosition)
{
    yield return null;
}
...
```

```
...
if (Physics.Raycast(ray, out var hitInfo, _maxDistance, _layerMask))
    StartCoroutine(GoToPosition(hitInfo.point));
...
```

În interiorul acestei corutine, primul pas este să aflăm direcția de deplasare.

```
var initialPosition = transform.position;
var direction = targetPosition - initialPosition;
direction.y = 0.0f;
direction.Normalize();
...
```

Bila nu își va schimba coordonata Y, deoarece se va deplasa pe un plan paralel cu planul XZ. Direcția o normalizăm după ce avem deja un vector care se află pe planul podelei.

La fiecare frame vom verifica dacă bila a ajuns la destinație, iar dacă nu, actualizăm valoarea variabilei *_movement* corespunzător cu direcția calculată anterior.

```
private IEnumerator GoToPosition(Vector3 targetPosition)
{
    ...
    while (!reachedTarget)
    {
        _movement = new Vector2(direction.x, direction.z);
        yield return new WaitForEndOfFrame();
    }
}
```

Putem observa mai multe probleme:

- Direcția de deplasare nu este ideală deoarece aceasta se schimbă atunci când bila începe să se miște.
- Bila depășește destinația.
- Controalele din tasatatură nu mai merg odată ce s-a pornit corutina.
- Dacă apăsăm de două ori click se vor porni două corutine.
- Acest mod de control nu are sens atunci când controalele sunt inversate.

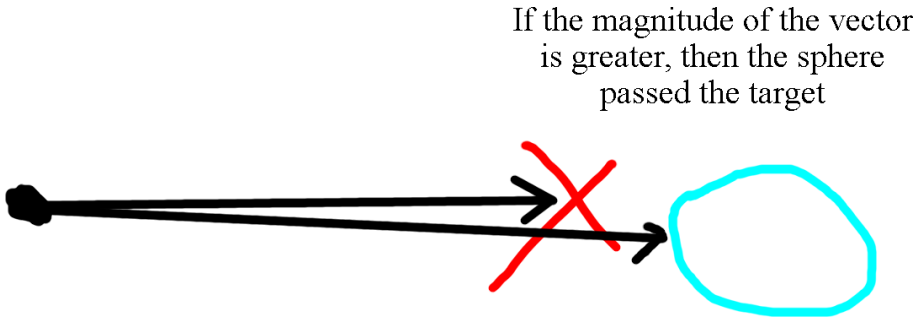
Vom repara pe rând aceste probleme.

Pentru a repara problema direcției, putem calcula direcția direct în interiorul buclei în loc să o calculăm înainte.

```
private IEnumerator GoToPosition(Vector3 targetPosition)
{
    var reachedTarget = false;

    while (!reachedTarget)
    {
        var position = transform.position;
        var direction = targetPosition - position;
        direction.y = 0.0f;
        direction.Normalize();

        _movement = new Vector2(direction.x, direction.z);
        yield return new WaitForEndOfFrame();
    }
}
```

Acum, bila nu mai depășește punctul țintă, dar se tot duce înainte și înapoi în jurul acelui punct. În continuare vom detecta dacă bila a ajuns la destinație. Pentru asta, vom compara lungimea vectorului dinspre poziția inițială a bile către punctul țintă cu lungimea vectorului dinspre poziția inițială a bilei către poziția sa actuală.

```
private IEnumerator GoToPosition(Vector3 targetPosition)
{
    var initialPosition = transform.position;
    var initialVector = targetPosition - initialPosition;
    initialVector.y = 0.0f;
    while (!reachedTarget)
    {
        ...
        var currentVector = position - initialPosition;
        reachedTarget = currentVector.magnitude > initialVector.magnitude;
    }

    _movement = Vector2.zero;
}
```



Ținta încă este depășită, dar nu cu mult. Pentru a face ca bila să se oprească fix în punctul țintă, matematica este mai complicată, și nu face parte din scopul acestui laborator.

Pentru a face ca bila să poată fi controlată din tastatură din nou, putem adăuga o variabilă booleană care să specifice dacă bila va fi controlată de această corutină, sau de către tastatură.

```
private bool _manualMovement;

public void Movement(InputAction.CallbackContext context)
{
    _manualMovement = true;
    _movement = context.ReadValue<Vector2>();
}
```

```
private IEnumerator GoToPosition(Vector3 targetPosition)
{
    ...
    _manualMovement = false;

    while (!reachedTarget && !_manualMovement)
    {
        ...
    }

    if (!_manualMovement) // We need this check so that the sphere
                          // doesn't stop the moment
                          // the keys are pressed.
        _movement = Vector2.zero;
}
```

Pentru a nu avea mai multe corutine rulând simultan, vom proceda în același mod în care am procedat și în Laboratorul 5.

```
...
private Coroutine _movementCoroutine;
...
```

```
public void FloorClick(InputAction.CallbackContext context)
{
    ...
    if (Physics.Raycast(ray, out var hitInfo, _maxDistance, _layerMask))
    {
        if (_movementCoroutine != null)
            StopCoroutine(_movementCoroutine);

        _movementCoroutine = StartCoroutine(GoToPosition(hitInfo.point));
    }
}
```

```
private IEnumerator GoToPosition(Vector3 targetPosition)
{
    ...
    _movementCoroutine = null;
}
```

Deoarece acest mod de control nu are sens când controalele sunt inversate, vom verifica și acest lucru în *while*-ul corutinei.

```
    private IEnumerator GoToPosition(Vector3 targetPosition)
    {
        ...
        while (!reachedTarget && !_manualMovement &&
               !GameController.Instance.InvertControls)
        {
            ...
        }
        ...
    }
```

3 Exerciții

3.1 Implementați din nou deplasarea bilei către puncul țintă, dar fără să folosiți corutine

Hint: use a state machine.