# Laboratorul 2

## 1 Transformări

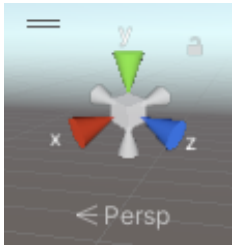
### 1.1 Sisteme de coordonate

Vom crea un proiect nou, cum am creat și în laboratorul precedent, folosind URP.

În scena *SampleScene* vom adăuga un nou Object, pe care îl vom numi *Point*. Reamintim din laboratorul trecut, acesta are o componentă numită $Transform$ care controlează poziția, mărimea și rotația obiectului. Vom reseta componenta $Transform$, atffel ca toate field-urile pentru poziție și rotație să aibă valoarea 0, iar cele pentru scalare să aibă valoarea 1 (*click pe cele* 3 *puncte din dreapta sus a componentei/ Reset*).

În scene, spațiul de coordonate este unul 3D, în care fiecare poziție este reprezentată de un 3-tuplu $(Z, Y, Z)$. Modificând poziția din componenta $Transform$, se modifică și poziția obiectului în spațiu.

În partea din dreapta-sus a ferestrei *Scene* se află un cub din care ies 3 săgeți (roșu, verde și albastru). Săgeata roșie indică direcția axei $X$ a sistemului de coordonate, cea verde indică direcția axei $Y$, iar cea albastră indică direcția axei $Z$.




Modificând valorile $(X, Y, Z)$ din field-ul *Position* al componentei $Transform$, obiectul este deplasat de-alungul axelor corespunzătoare. De asemenea, când obiectul este mutat în scenă folosind săgețile care apar deasupra acestuia când este selectat, se modifică valorile $(X, Y, Z)$ din field-ul *Position* al componentei $Transform$.

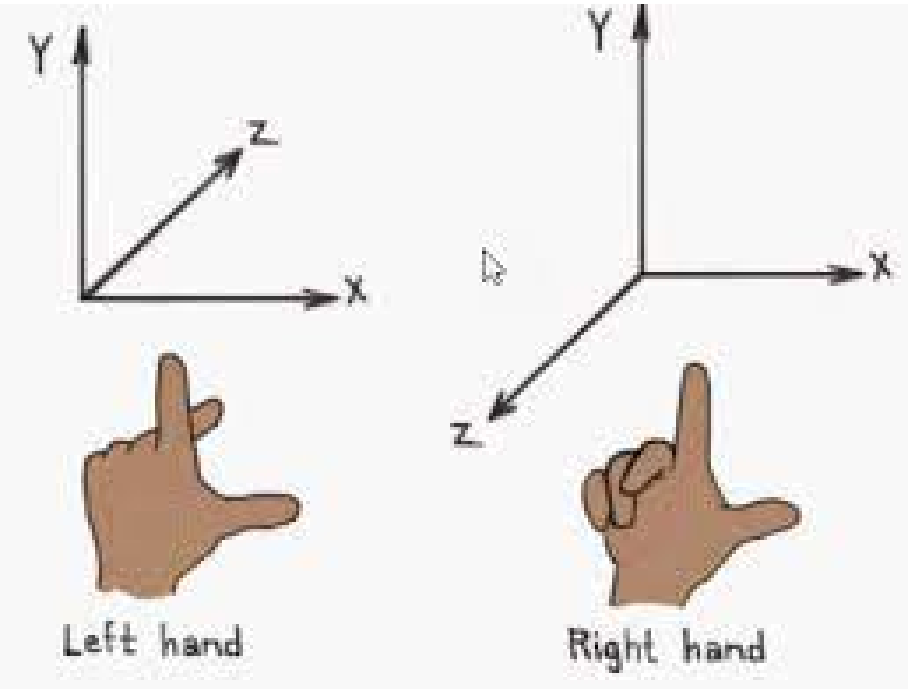
Vom șterge obiectul creat.

### 1.2 Sisteme de coordonate *Left-handed* și *Right-handed*

În spațiul 3D există două tipuri de sisteme de coordonate: *Left-handed* și *Right-handed*. Se numesc *Left* și *Right-handed*, deoarece, axele lor pot fi vizualizate folosind mâna stângă, respectiv mâna dreaptă astfel:

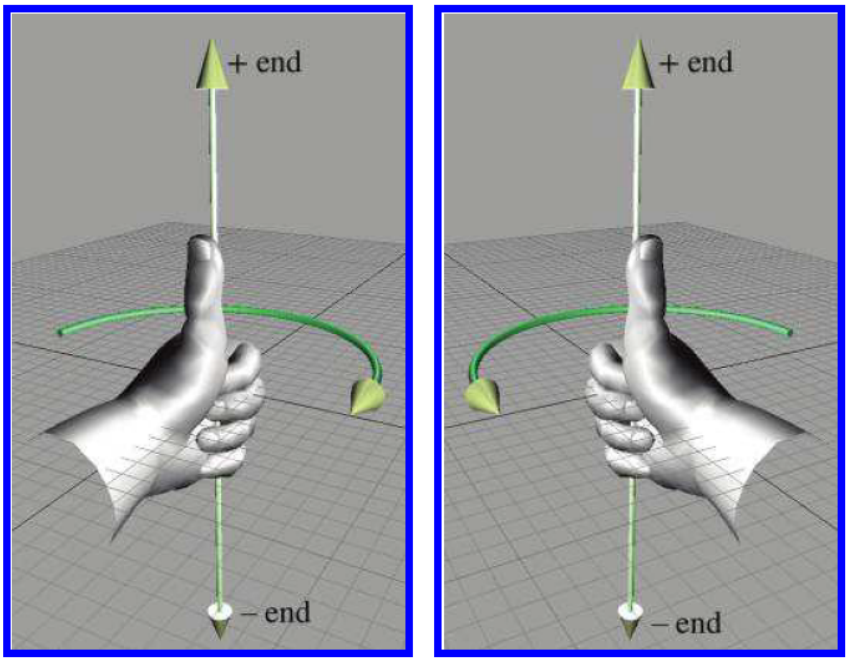
- Se îndreaptă degetul mare spre dreapta, reprezentând axa $X$.
- Se îndreaptă degetul arătător în sus, reprezentând aza $Y$.
- În cazul mâinii stângi, degetul mijlociu se va îndrepta spre exterior, și va reprezenta aza $Z$, care va avea valori pozitive în spatele sistemului de coordonate. În cazul mâinii drepte, degetul mijlociu se

va îndrepta spre interior, și va reprezenta aza $Z$, care va avea valori pozitive în fața sistemului de coordonate.




Un sistem de coordonate definește și o direcție de rotație de-alungul axelor (trigonometrică sau în direcția acelor de ceas). Pentru sistemele de coordonate de tip $Left$-*handed*, rotația în jurul unei axe este în sensul acelor de ceas, iar în cazul sistemelor de coordonate de tip *Right-Handed*, rotația în jurul unei axe este în sens trigonometric.

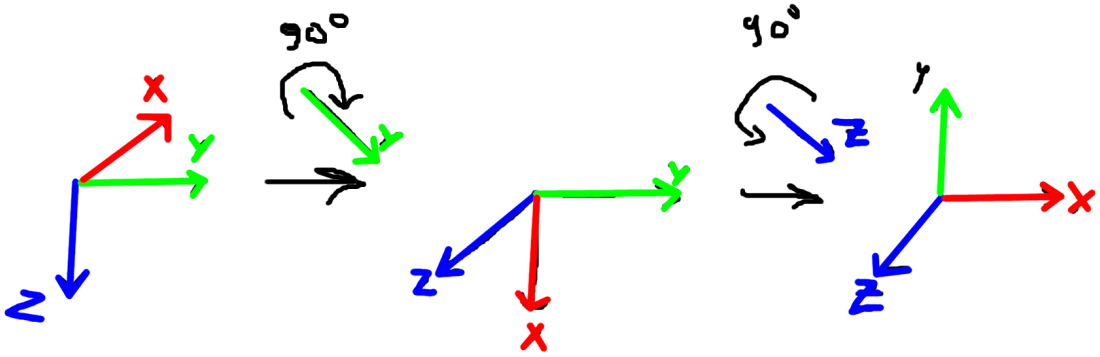
Pentru a determina direcția de rotație în jurul unei axe, se pot folosi din nou mâinile. Se aliniază degetul mare paralel cu axa de rotație, iar apoi se strâng celelalte degete. Direcția celorlalte degete reprezintă rotația în jurul acelei axe pentru sistemele de coordonate $Left$ sau *Right-handed*.




Left-hand rule Right-hand rule

Dacă un sistem de coordonate 3D poate poate fi adus la sistemul de coordonate *Left-handed* din prima figură folosind doar rotații succesive, atunci el este un sistem de coordonate *Left-handed*. Dacă nu, atunci el poate fi adus la sistemul de coordonate *Right-handed* din prima figură folosind doar rotații succesive, caz în care sistemul de coordonate este un sistem de coordonate *Right-handed*.

În exemplul următor, axa $X$ este în față, axa $Y$ duce către dreapta, iar axa $Z$ duce în jos. Acest sistem de coordonate este un sistem de coordonate *Right-handed*, deoarece sistemul de coordonate poate fi adus la forma celui din prima figură folosind doar rotații repetate în jurul unor axe.

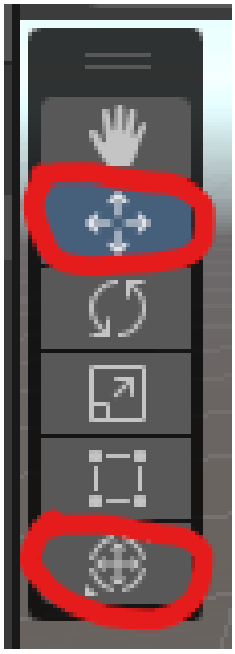


Sistemul de coordonate din Unity este unul $Left\text{-}handed$. În Unity, axa $X$ reprezintă direcția *dreapta* ($Vector3.right$), axa $Y$ reprezintă direcția *sus* ($Vector3.up$), iar aza $Z$ reprezintă direcția *în față* ($Vector3.forward$) (exact ca în sistemul de coordonate $Left\text{-}handed$ din prima figură).

## 1.3 Translație

Translația este operația pe care am văzut-o anterior de a muta obiecte prin intermediul săgeților care apar atunci când obiectele sunt selectate, sau prin modificarea valorilor field-ului *Position* al componentei $Transform$.

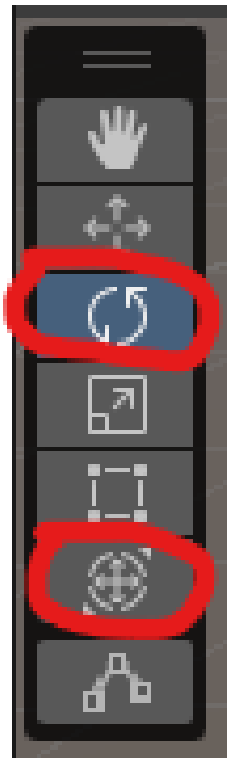
Pentru ca săgețile cu care mutăm obiectele să apară în scenă, trebuie să folosim $Move\ Tool$ sau $Transform\ Tool$. Acestea pot fi selectate din partea din stânga sus a ferestrei *Scene*.



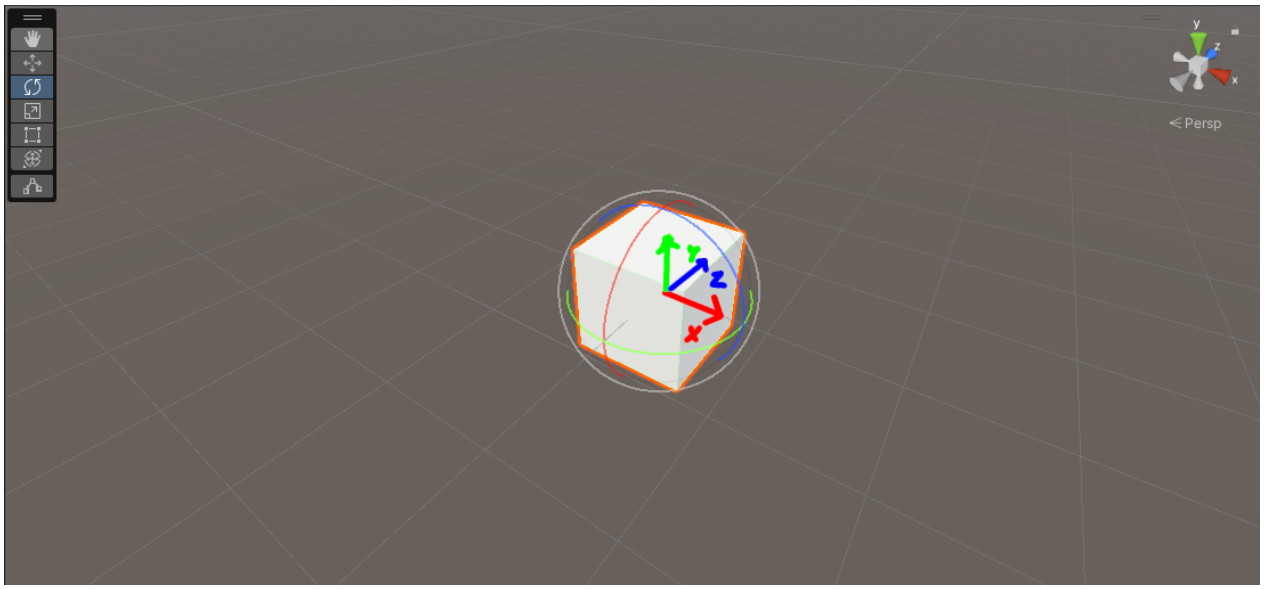
## 1.4 Rotație

Vom adăuga în scenă un cub, căuria nu îi vom da un nume nou ($Create/\ 3D\ Object/\ Cube$). Din nou, îi vom reseta componenta $Transform$, cum am făcut și în cazul punctului anterior.

Folosind $Rotate\ Tool$ sau $Transform\ Tool$ putem roti cubul selectat direct din scenă.



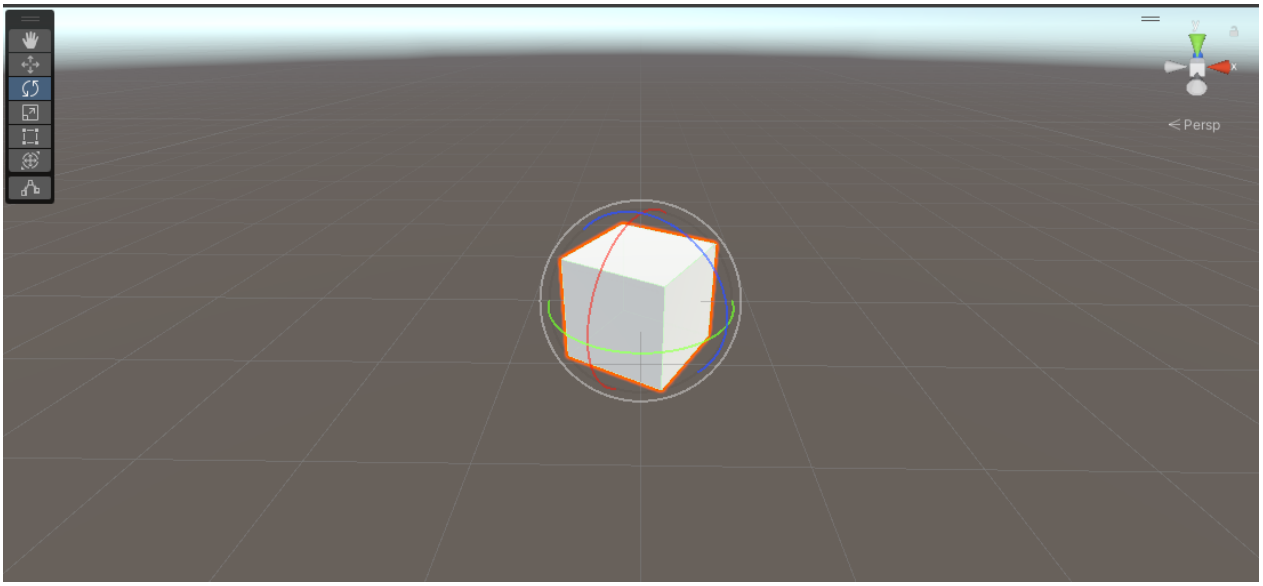
Folosind unul din aceste două unelte, deasupra obiectului vor apărea 3 cercuri care pot fi folosite pentru a roti obiectul în jurul axelor de coordonate $X$, $Y$ și $Z$ (cercul roșu este pentru a roti obiectul în jurul axei $X$, cercul verde este pentru a-l roti în jurul axei $Y$, iar cel albastru este pentru a-l roti în jurul axei $Z$). Se observă că axa de coordonate asociată unui cerc este normala planului în care se află cercul.



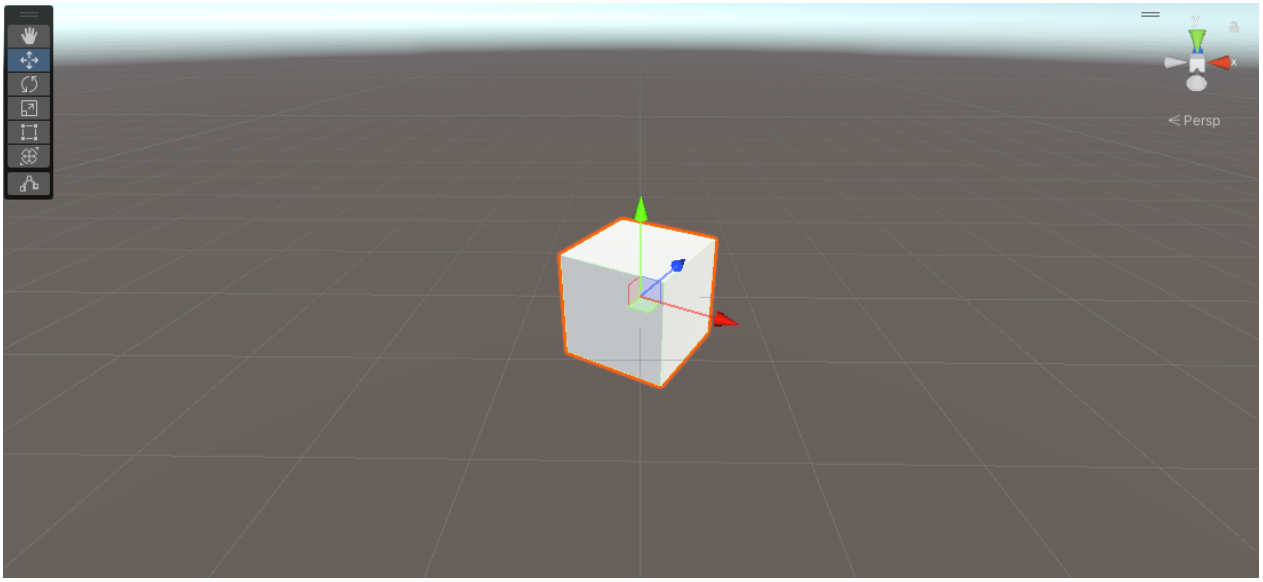

Obiectul poate fi și rotit liber atunci când se folosește *Rotate Tool* (se ține apăsat click pe obiect, iar apoi se mișcă mouse-ul).

Valoarea rotației se poate vedea în componenta $Transform$, în field-ul *Rotation*. Acolo, sunt exprimate în grade unghiurile de rotație în jurul fiecărei axe de coordonate. Rotațiile pot fi atât pozitive cât și negative. Rotațiile sunt pozitive atunci când obiectul este rotit în direcția de rotație specifică sistemului de coordonate ($Left$-*handed*), și negative atunci când obiectul este rotit în sens opus.

În componenta $Transform$ vom seta valoarea rotației să fie egală cu $(0, 30, 0)$. Asta va face ca obiectul să fie rotit 30 de grade în jurul axei $Y$. Privind această rotație de sub obiect o vom vedea ca fiind o rotație în sensul acelor de ceasorinc, iar privind de deasupra obiectului, rotația este una în sens trigonometric.




Dacă acum folosim $Move\ Tool$, vom observa că săgețile atașate obiectului în scenă nu mai corespund cu axele $X, Y \text{și} Z$ ale sistemului de coordonate. Dacă folosim săgeata roșie sau cea albastră pentru a muta obiectul, în componenta $Transform$ vom observa mai multe componente ale poziției sunt modificate, nu doar una singură. De asemenea, în scenă se observă că obiectul se mișcă de-alungul acelor săgeți, nu de-alungul axelor de coordonate din spațiu.

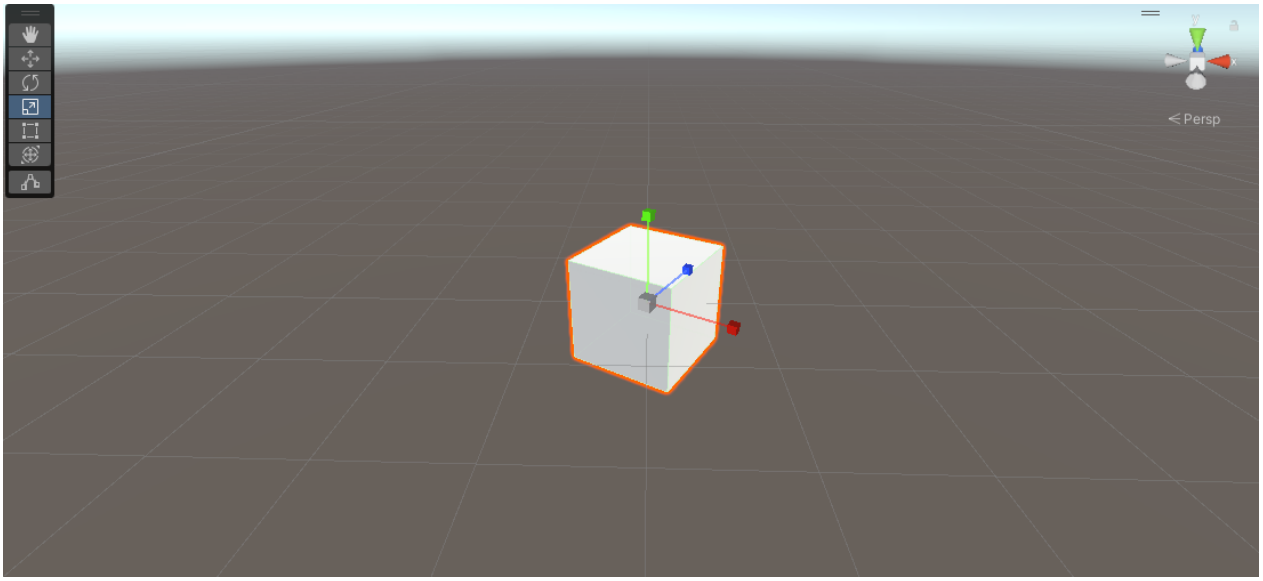



Acest lucru se întâmplă deoarece săgețile atașate obiectului reprezintă vectorii care dau direcția obiectului (dreapta/ sus/ în față). Acești vectori sunt calculați prin aplicarea rotației obiectului asupra vectorilor atașați axelor de coordonate ($X : (1, 0, 0)$, $Y : (0, 1, 0)$, $Z : (0, 0, 1)$). Vom reveni asupra acestor vectori când îi vom folosi în cod.

## 1.5 Scalare

Folosind *Scale Tool* sau *Transform Tool* putem modifca dimensiunile cubului.

În scenă vor apărea niște linii care au în capăt cuburi, care pot fi folosite pentru a mări/ micșora obiectul pe acea direcție.




Vom reseta din nou componenta *Transform* a obiectului pentru a ne fi ușor să înțelegem cum funcționează scalarea.

Modificând valorile $(X, Y, Z)$ din field-ul $Scale$ al componentei $Transform$, obiectul se poate mări sau micșora de-alungul acelor axe.

De exemplu, o scalare cu valoarea 2 pe axa $X$ va face ca obiectul să fie de două ori mai lung. O scalare cu valoarea 0.5 pe axa $Y$ va face ca obiectul să aibă jumătare din înalțimea sa inițială. Atribuiți field-ului $Scale$ al proprietății $Transform$ a cubului valoarea $(2, 0.5, 0)$.

Dacă rotim din nou cubul cu 30 de grade în jurul axei $Y$, observăm că valoarea pentru $Scale$ rămâne neschimbată, deși scalarea este încă relativă la părțile obiectului (dreapta, sus, în față). Acest lucru se întâmplă deoarece intern, Unity prima dată aplică operația de scalare asupra obiectului, apoi cea de rotație, iar la final cea de translație.

# 2 Relația părinte-copil

Vom reseta din nou cubul folosit anterior, și îl vom redenumi $Parent$. Acestui cub îi vom atribui un obiect copil. Pentru a face asta, în fereastra $Hierarchy$ dăm click dreapta pe obiect, iar din meniul care se deschide vom crea un nou cub. Acestui cub îi vom pune numele $Child$.

Alternativ, cubul copil putea fi creat separat în scenă, iar apoi, cu drag and drop, sa fie atașat unui obiect părinte.

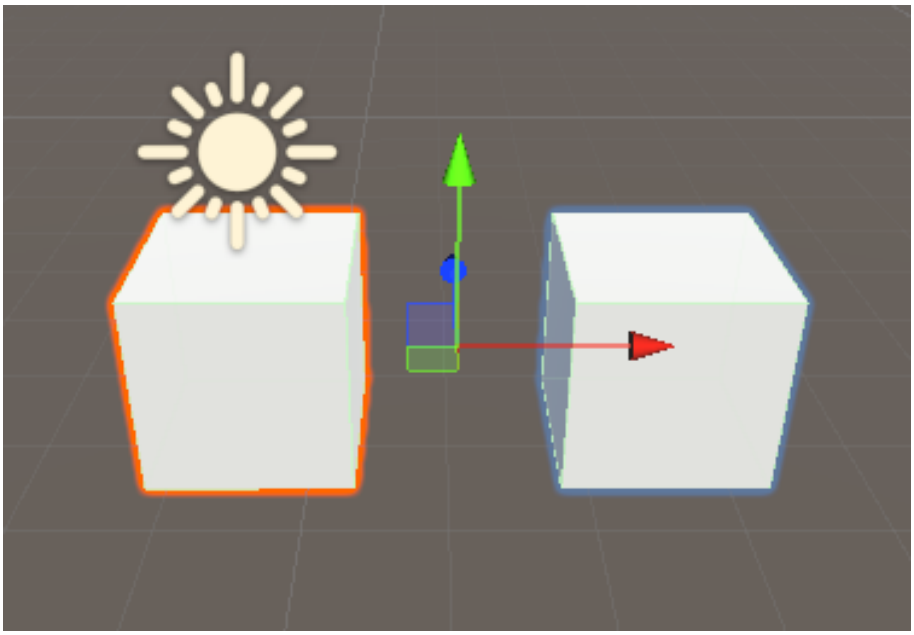
Inițial, copilul este poziționat în același loc cu părintele. Vom seta în componenta $Transform$ a copilului poziția $(2, 0, 0)$.

Dacă proprietățile de transformare ale părintelui sunt modificate, atunci și copilul își schimbă poziția/ rotația/ scalarea.

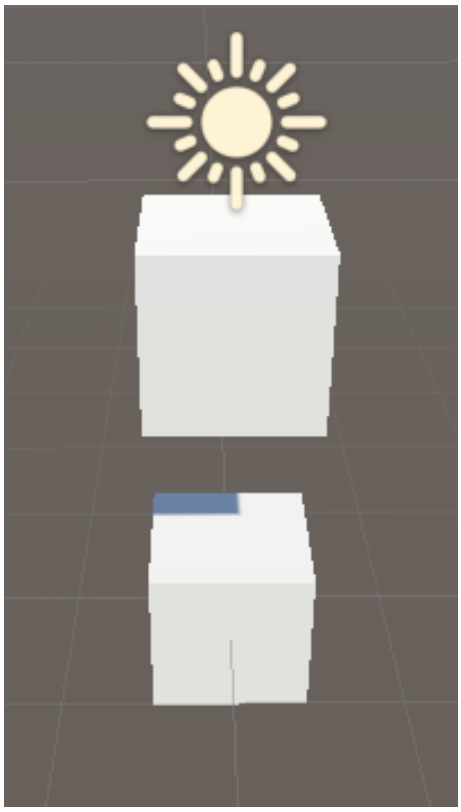
Vom seta succesiv următoarele proprietăți cubului părinte:

- Position: (0, 2, 0)
- Rotation: (0, 0, -90)
- Scale: (2, 0, 0)

După operația de translație, atât obiectul părinte cât și cel copil apar la înălțimea 2 în scenă. Deși nu am modificat componenta transform a copilului, poziția acestuia în spațiu s-a schimbat. Copilul moștenește translațiile aplicate asupra părintelui.

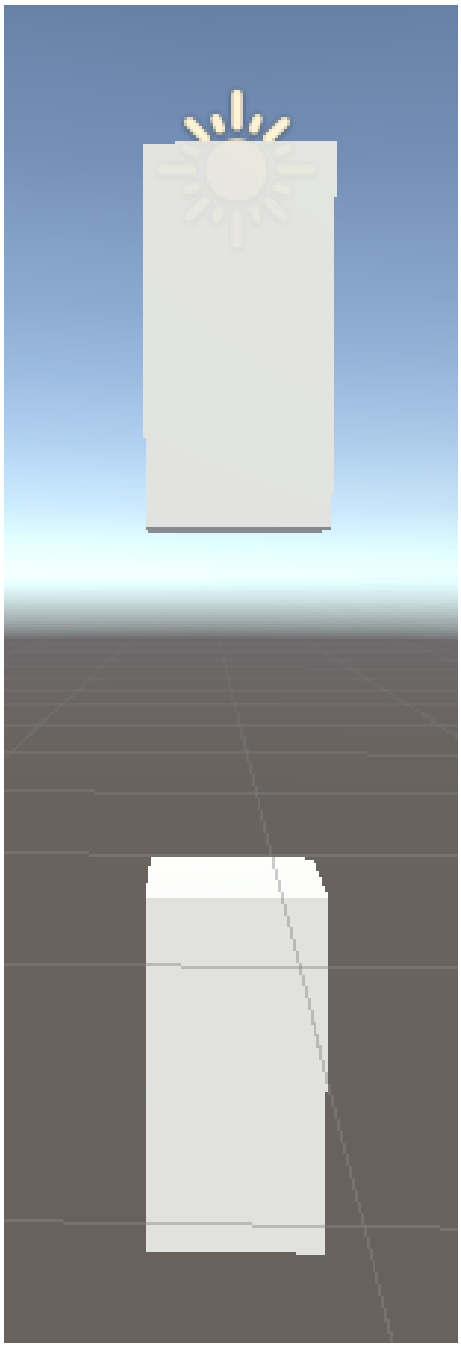


După operația de rotație, cubul părinte va apărea rotit cu -90 de grade de-alungul axei $Z$ (poate nu este evident deoarece un cub rotit cu 90 de grade arata la fel, indiferent de axa de rotație). Cubul copil, este și el rotit cu 90 de grade, dar nu este rotit în jurul centrului său, ci în jurul cubului părinte. Rotația o moștenește de la părinte.



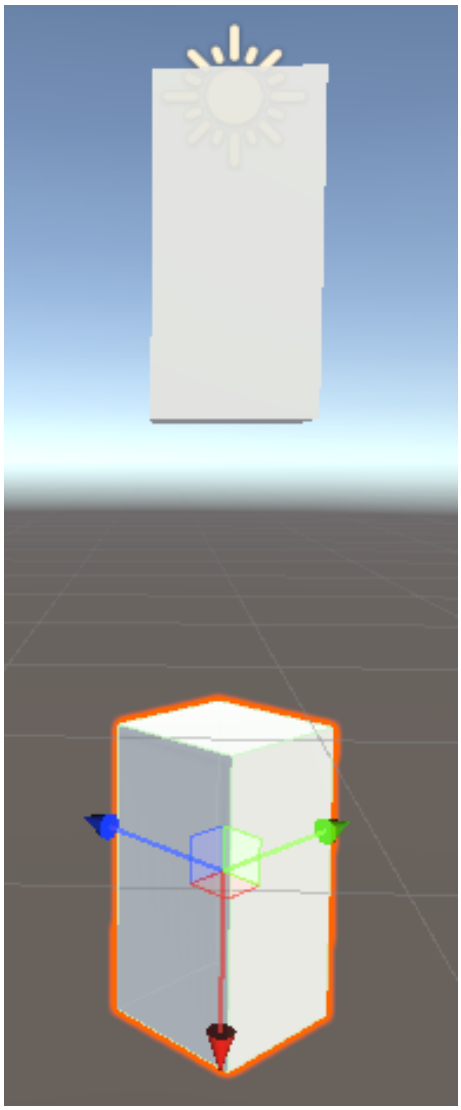
După operația de scalare, ambele cuburi vor fi de două ori mai mari de-alungul axei $X$ (care, datorită rotației anterioare, în spațiul 3D corespunde axei $Y$). Cubul părinte este scalat în același mod în care ar fi fost scalat

și în cazul în care nu ar fi avut copii. Cubul copil, este scalat și devine de două ori mai mare, dar această scalare nu are ca origine centrul copilului, ci centrul părintelui, prin urmare, nu doar că este scalat obiectul copil, dar și spațiul dintre obiectul copil și obiectul părintre este scalat.



Observăm că deși proprietățile copilului s-au schimbat, în inspector, în dreptul componentei sale $Transform$, valorile rămân neschimbate. Asta se întâmplă deoarece acele valori sunt relative la părinte, ci nu la spațiu. Axele $X, Y$ și $Z$ din componeta $Transform$ a copilului reprezintă axele roșii, verzi și albastre ($dreapta$, $sus$, $în\ față$) ale părintelui, nu axele sistemului de coordonate 3D.

Acum vom roti cubul copil cu 45 de grade în jurul axei $X$ (datorită rotațiilor anterioare, pentru copil, această axă are direcția $jos$).



Se observă că atunci când moidificăm proprietățile componentei $Transform$ a copilului, părintele nu este afectat în niciun fel.

Când decide poziția, orientarea și scalarea unui obiect copil în spațiu, Unity evaluează proprietățile din $Transform$-ul lui, iar apoi evaluează proprietățile din $Transform$-ul părintelui. Același lucru, este valabil și pentru o înlănțuire de mai multe obiecte copil/ părinte (caz în care sunt evaluate mai multe $Transform$-uri). Acesta este motivul pentru care $Transform$-urile obiectelor copil rămân neschimbate atunci când cele ale părinților sunt motificate. Mic pseudocod (nu încercați în Unity, codul nu este valid, este doar un proof of concept, intern se folosesc înmulțiri de matrici, dar este prea complex pentru exemplul nostru didactic):

```
WorldTransform EvaluateChild(LocalTransform child)
{
    // Presupunem ca ordinea in care efectuam operayiile este
    // cea in care acestea vor fi aplicate de catre Unity.
    // in realitate, ordinea este inversa, dar acesta
    // este un exemplu didactic.
```

```
    var result = new WorldTransform();
    result.ApplyScale(child.Scale);
    result.ApplyRotation(child.Rotation);
    result.ApplyTranslation(child.Position);

    if (child.Parent is null)
        return result;

    result.ApplyWorldTransform(EvaluateChild(child.Parent));

    return result;
}
```

Vom șterge cubul părinte, iar cubul copil va fi șters automat.

# 3 Scripturi în Unity

## 3.1 Convenții

Scripturile în Unity sunt scrise în *C#*. Pentru a ne fi mai ușor să înțelegem codul, vom folosi următoarele convenții în cadrul laboratorului:

- Numele de variabile locale sunt definite folosind *camelCase*.
- Numele de funcții sunt definite folosind *PascalCase*.
- Variabilele publice sau protejate și proprietățile sunt definite folosind *PascalCase*.
- Variabilele private sunt definite folosind *camelCase*, și sunt prefixate cu _ (underscore).
- Atunci când situația permite, vom folosi *var* în loc de a scrie tipul explicit al variabilei.
- În cazul variabilelor publice, vom prefera să folosim proprietăți în loc (dacă situația permite, deoarece de multe ori este nevoie să avem chiar variabile publice, nu proprietăți).
- Metodele care nu modifică variabilele membre vor fi definite ca *static*.
- În interiorul unei clase, ordinea în care vom defini conținutul este următoarea: structuri de date interne, variabie și proprietăți, metode. Pentru orice element dintr-o clasa, ordinea este următoarea: public, protected, private. Elementele statice vor fi definite după cele statice.
- Atunci când avem un *if*, un *for* sau un *while*, iar în interiorul acestuia există o singură instrucțiune, vom omite acoladele.
- Funcțiile care conțin o singură linie de cod le vom transforma în *body expressions*.
- Vom menționa explicit nivelul de acces al elementelor dintr-o clasă (implicit, ele sunt *private*).

## 3.2 Componente

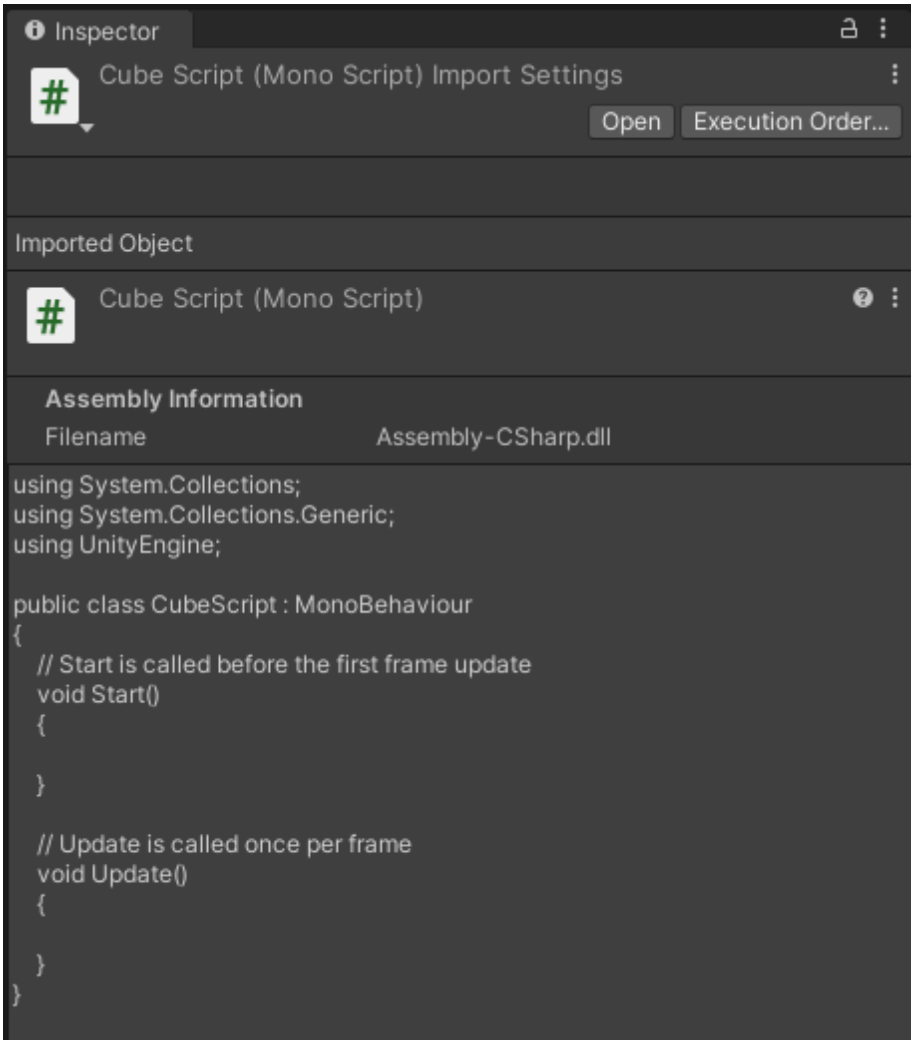
Așa cum am vazut în laboratoarele anterioare, componentele adaugă funcționalități obiectelor asupra cărora sunt atașate. Pentru a defini funcționalități noi, trebuie să ne definim propriile componente. Acest lucru îl facem cu ajutorul scripturilor.

Vom adăuga din nou un cub în scenă, fără să-i schimbăm numele.

În interiorul directorului *Assets* vom crea un nou director numit *Scripts*. În interiorul directorului vom crea un nou script numit *CubeScript*. Pentru a crea un script nou, în director se apasă *click dreapta/*

*Create/ C# Script*.

După ce creăm un script, în inspectorul din partea dreaptă a ecranului vom vedea codul acelui script. Implicit, Unity adaugă codul necesar unei componente.




Pentru a edita acest script, trebuie să apăsăm dublu-click pe el, iar Unity va deschide automat IDE-ul cu care a fost configurat să deschidă *Script*-urile. Vom șterge tot codul din Script și vom începe să-l rescriem manual.

*C#* este un limbaj orientat pe obiecte, deci în majoritatea cazurilor se folosesc clase. Pentru a defini o componentă nouă este nevoie să definim o clasă cu același nume ca și scriptul. Vom defini o clasă publică pentru a nu avea probleme de acces către această clasă.

```
public class CubeScript
{
}
```

Momenta, acest script nu poate fi folosit pe post de componentă. Pentru asta, este nevoie ca această clasă să moștenească clasa *UnityEngine.MonoBehaviour*.

```
public class CubeScript : UnityEngine.MonoBehaviour
{
}
```

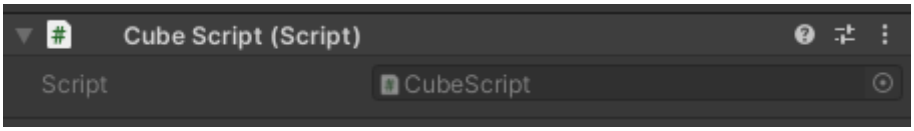
Pentru a nu scrie *UnityEngine.MonoBehaviour*, putem scriem direct *MonoBehaviour*, iar la începutul fișierului să scriem *using UnityEngine*; pentru a avea acces la tot ce conține namespace-ul *UnityEngine* (echivalent cu *using namespace* din C++).

```
using UnityEngine;

public class CubeScript : MonoBehaviour
{
}
```

Acum, scriptul poate fi folosit pe post de componentă.

Revenind la cubul creat anterior, pentru atașa această componentă asupra lui, putem fie să folosim butonul *Add Component* din inspector, fie să îi dăm drag and drop scriptului peste obiectul din fereastra *Hierarchy*.




Momentan, componenta creată nu face nimic. Pentru a îi adăuga funcționalitate, va trebui să definim anumite metode pe care Unity intern să le apeleze atunci când obiectul este activ. Vom defini funcția *Awake*, deoarece este prima funcție care este apelată atunci când o componentă este adăugată. Această metodă este apelată o singură dată. Poate fi văzută ca un fel de constructor.

```
using UnityEngine;

public class CubeScript : MonoBehaviour
{
    private void Awake()
    {
    }
}
```

Am definit funcția ca fiind *private* deoarece Unity nu ține cont de nivelul de acces al metodei, funcția nu este apelată în interiorul mediului .NET.

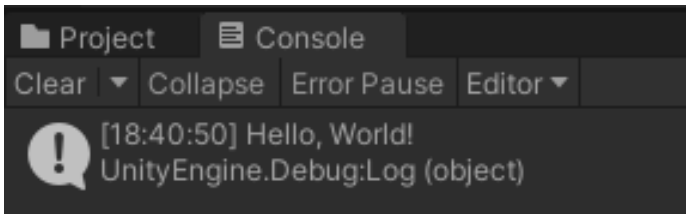
Pentru a afișa mesaje utile în debugging folosim metoda *Log* din clasa *Debug*.

```
using UnityEngine;

public class CubeScript : MonoBehaviour
{
```

```
    private void Awake()
    {
        Debug.Log("Hello, World!");
    }
}
```

Dacă dăm play jocului, în fereastra *Console* din partea de jos a Unity vom vedea mesajul *Hello*, *World*!.




O simplă funcționalitate pe care o putem implementa este să mutăm cubul prin intermediul acestui script. Am văzut anterior că pentru a modifica poziția unui obiect este nevoie să modificăm field-ul *Position* al componentei *Transform*. În interiorul componentelor avem acces la o variabilă numită *transform*, prin care avem acces la componenta *Transform* a obiectului. Pentru a schimba poziția cubului putem face în felul următor:

```
using UnityEngine;

public class CubeScript : MonoBehaviour
{
    private void Awake()
    {
        // Atribuim o valoare nou  field-ului position
        // al componentei Transform.
        transform.position = new Vector3(0.0f, 2.0f, 0.0f);
    }
}
```

Dacă dăm *play*, poziția cubului va fi (0, 2, 0), indiferent de poziția pe care acesta a avut-o înainte de a da *play*.

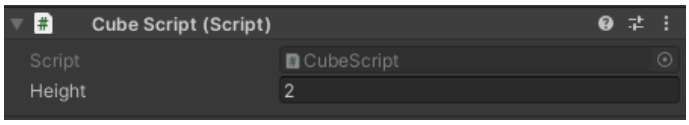
Să presupunem că am avea două cuburi în scenă. Unul am vrea să aibă poziția $(0, 2, 0)$, iar celălalt poziția $(0, 3, 0)$ atunci când pornește jocul. În loc să facem două *script*-uri, o soluție mai bună este să folosim o variabilă care să definească înălțimea la care vrem să se afle cubul.

```
using UnityEngine;

public class CubeScript : MonoBehaviour
{
    public float Height = 2.0f;

    private void Awake()
    {
        transform.position = new Vector3(0.0f, Height, 0.0f);
    }
}
```

Această variabilă are o valoare default $2.0f$. De fiecare dată când scriptul va fi aplicat asupra unui obiect, va avea valoarea $2.0f$ implicit pe acel obiect. Totuși, valoarea poate fi schimbată din editorul Unity, astfel ca obiecte diferite să aibă valori diferite ale acelei variabile. Dacă privim acum inspectorul, putem observa că acum a apărut field-ul *Height* în componenta *CubeScript*.




Dacă modificăm această valoare în inspector înainte de a da *play*, obiectul se va afla la altă înălțime.

Observăm că nimic nu se întâmplă dacă modificăm valoarea field-ului *Height* în timp ce jocul rulează. Asta deoarece metoda *Awake* este apelată o singură data, atunci când componenta devine activă pentru prima dată. Vom repara acest lucru în cele ce urmează.

Pentru *Height* am folosit o variabilă publică, pentru ca aceasta să fie vizibilă în inspector. Totuși, această variabilă nu este nevoie să fie publică deoarece nu o folosim în interiorul altor scripturi. O vom defini ca variabilă privată, iar desupra definiției ei, vom scrie [*SerializeField*] pentru a îi indica lui Unity că variabila trebuie să fie vizibilă în inspector.

```
using UnityEngine;

public class CubeScript : MonoBehaviour
{
    [SerializeField]
    private float _height = 2.0f;

    private void Awake()
    {
        transform.position = new Vector3(0.0f, _height, 0.0f);
    }
}
```

Deoarece codul nostru rulează în interiorul funcției Awake, acesta este apelat o singură dată. Noi am dori ca acesta să fie apelat pentru fiecare cadru al jocului, pentru ca poziția să se modifice instant atunci când noi modificăm valoarea *Height* din inspector. Pentru acest lucru vom folosi metoda *Update*, care este apelată pentru fiecare cadru al aplicației.

```
using UnityEngine;

public class CubeScript : MonoBehaviour
{
    [SerializeField]
    private float _height = 2.0f;

    private void Update()
    {
        transform.position = new Vector3(0.0f, _height, 0.0f);
    }
}
```

Acum, dacă dăm play și modificăm valoarea lui *Height* în timp ce jocul rulează, cubul își va schimba poziția în timp real. De observat este faptul că valoarea pe care o are *Height* în inspector rămâne neafectată atunci când jocul este închis.

## 3.3 Mișcare

### 3.3.1 Mișcare de bază

Din moment ce funcția *Update* este apelată pentru fiecare cadru al aplicației, înseamnă că putem ca la fiecare cadru să mutăm obiectul puțin-câte-puțin într-o anumită direcție, și să simulăm efectul de mișcare. În loc să atribuim o poziție nouă la fiecare cadru, vom incrementa poziția existentă.

```
using UnityEngine;

public class CubeScript : MonoBehaviour
{
    // am eliminat variabila _height
    private void Update()
    {
        // incrementeaza putin pozitia la fiecare frame
        transform.position += new Vector3(0.0f, 0.01f, 0.0f);
    }
}
```

Dacă dăm *play* vom observa cum cubul se duce în sus.

Pentru a nu modifica manual poziția de fiecare dată. ne putem folosi de metoda *Translate* a componentei *transform*. Argumentul ei este un vector 3-dimensional care reprezintă valoarea care va fi adăugată poziției (în cazul de mai sus este vorba de vectorul $(0.0, 0.01, 0.0)$).

```
using UnityEngine;

public class CubeScript : MonoBehaviour
{
    private void Update()
    {
        transform.Translate(0.0f, 0.01f, 0.0f);
    }
}
```

Efectul este același, dar codul este puțin mai curat acum.

### 3.3.2 Mișcare bazată pe timp

Exemplul de mai sus nu este unul bun, deoarece obiectul este mișcat cu aceeași valoare la fiecare frame, indiferent de cât de frecvente sunt frame-urile. Asta va face ca pe un calculator mai performat să se miște mai rapid, iar pe un calculator mai slab să se miște mai lent. Fie următorul exemplu:

Pe un calculator jocul are 60 de cadre pe secundă, iar pe altul are 30 de cadre pe secundă. Pe calculatorul pe care jocul are 60 de cadre pe secunda, pe parcurul unei secunde, cubul se va fi mișcat în sus cu 0.6 unități. Pe celălalt calculator se va fi mișcat doar 0.3 unități.

Pentru a rezolva această problemă este nevoie ca viteza cu care obiectele se mișcă într-un frame să depindă de timpul scurs între frame-uri. În exemplu de mai sus, pentru a parcurge aceeași distanță pe durata unei secunde, ar trebui ca obiectul de pe calculatorul slab să se miște pe o distanță de două ori mai mare.

De regulă, astfel de viteze sunt înmulțite cu timpul scurs între cadre. exprimat în secunde. În Unity există o variabilă specială numită *Time.deltaTime* care conține timpul scurs dintre cadre, măsurat în secunde. Fie un exemplu care folosește timpul dintre cadre:

Avem un obiect care are viteza de 2 unitatăți pe secundă.

Pe un calculator care rulează aplicația la 60 de cadre pe secundă, timpul dintre cadre va fi egal cu $\frac{1}{60}$ secunde. Înseamă că într-un singur cadru, obiectul se va deplasa cu o distanță egală cu $\frac{2}{60}$. Cum jocul rulează la 60 de cadre pe secundă, într-o secundă obiectul se va deplasa de 60 de ori, așsadar se va deplasa $60\frac{2}{60} = 2$ unități.

Pe un calculator care rulează aplicația la 30 de cadre pe secundă, timpul dintre cadre va fi egal cu $\frac{1}{30}$ secunde. Înseamă că într-un singur cadru, obiectul se va deplasa cu o distanță egală cu $\frac{2}{30}$. Cum jocul rulează la 30 de cadre pe secundă, într-o secundă obiectul se va deplasa de 30 de ori, așsadar se va deplasa $30\frac{2}{30} = 2$ unități.

Deci, prin folosirea timpului dintre cadre putem face ca jocul să se miște la fel de rapid, indiferent de hardware-ul pe care este rulat.
Modifcând exemplul anterior, obținem următorul cod:

```
using UnityEngine;

public class CubeScript : MonoBehaviour
{
    [SerializeField]
    private float _speed = 2.0f; // unitati pe secunda

    private void Update()
    {
        transform.Translate(0.0f, _speed * Time.deltaTime, 0.0f);
    }
}
```

# 4 Exerciții

## 4.1 Exercițiul 1

### 4.1.1 a)

Se dă un sistem de coordonate în care axa $X$ duce către dreapta, $Y$ duce în față, iar $Z$ duce în sus. Ce fel de sistem de coordonate este?

### 4.1.2 b)

Definiți o funcție matematică care să transforme coordonatele din acest sistem de coordonate în coordonate ale sistemului de coordonate din Unity.

### 4.1.3 c)

Definiți o funcție matematică care transformă coordonate din sistemul de coordonate folosit de Unity în sistemul de coordonate definit anterior.

## 4.2 Exercițiul 2

Într-un sistem de coordonate *Left-handed*:

### 4.2.1 a)

Când privim o rotație cu valoare pozitivă din capătul pozitiv al axei de rotație, aceasta este în sens trigonometric sau în sensul acelor de ceasornic?

#### 4.2.2 b)

Când privim o rotație cu valoare pozitivă din capătul negativ al axei de rotație, aceasta este în sens trigonometric sau în sensul acelor de ceasornic?

Într-un sistem de coordonate *Right-handed*:

#### 4.2.3 a)

Când privim o rotație cu valoare pozitivă din capătul pozitiv al axei de rotație, aceasta este în sens trigonometric sau în sensul acelor de ceasornic?

#### 4.2.4 b)

Când privim o rotație cu valoare pozitivă din capătul negativ al axei de rotație, aceasta este în sens trigonometric sau în sensul acelor de ceasornic?

### 4.3 Exercițiul 3

Modificați scriptul astfel încât obiectul să se rotească cu o anumită viteză în jurul unei axe (hint: folosiți transform.Rotate).

### 4.4 Exercițiul 4

Modifcați scriptul astfel încât obiectul să își schimbe singur sensul de rotație o dată la un anumit număr de secunde.

### 4.5 Exercițiul 5

Creați un lanț de cuburi copil care conțin componenta definită mai sus. Ce rezultate puteți obține jucându-va cu parametrii scriptului pe diferite obiecte?

### 4.6 Exercițiul 6

Modificați exercițiul din laboratorul anterior astfel ca obiectele din care este alcătuit robotul să aibă o relație de tip părinte copil.