Laboratorul 5

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

1 Comportamente ale *Pickup*-urilor

1.1 Bănuți

Momentan, când un bănuț este colectat, în consolă este afișat mesajul *Pickup collected.*. Comportamentul dorit este ca această acțiune să incrementeze scorul jucătorului. Vom adăuga o proprietate publică și statică care ține evidența scorului în interiorul clasei *GameController*.

```
public static int Score { get; set; }
```

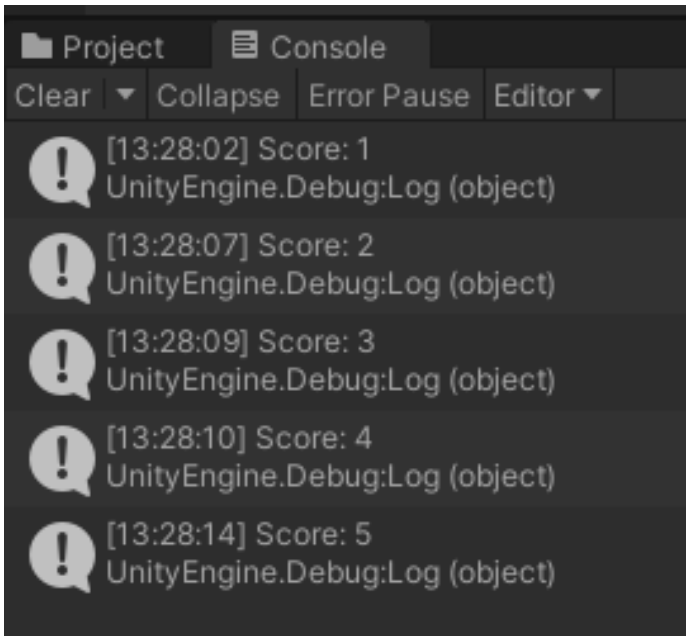
În interiorul clasei *Pickup* vom face ca acea variabilă să fie incrementată de fiecare dată când un bănuț este colectat.

```
public void Collect() =>
    GameController.Score++;
```

În *C#*, o proprietate reprezintă o variabilă care poate avea *getter* și *setter*. Variabila declarată anterior are *getter* și *setter* implict, lucru care o face să se comporte ca o variabilă publică obișnuită. Putem suprascrie *setter*-ul astfel încât să afișeze în consolă noul scor atunci când acesta este apelat.

```
public static int Score
{
    get => _score;
    set
    {
        _score = value;
        Debug.Log("Score: " + _score);
    }
}
...
private static int _score;
```

Observație: Când sunt suprascrise *getter*-ele și *setter*-ele implicite ale proprietii, variabila internă folosită dispare, deci trebuie declarată altă variabilă manual. În *setter*, valoarea nouă este transmisă prin intermediul variabilei *value*.



1.2 Refactorizare

Pentru scor am folosit o variabilă statică, totuși am dori ca datele pe care le folosim să fie parte a instanței *GameController* care se află în scenă. Vom implementa un sistem similar cu un *Singleton*. Vom folosi o proprietate statică *Instance* căreia îi vom atribui valoarea unicului obiect de tip *GameController* din scenă folosind metodele *OnEnable* și *OnDisable*.

```
public static GameController Instance { get; set; }
...
private void OnEnable() =>
    Instance = this;

private void OnDisable() =>
    Instance = null;
```

Vom face ca proprietatea *Score* să fie parte a clesi, eliminând *keyword*-ul *static*.

```
public int Score
{
    ...
}
...
private int _score;
```

În interiorul clasei *Pickup* vom accesa proprietatea *Score* prin intermediul obiectului *GameController.Instance*.

```
public override void Collect() =>
    GameController.Instance.Score++;
```

Lista *InstantiatedPickups* va rămâne obiect static care nu este parte a instanțelor de *GameObject*. Asta se datorează faptului că există posibilitatea ca *Pickup*-uri să fie instațiate în scenă, înainte ca valoarea lui *GameController.Instace* să fie setată.

1.3 Mai multe *Pickup*-uri

În acest joc ne dorim să avem mai multe tipuri de *Pickup*-uri pe lângă bănuți, iar acestea să aibă comportamente diferite. Vom transforma clasa *Pickup* într-o clasă care trebuie suprascrisă de obiectele de tip *Pickup*.

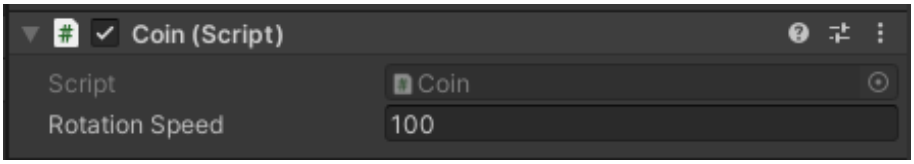
Vom face ca metoda *Collect* să fie virtuală, și vom elimina codul pentru rotație, specific bănuțului.

```
using UnityEngine;

public class Pickup : MonoBehaviour
{
    public virtual void Collect() { }

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

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

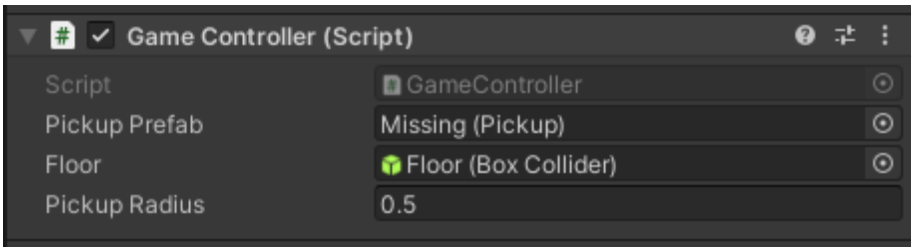
Pentru a defini comportamentul bănuțului, vom defini o nouă clasă numită *Coin*, care moștenește clasa *Pickup* și care definește comportamentele eliminate anterior din clasa *Pickup*.

```
using UnityEngine;

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

    public override void Collect() =>
        GameController.Instance.Score++;

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

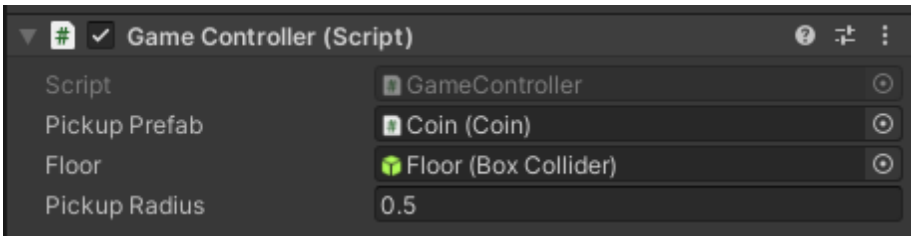
În *prefab*-ul coin vom elimina componenta *Pickup* și vom adăuga componenta *Coin*.



Se observă că referința *Pickup Prefab* a componentei *GameController* a obiectului *GameController* este pierdută. Asta pentru că în momentul în care am eliminat componenta *Pickup* a *prefab*-ului, referința nu a mai fost validă, deci *Unity* e eliminat-o.

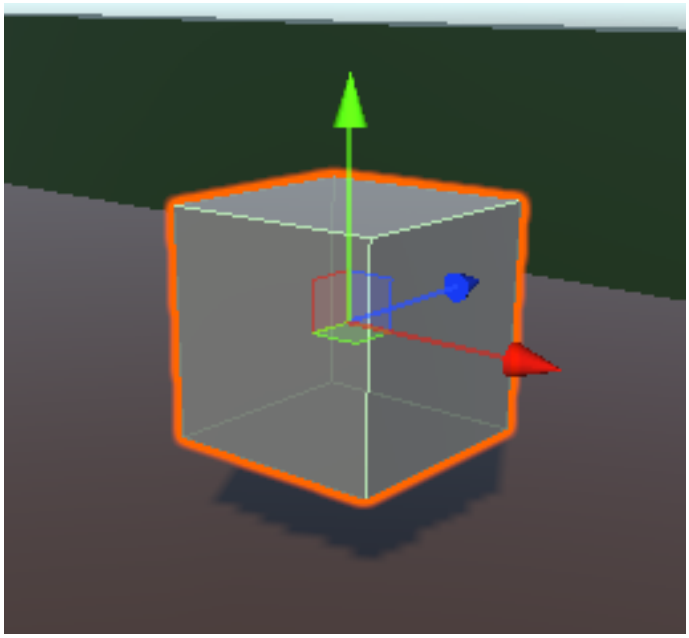


O vom adăuga din nou cu drag and drop, ca în laboratorul anterior.

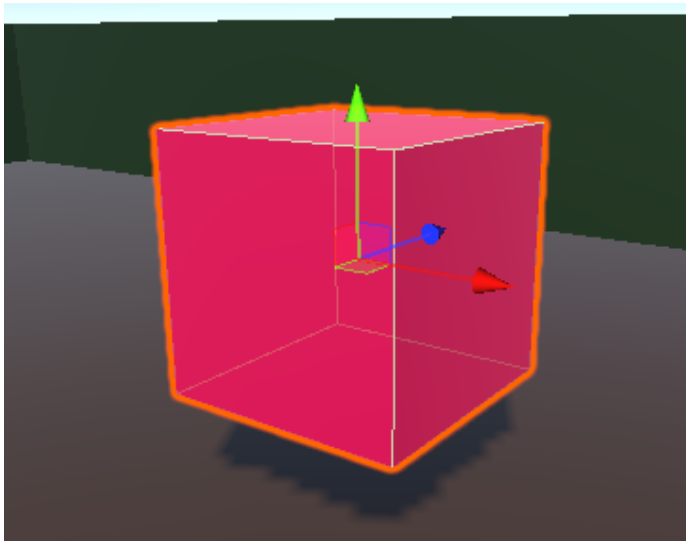


1.4 *Pickup* inversarea controalelor

Vom definiun nou *Pickup* care inversează controalele jucătorului atunci când este colectat. Vom începe prin a adăuga un cub în scenă, căruia îi vom seta poziția $(0, 1, 5)$ pentru a-l putea manipula cu ușurință. Vom numi acest cub *InversePickup*.



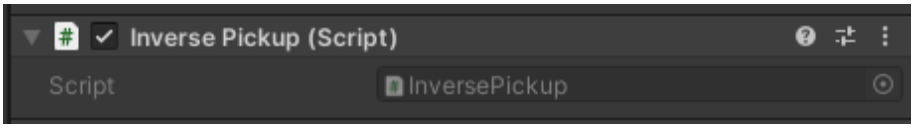
Îi vom atribui un material care să sugereze faptul că acest *Pickup* nu este unul bun.



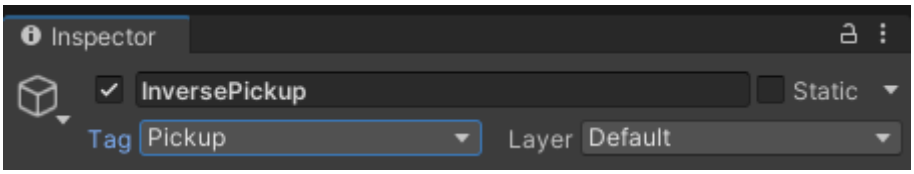
Pentru acest cub vom defini un nou script, numit *InversePickup*, care moștenește clasa *Pickup* definită anterior.

```
public class InversePickup : Pickup
{
}
```

Vom atașa această componentă asupra obiectului *InversePickup* din scenă.



De asemenea, îi vom seta *tag*-ul *Pickup* definit în laboratorul anterior.



Dacă rulăm aplicația, vom observa că obiectul se comportă ca un *Pickup* normal. Atunci când este atins, acesta dispare, iar în locul lui apare un *Coin*. Motivul pentru care bănuțul apare după ce acest obiect este colectat, ci nu când pornește aplicația este următorul: Când pornește jocul, se apelează metoda *OnEnable* a componentei *Pickup* a cubului. În această metodă, se transmite o referință a obiectului către lista *InstantiatedPickups* din *GameController*. Jocul adaugă noi *Pickup*-uri doar atunci când această listă este goală (nu există *Pickup*-uri în scenă), iar în acest caz, lista nu este goală la primul cadru, prin urmare, nu se va adăgua un bănuț în scenă, decât atunci când *InversePickup* va fi eliminat din scenă.

Vom dori ca acest *Pickup* să se rotească constant, și acesta să aibă câte o viteza de rotație pe fiecare axă.

```
[SerializeField]
private Vector3 _rotationSpeed = new(20.0f, 30.0f, 50.0f);

private void Update() =>
    transform.Rotate(_rotationSpeed * Time.deltaTime);
```

În clasa *GameController* vom defini o proprietate booleană pentru inversarea controalelor.

```
public bool InvertControls { get; set; }
```

În clasa *MovingSphere* vom folosi această variabilă pentru a inversa controalele. Pentru a inversa controalele trebuie doar să inversăm vectorul *_desiredVelocity*.

```
private void Update()
{
    var sign = GameController.Instance.InvertControls ? -1 : 1;
    _desiredVelocity = new Vector3(_movement.x, 0.0f, _movement.y) *
                       _maxSpeed * sign;
}
```

În scriptul *InversePickup* vom defini funcția *Collect* pentru a seta valoarea *InvertControls* atunci când *Pickup*-ul este colectat.

```
public override void Collect() =>
    GameController.Instance.InvertControls = true;
```

Acum, atunci când este colectat *Pickup*-ul de inversare a controalelor, controalele sunt inversate permanent. Am dori ca acestea să revină la normal după un anumit număr de secunde. Vom declara o variabilă pentru asta în interiorul clasei *GameController*.

```
[SerializeField, Range(0.5f, 10.0f)]
private float _invertDuration = 5.0f;
```

1.4.1 Corutine

O soluție de a reseta proprietatea *InvertControls* ar fi să numărăm secundele scurse de când variabila a devenit *true*, iar când acest număr depășește valoarea lui *_invertDuration* să resetăm valoarea la *false*. Totuși, această soluție necesită destul de mult cod, și ar complica destul de mult structura proiectului.

Altă soluție este folosirea corutinelor. O corutină este o funcție care returnează un obiect de tip *IEnumerator* și a cărei executare se poate întinde pe mai multe frame-uri.

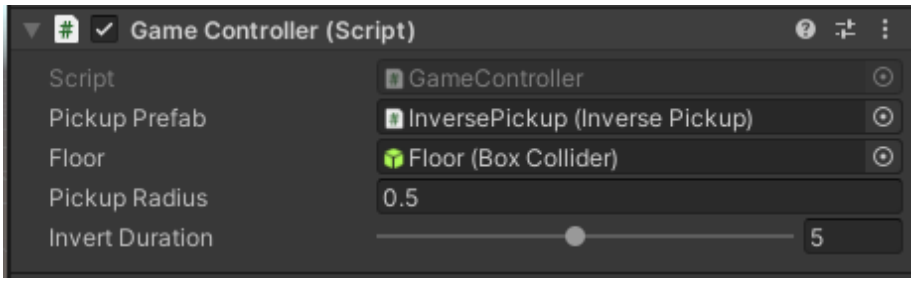
În interiorul clasei *GameController*, vom declara o corutină care așteaptă *_invertDuration* secunde, iar apoi resetează valoarea lui *InvertControls*.

```
using System.Collections;
...
    private IEnumerator ResetInvertControls()
    {
        yield return new WaitForSeconds(_invertDuration);
        InvertControls = false;
    }
```

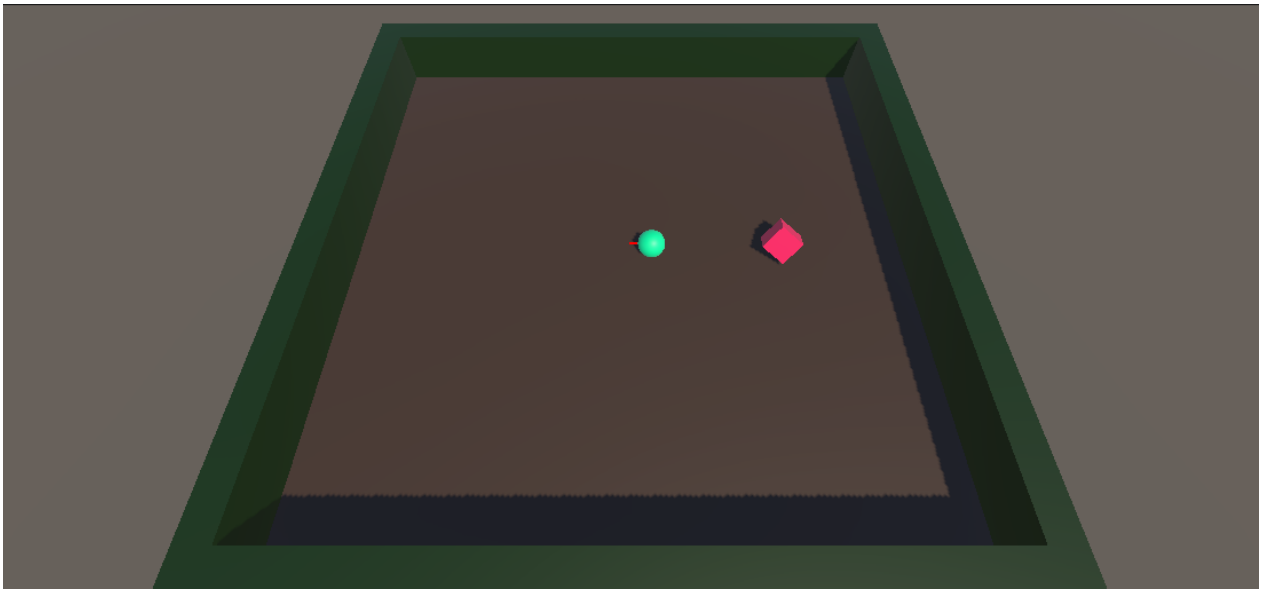
Pentru a porni o corutină se folosește metoda *StartCoroutine*. Vom porni această corutină în setter-ul proprietății *InvertControls*.

```
public bool InvertControls
{
    get => _invertControls;
    set
    {
        _invertControls = value;
        StartCoroutine(ResetInvertControls());
    }
}
...
private bool _invertControls;
...
private IEnumerator ResetInvertControls()
{
    yield return new WaitForSeconds(_invertDuration);
    _invertControls = false;
}
```

Vom reseta poziția acestui cub, după care vom crea un *prefab* pentru *InversePickup*. Vom elimina obiectul din scenă, iar *prefab*-ul îl vom atribui field-ului *Pickup Prefab* al componentei *GameController*.



Acum, în scenă vor apărea doar *Pickup*-uri de tip *InversePickup*.



Se poate observa, că dacă un *InversePickup* este colectat cât timp jucătorul încă are controalele inversate, controalele vor reveni la normal după mai puțin de 5 secunde (valoarea specificată de către noi). Asta se datorează faptului că deși un *InversePickup* noi a fost colectat, corutina celui anterior încă se execută. Putem lua o referință către corutina ultimului *InversePickup* și să o oprim atunci când alt *InversePickup* este colectat. Pentru a opri o corutină se poate folosi metoda *StopCoroutine*.

```
public bool InvertControls
{
    get => _invertControls;
    set
    {
        _invertControls = value;

        if (_invertControlsCoroutine != null)
            StopCoroutine(_invertControlsCoroutine);

        _invertControlsCoroutine = StartCoroutine(ResetInvertControls());
    }
}
...
private Coroutine _invertControlsCoroutine;
...
private IEnumerator ResetInvertControls()
{
```

```
    yield return new WaitForSeconds(_invertDuration);
    _invertControlsCoroutine = null;
    _invertControls = false;
}
```

1.5 Plasare

Momentan, scriptul *GameController* poate plasa în scenă un singur tip de *Pickup*-uri. Vom dori ca obiectul *GameController* să poată plasa mai multe tipuri de *Pickup*-uri, iar fiecare tip de *Pickup* să aibă o anumită șansă.

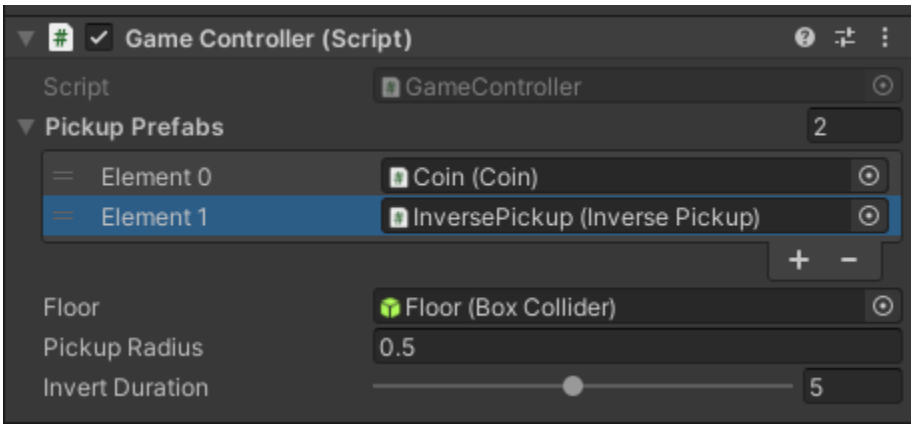
Vom înlocui referința *prefab*-ului din *GameController* cu un *array* de referințe pentru prefab-urile posibile pentru instanțiere.

```
[SerializeField]
private Pickup[] _pickupPrefabs;
```

Când trebuie să instanțiem un nou obiect în scenă, vom alege un element la întâmplare din această listă.

```
private void Update()
{
    if (InstantiatedPickups.Count < 1)
        SpawnPickup(_pickupPrefabs[Random.Range(0, _pickupPrefabs.Length)]);
}
```

În inspector vom atribui *prefab*-urile create pentru *Pickup*-uri elementelor *array*-ului declarat anterior.



Dacă pornim jocul vom observa că acum sunt plasate în scenă ambele tipuri de *Pickup*-uri. *Prefab*-urile au șanse egale să fie plasate, deoarece nu am setat niciun fel de probabilitate de a fi alese.

1.6 Atribuirea probabilităților

Pentru fiecare *Pickup prefab* vom atribui o probabilitate de a fi ales pentru a fi instanțiat. Pentru asta vom crea o structură numită *PickupProperties* care va conține o referință către *prefab* și o probabilitate.

```
public struct PickupProperties
{
    public Pickup Prefab;
    public float Chance;
}
...
[SerializeField]
//private Pickup[] _pickupPrefabs;
private PickupProperties[] _pickupsProperties;
```

Având probabilități atribuite fiecărui *Pickup*, putem folosi meoda selecției prin ruletă (s-a discutat despre aceasta la partea de algoritmi genetici a cursului de Algoritmi Avansați) pentru a alege un *prefab* care urmează să fie instanțiat.

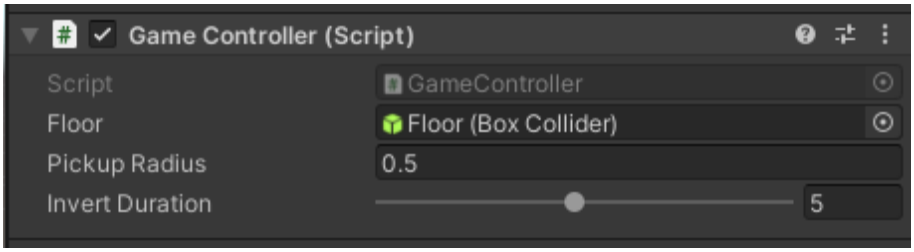
```
using System.Linq;
...
    private Pickup RouletteWheelSelection()
    {
        var sum = _pickupsProperties.Select(x => x.Chance).Sum();
        var point = Random.value * sum;
        var accumulator = 0.0f;

        foreach (var pickupProperties in _pickupsProperties)
        {
            accumulator += pickupProperties.Chance;
            if (accumulator >= point)
                return pickupProperties.Prefab;
        }

        return _pickupsProperties.First().Prefab;
    }
```

Am folosit *Linq* pentru a calcula suma tuturor probabilităților. *Linq* face mai ușoară scrierea de cod care manipulează colecții.

Vom folosi această metodă pentru a alege *prefab*-ul care urmează să fie instanțiat.

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

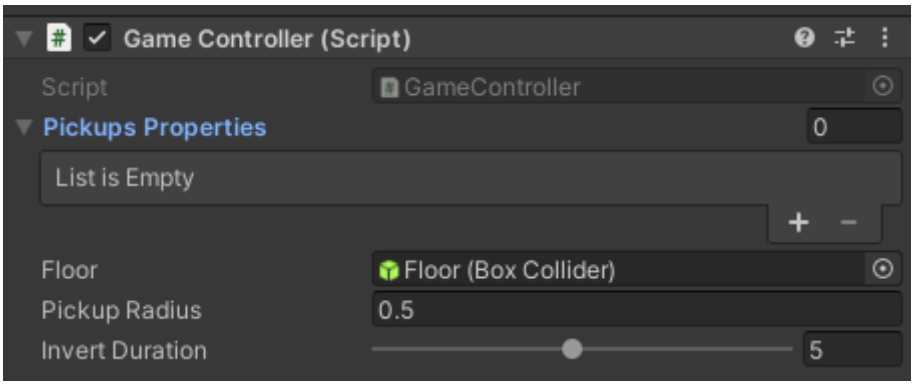
Dacă intrăm în *Unity*, vom observa că în inspector nu mai apare lista de *prefab*-uri și că aceasta nici nu a fost înlocuită cu o listă cu elemente de tip *PrefabProperties*.



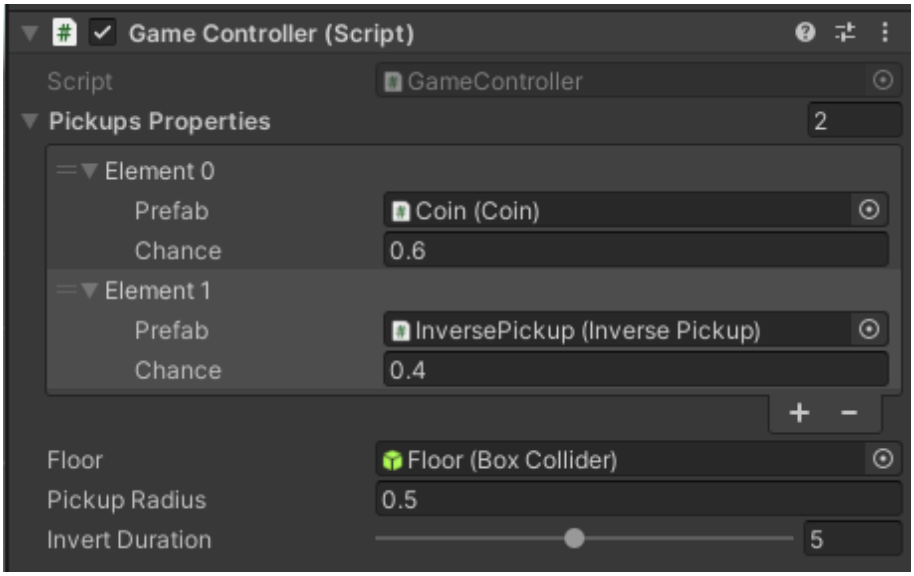
Asta se întâmplă deoarece structura *PrefabProperties* nu este vizibilă pentru *Unity*. Pentru a o face vizibilă, aceasta trebuie marcată ca fiind [*System.Serializable*].

```
using System;
...
    [Serializable]
    public struct PickupProperties
    {
        ...
    }
```

Acum lista va fi vizibilă in inspector.



Vom adăuga în aceasta *prefab*-urile create anterior. *Prefab*-ului *Coin* îi vom atribui probabilitatea 0.6 (60%), iar *prefab*-ului *InversePickup* îi vom atribui probabilitatea 0.4 (40%).

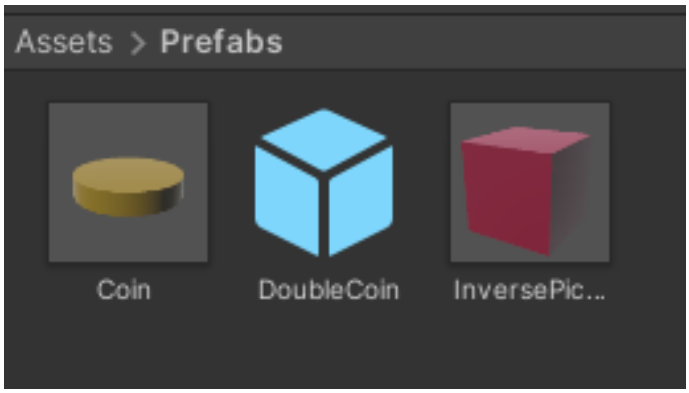


Dacă rulăm, vom observa că monedele au o șansă mai mare de apariție decât *Pickup*-urile care inversează controalele.

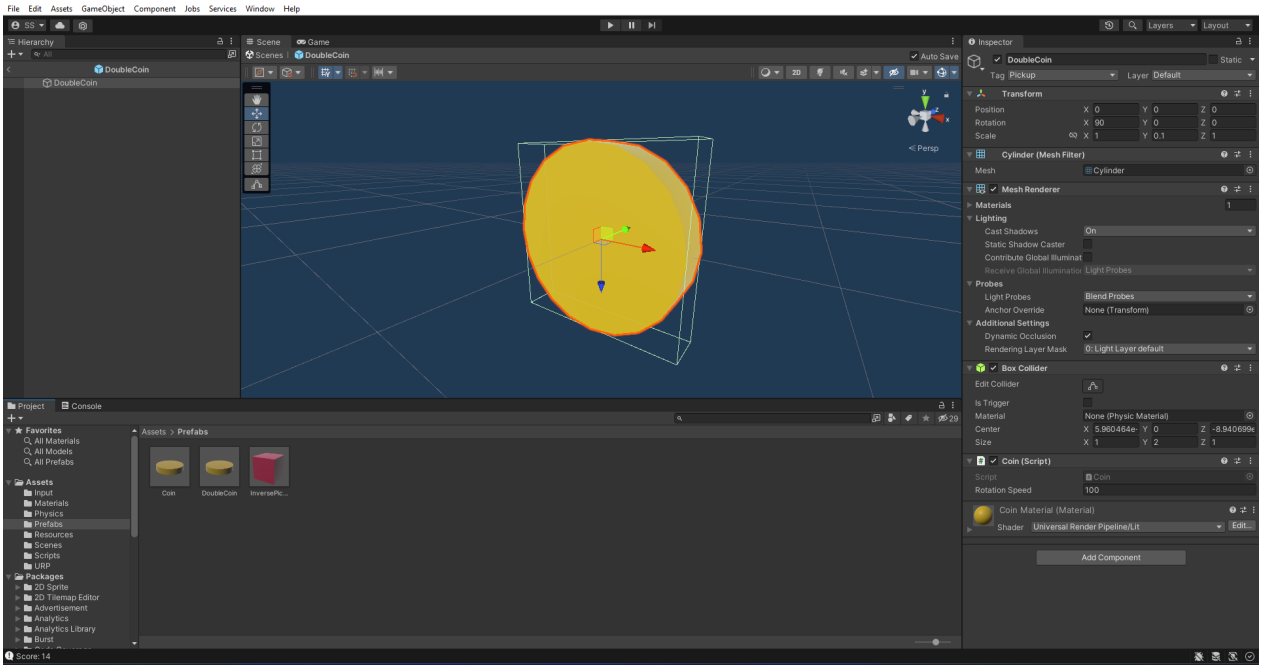
1.7 Bănuți speciali

Ultimul *Pickup* pe care îl vom adăuga este un tip de bănuț care incrementează scorul cu 2 atunci când acesta este colectat.

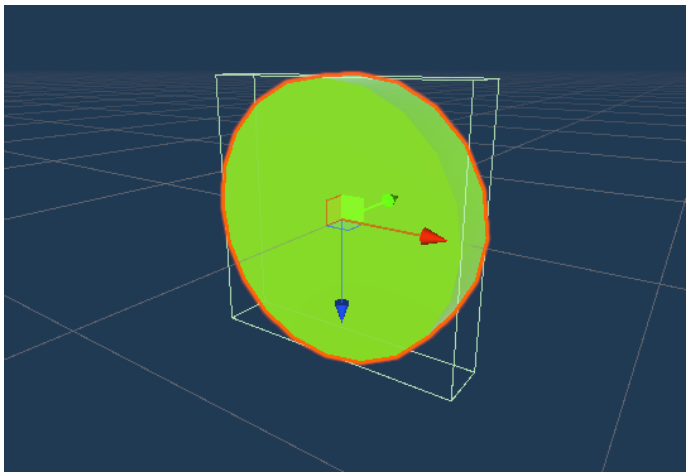
Folosind $Ctrl + D$ vom duplica prefab-ul *Coin*, și vom numi copia *DoubleCoin*.



Pentru a edita acest *prefab* putem folosi *dublu click*, iar *Unity* va deschide o fereastră în care putem edita *prefab*-ul de parcă ar fi un obiect al unei scene.



Vom crea un nou material pentru acest *prefab* și îl vom atribui *prefab*-ului cu drag and drop în această fereastră.

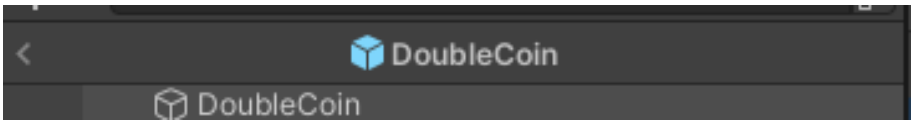


Putem modifica *script*-ul *Coin.cs* pentru a incrementa scorul cu un anumit număr de unități. Pentru asta vom adăuga o variabilă nouă, care are valoarea implicită egală cu 1.

```
[SerializeField]
private int     _increment      = 1;

public override void Collect() =>
    GameController.Instance.Score += _increment;
```

Pentru *prefab*-ul *DoubleCoin* vom seta ca valoarea *Increment* să fie egală cu 2. Pentru *prefab*-ul *Coin* vom păstra valoarea implicită 1.

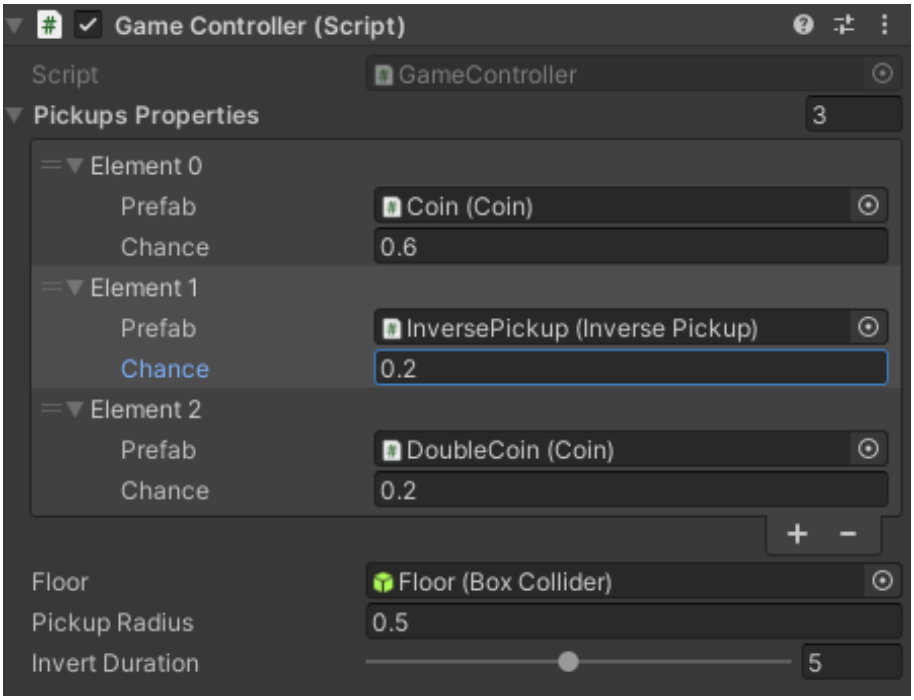
După ce am terminat de configurat noul *Pickup*, ne putem întoarce în scena jocului folosind săgeata din stânga sus a ferestrei *Hierarchy*.



Vom adăuga o referință către acest prefab în componenta *GameController* a obiectului *GameController*. Vom modifica șansele Pickup-urilor astfel:

- *Coin* - 0.6
- *InversePickup* - 0.2
- *DoubleCoin* - 0.2

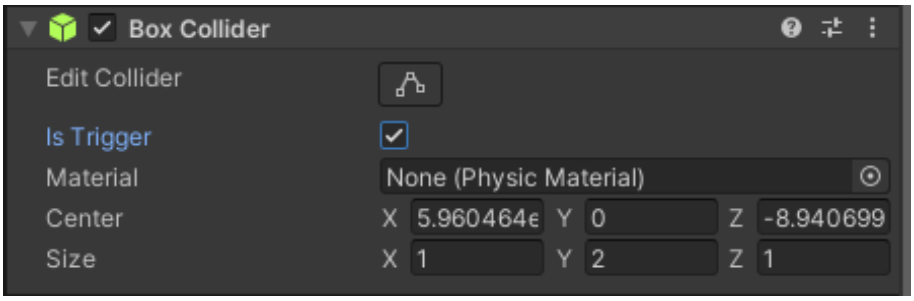
Observație: suma probabilităților nu trebuie să fie 1, poate fi oricât și algoritmul de selecție prin ruletă va funcționa în continuare. Am ales ca suma acestor șanse să fie egală cu 1 pentru a avea o intuiție mai ușoară asupra probabilităților reale.



1.8 Triggere

Până acum, bila a încetinit de fiecare dată când a colectat un *Pickup*. Asta se datorează coliziunii între obiecte și bilă, care face ca viteza bilei să fie afectată. Pentru ca această coliziune să nu aibă un impact asupra bilei, trebuie ca *Pickup*-urile să aibă *collider* de tip *Tigger*.

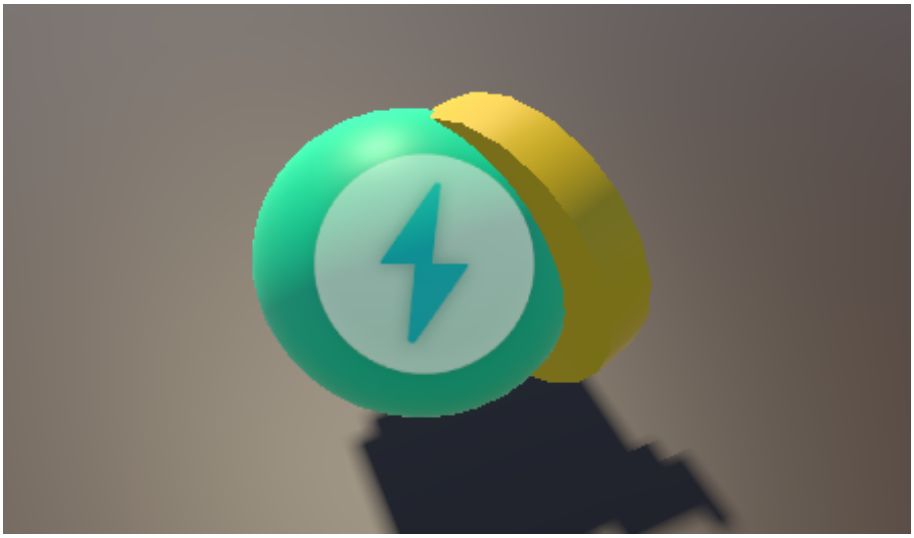
Pentru a face ca un obiect/ *prefab* să aibă un collider de tip *Trigger* trebuie doar ca proprietatea *Is Trigger* a acestuia să aibă valoarea *true*.



Pentru a nu modifica fiecare *prefab* al *Pickup*-urilor, vom seta această proprietate în scriptul *Pickup.cs* în metoda *Awake*.

```
[RequireComponent(typeof(Collider))]
public class Pickup : MonoBehaviour
{
    ...
    private void Awake() =>
        GetComponent<Collider>().isTrigger = true;
    ...
}
```

Observăm că acum, când *Pickup*-urile au *collidere* de tip *Trigger*, jucătorul poate trece prin acestea.



Se mai observă și faptul că acum *Pickup*-urile nu mai sunt colecate la atingere. Acest lucru se întâmplă deoarece metoda *OnCollisionEnter* nu este apelată la coliziunea cu *triggere*. Pentru coliziunea cu obiecte de tip *Trigger* există alte metode. Vom folosi metoda *OnTriggerEnter* pentru a detecta această coliziune. Această metodă primește ca argument un obiect de tip *Collider*, care reprezintă obiectul cu care s-a făcut coliziunea.

```
private void OnTriggerEnter(Collider collider)
{
    var other = collider.gameObject;

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

```
    }
}
```

De regulă, se folosesc *Triggere* pentru situații asemănătoare cu cea prezentată anterior, sau în situații în care dorim să detectăm dacă un anumit obiect a ajuns într-o regiune a hărții jocului. În acest caz, se folosesc *collidere* de tip *Trigger* asupra unor obiecte invizibile, cu scopul doar de a detecta obiectele care ajung în acea zonă.

2 Exerciții

2.1 Adăugați un nou *Pickup* care să facă jucătorul să se deplaseze mai repede pentru un anumit număr de secunde. Puteți folosi corutine cum am folosit în acest laborator, sau puteți implementa manual mecanismul folosind un timer implementat manual.