Laboratorul 1
Unity DOTS

1 Configurarea proiectului

DOTS (Data Oriented Tech Stack) reprezintă o mulțime de pachete pe care le putem folosi pentru a scrie cod mai eficient în Unity. Optimizările pe care le putem face folosind aceste pachete sunt de tip Low-Level (Caching, SIMD, multi-threading, etc).

În acest laborator vom folosi Unity 2022. Vom crea un proiect nou folosind template-ul $3D$ (URP) $Core$ 1. Acest template trebuie descărcat înainte de a crea un proiect.

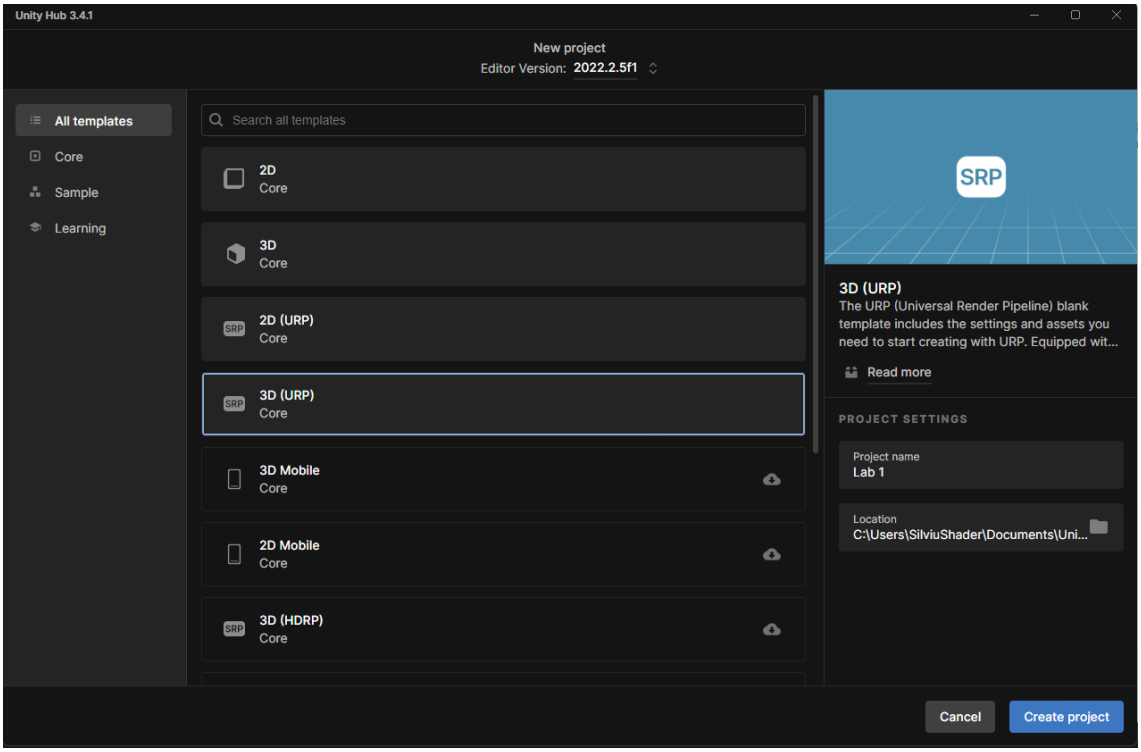

Figure 1: Crearea unui proiect

După ce este creat proiectul, vom elimina fișierele de care nu avem nevoie apăsând click pe butonul *Remove Readme Assets* din Inspector 2.

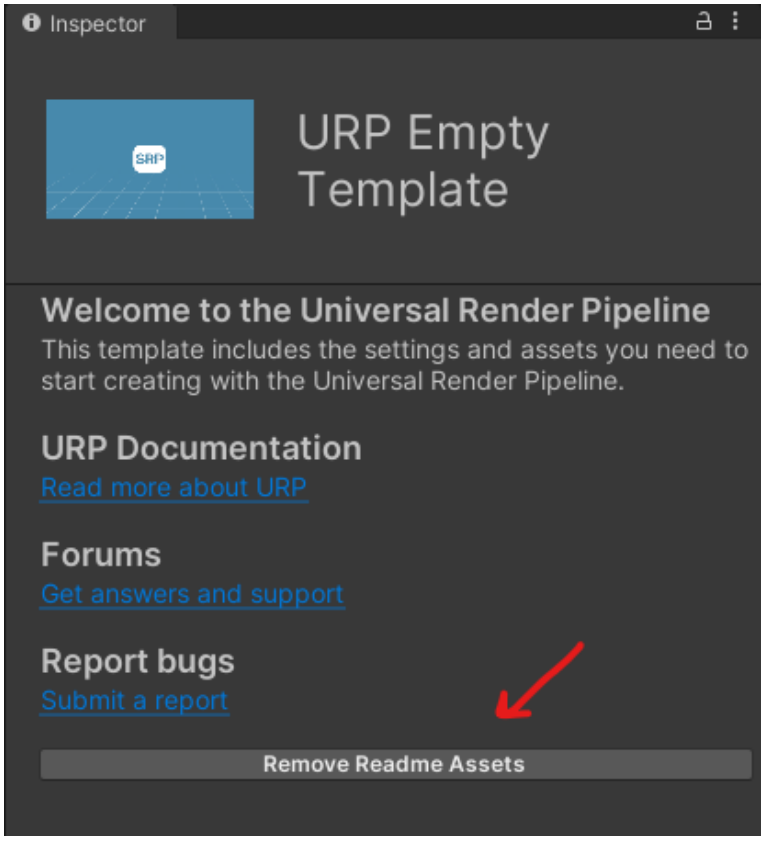

Figure 2: Eliminarea fișierelor extra.

În continuare vom instala un pachet din Unity pentru a ne permite să lucrăm folosind DOTS. Acest pachet depinde de alte pachete care ne vor fi utile în laborator. Acele pachete vor fi preluate automat o dată cu instalarea pachetului de bază. Accesți `Window > Package Manager`, apoi apăsați pe butonul `+` și selectați *Add package by name* 3. Introduceți numele *com.unity.entities.graphics*, lăsați câmpul *Version* gol, iar apoi apăsați pe butonul *Add* și așteptați ca pachetele să se instaleze.

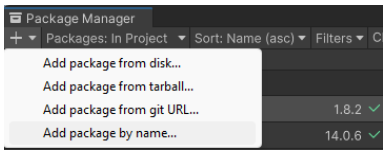

Figure 3: Adăugarea pachetelor după nume.

În `Edit > Project Settings > Editor` vom activa *Enter Play Mode Options* și vom lăsa sub-opțiunile *Reload Domain* și *Reload Scene* dezactivate 4.

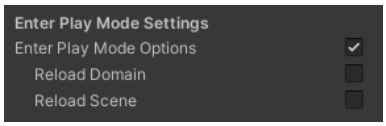

Figure 4: Enter Play Mode Options activat.

Ultimul lucru pe care îl avem de făcut este să ne asigurăm că opțiunea *Scene View Mode* din `Preferences/Entities` are valoarea *Runtime Data* 5.

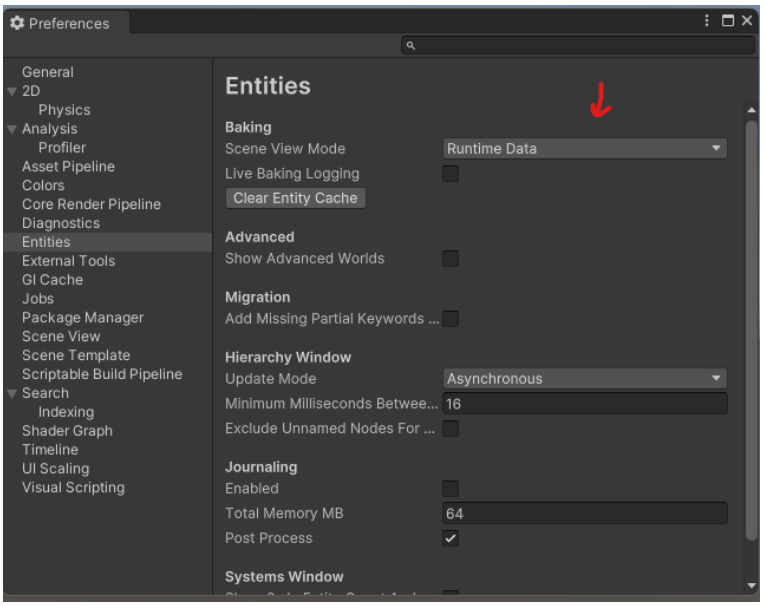

Figure 5: *Scene View Mode* setat pe *Runtime Data*.

2 Job System și compilatorul Burst

Job-urile stau la baza tuturor optimizărilor pe care le putem face folosind DOTS. Un job este o structură prin care putem lucra cu multe date într-un mod eficient. Pentru a obține performanță, job-urile sunt de regulă compilate folosind un nou compilator introdus de Unity, numit Burst Compiler. Acesta convertește cod scris in C# în assembly și este foarte este foarte agresiv când vine vorba de optimizări.

Cel mai simplu tip de job este job-ul de tip `for`. Un job de tip `for` se definește definind o structură care moștenește interfața `IJobFor` și dispune de o metodă care se apelează pentru fiecare iterație a unui loop de tip `for`.

Exemplu:

```
using Unity.Burst;
using Unity.Jobs;

// We want to compile jobs using Burst
// so we can improve the performance.

// We set CompileSynchronously to true so that
// Unity waits for the for the job to compile
// before running the game. This ensures
// that the job version used by Unity
// was compiled using Burst.
[BurstCompile(CompileSynchronously = true)]
partial struct ExampleJobFor : IJobFor
{
    // This functions is called for every
    // iteration of a for loop. Each type of job
    // must define an Execute function, however
    // the arguments it receives. The for job
    // receives an integer as an argument.
    public void Execute(int i)
    {
    }
}
```

Job-urile pot lucra doar cu obiecte de tip unmanaged (pentru care nu se aplică garbage collection). Aceste obiecte includ structurile și tipurile de date primitive. Obiectele rezultate din instanțierea claselor nu fac parte din această categorie. Cum colecțiile standard in C# sunt de tip managed, Unity ne pune la dispoziție tipul de date `NativeArray<T0>` care se comportă precum un array alocat dinamic în C++. Evident, memoria obiectelor de tip `NativeArray<T0>` trebuie eliberată atunci când nu mai este necesară, pentru a nu avea memory leak-uri.

Constructorul unui `NativeArray<T0>` are următoarea formă:

```
public NativeArray<T0>(int length, Unity.Collections.Allocator allocator,
    Unity.Collections.NativeArrayOptions options);
```

Primul argument este lungimea array-ului. Prin al doilea argument se specifică tipul de alocator care va fi folosit pentru array. În funcție de modul în care dorim să lucrăm cu array-ul, putem seta una din următoarele valori:

- `Invalid`
- `None`
- `Temp` - Folosim asta in cazul array-urilor de care avem nevoie foarte puțin înainte de a le dealoca.
- `TempJob` - Similară cu Temp, aceaste array-uri pot fi folosite în interiorul job-urilor, pe când array-urile alocate folosind Temp nu pot.
- `Persistent` - O folosim în cazul array-urilor de care avem nevoie multă vreme în executarea jocului.
- `AudioKernel`

Prin ultimul argument al constructorului specificăm dacă dorim ca memoria alocată să fie inițializată cu valoarea 0, sau sa fie neinițializată.

Când terminăm de folosit un array, memoria acestuia trebuie eliberată folosind metoda `Dispose`, similară cu `delete[]` din C++.

Exemplu de array în care inserăm valori random:

```
using Unity.Collections;

using Random = System.Random;
...
private void NativeMemory()
{
        var random = new Random(0);

        var length = 1000000;
        var nativeArray = new NativeArray<int>(length, Allocator.TempJob,
            NativeArrayOptions.UninitializedMemory);

        for (var i = 0; i < length; i++)
                nativeArray[i] = random.Next();

        nativeArray.Dispose();
}
```

Exemplu de job care incrementează valorile unui array:

```
[BurstCompile(CompileSynchronously = true)]
partial struct IncrementJob : IJobFor
{
        public NativeArray<int> Array;

        public void Execute(int i) =>
                Array[i]++;
}
```

Pentru a porni un Job se folosesc metodele `Schedule` sau `ScheduleParallel`. Acestea returnează un obiect de tip `JobHandle` prin care se poate determina dacă job-ul și-a terminat executarea și prin care se poate aștepta terminarea acestuia.

Metoda `Schedule` primește ca argument numărul de iterații al for-ului și un obiect de tip `JobHandle` care reprezintă dependința acestui job. Acest job se va executa o dată ce job-ul de care depinde își va termina executarea. Job-ul va fi executat pe un singur thread, paralel cu thread-ul principal al aplicației.

Metoda `ScheduleParallel` este asemănătoare cu metoda `Schedule` cu observația că aceasta pronește mai multe worker threads pentru a executa job-ul, nu unul singur. Această metodă mai primește un argument numit `innerLoopBatchCount` prin care se specifică numărul de iterații al for-ului de care se va ocupa un worker thread.

De exemplu, pentru un `for` cu 100 de iterații și 10 `innerLoopBatchCount` vom avea 10 worker threads care se vor ocupa de următoarele range-uri pentru `i`:

Worker thread id	Range
0	0..9
1	10..19
2	20..29
3	30..39
4	40..49
5	50..59
6	60..69
7	70..79
8	80..89
9	90..99

Ca un rule of thumb, vom dori să folosim valori mari pentru `innerLoopBatchCount` atunci când computația unei singure iterații durează puțin și valori mici atunci când computația unei singure iterații este complexă.

Pentru a aștepta ca un job căruia i-am dat `Schedule` să-și termine executarea, putem folosi metoda `Complete`.

Exemplu de folosire al job-ului definit anterior:

```
// Display the initial numbers
var numbersToDisplay = 5;
var initialOutput = "";
for (var i = 0; i < Mathf.Min(numbersToDisplay, length); i++)
        initialOutput += nativeArray[i] + " ";
Debug.Log(initialOutput);

var innerLoopBatchCount = 100;

new IncrementJob
        {
                Array = nativeArray // Set the variables of the job
        }                           // Create an instance of the job and set
                                    // its variables
        .ScheduleParallel(length,
                innerLoopBatchCount,
                default)            // Schedule the job
        .Complete();                // Wait for the job to finish executing

// Display the values after they were implemented
var finalOutput = "";
for (var i = 0; i < Mathf.Min(numbersToDisplay, length); i++)
        finalOutput += nativeArray[i] + " ";
Debug.Log(finalOutput);
```

În continuare vom explora un exemplu de Job care actualizează mai multe poziții în funcție de vitezele atribuite acestora. Pentru aceasta, vom lucra cu numere de tip `float`. Pentru astfel de calcule, Unity dispune de o nouă librărie de matematică, Unity.Mathematics care lucrează cu tipuri de date similare cu cele din HLSL (High Level Shading language, limbajul de programare în care sunt scrise shaderele din Unity). Acestă librărie de matematică este optimizată pentru lucrul cu Job-uri, iar Burst încearcă pe cât posibil să folosească SIMD (Single Instruction Multiple Data) atunci când această librărie este folosită.

Pentru a optimiza și mai mult calculele, îi vom specifica compilatorului Burst că dorim să reordoneze operațiile matematice astfel încât performanța să fie mai bună. De exemplu evaluarea expresiei `a + b * c` poate fi mai ineficientă decât evaluarea expresiei `b * c + a`. De asemenea, putem seta precizia funcțiilor trigonometrice `sin` și `cos`. Pentru a îi specifica compilatorului Burst aceste lucruri, putem proceda în felul următor:

```
[BurstCompile(FloatPrecision.Standard, FloatMode.Fast, CompileSynchronously = true)]
```

Folosind această linie, precizia funcțiilor trignometrice va fi cea standard, iar compilatorul va reordona operațiile matematice acolo unde este loc pentru optimizări.

Exemplu de script în care se actualizează poziții (ele nu sunt aplicate pe niciun obiect, este doar un exemplu didactic):

```
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;
using UnityEngine;

// We use this so we can access methods from
// Unity.Mathematics.math directly
using static Unity.Mathematics.math;

public class UpdatePositions : MonoBehaviour
{
        [BurstCompile(FloatPrecision.Standard, FloatMode.Fast, CompileSynchronously =
            true)]
        partial struct ApplyVelocities : IJobFor
        {
                public NativeArray<float3> Positions;

                // We specify that the Velocities array is read-only
                // If we try to access it, we'll get compiler errors.
                // It's a good practice to specify if your data
                // it read-only or not. Same goes for write-only data.
                [ReadOnly]
                public NativeArray<float3> Velocities;

                public float              DeltaTime;

                public void Execute(int i) =>
                        Positions[i] += Velocities[i] * DeltaTime;
        }

        [SerializeField, Range(1, 10000)]
        private int                _positionsCount = 5000;

        [SerializeField, Range(1.0f, 100.0f)]
        private float              _speed          = 10.0f;
```

```
        private NativeArray<float3> _positions;
        private NativeArray<float3> _velocities;

        private void OnEnable()
        {
                _positions  = new NativeArray<float3>(_positionsCount,
                    Allocator.Persistent); // Initialize the positions with 0
                _velocities = new NativeArray<float3>(_positionsCount,
                    Allocator.Persistent, NativeArrayOptions.UninitializedMemory);

                // Generate the velocities
                for (var i = 0; i < _positionsCount; i++)
                        _velocities[i] = up() * _speed; // up() is a method
                                                        // from the math
                                                        // class. It returns
                                                        // the float3(0.0f,
                                                        // 1.0f, 0.0f)
                                                        // vector
        }

        private void OnDisable()
        {
                // Makes sure we deallocate the memory we used.
                _positions.Dispose();
                _velocities.Dispose();
        }

        private void Update()
        {
                var innerLoopBatchCount = 64;

                // Start the job and wait for it to complete
                new ApplyVelocities
                        {
                                Positions  = _positions,
                                Velocities = _velocities,
                                DeltaTime  = Time.deltaTime
                        }
                        .ScheduleParallel(_positionsCount, innerLoopBatchCount,
                            default)
                        .Complete();
        }
}
```

Putem inspecta codul assembly generat de către Burst. Deschidem `Jobs > Burst > Open Inspector` și în partea stângă a ferestrei selectăm job-ul pe care vrem să-l inspectăm 6.

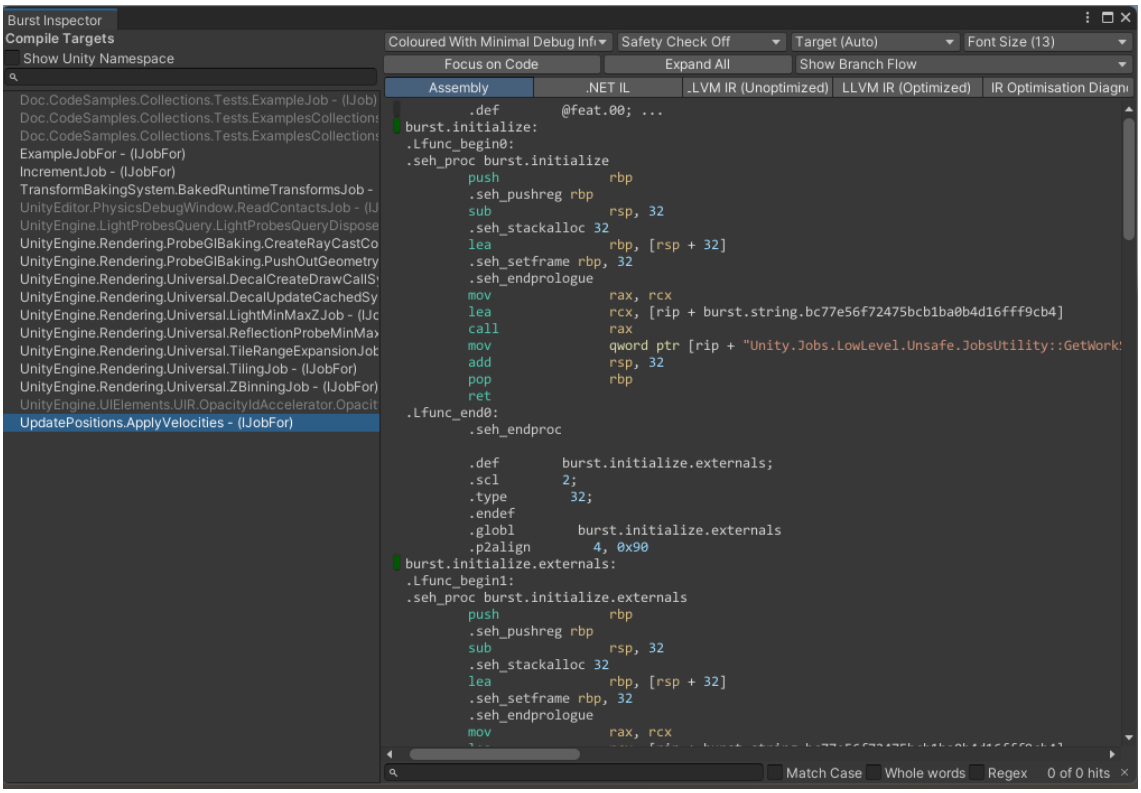

Figure 6: Inspectorul pentru codul generat de Burst.

Dacă examinăm metoda `Execute` definită anterior, vom vedea un rezultat asemănător cu acesta:

```
        .Ltmp1:
        .cv_inline_site_id 1 within 0 inlined_at 1 0 0
=== UpdatePositions.cs(28, 1)            Positions[i] += Velocities[i] * DeltaTime;
        vmovsd          xmm1, qword ptr [rcx + rbx]
        vinsertps       xmm1, xmm1, dword ptr [rcx + rbx + 8], 32
.Ltmp2:
        vmovsd          xmm2, qword ptr [rdx + rbx]
        vinsertps       xmm2, xmm2, dword ptr [rdx + rbx + 8], 32
.Ltmp3:
        vfmadd213ps      xmm2, xmm0, xmm1
.Ltmp4:
        vmovlps         qword ptr [rcx + rbx], xmm2
        vextractps      dword ptr [rcx + rbx + 8], xmm2, 2
```

Se observă că Burst nu a vectorizat codul, deoarece pentru o singură iterație se se calculează o singură valoare. Putem vectoriza codul manual, calculând 4 poziții per iterație folosind matrici.

2.1 Vectorizarea job-urilor

Primul pas pentru a vectoriza codul este să reinterpretăm array-urile, astfel încât fiecare element din array să reprezinte o matrice de dimensiune 3×4 care conține informațiile a 4 vectori. Implicit, librăria de matematică pe care o folosim împreună cu Burst păstrează matricele în memorie după coloană! Așadar, primele 4 poziții vor fi reinterpretate în modul următor 7:

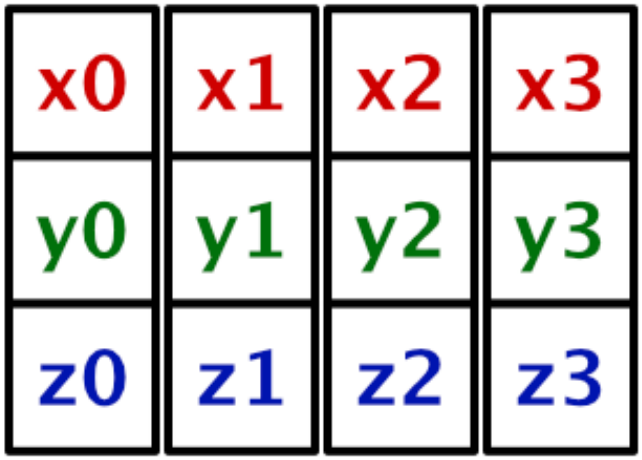

Figure 7: 4 poziții reinterpretate ca matrice de 3×4.

Pentru a ne asigura că actualizăm toate valorile și nu există overflow, vom adăuga elemente în plus array-urilor astfel ca numărul pozițiilor și al vitezelor să fie un multiplu de 4.

```
private NativeArray<float3x4> _positions;
private NativeArray<float3x4> _velocities;

private void OnEnable()
{
        // We check if the number of vectors we have is a multiple of 4.
        // If it's not, we add padding elements.
        if ((_positionsCount & 0b11) != 0)
                _positionsCount += 4 - (_positionsCount & 0b11);

        _positions = new NativeArray<float3x4>(_positionsCount / 4, // We now
                                                                    // work on
                                                                    // bulks of
                                                                    // 4
                                                                    // vectors
                Allocator.Persistent);
        _velocities = new NativeArray<float3x4>(_positionsCount / 4,
                Allocator.Persistent,
                NativeArrayOptions.UninitializedMemory);

        for (var i = 0; i < _positionsCount / 4; i++)
                _velocities[i] = float3x4(up(), up(), up(), up()) * _speed;
}
```

Pentru a efectua efectuații pe matrice folosind SIMD trebuie să operăm pe coloane. Dacă este nevoie de a schimba valorile componentelor a mai multor vectori simultan, uneori este fezabil să transpunem matricele pe care le folosim, așa cum se poate observa în figura următoare 8:

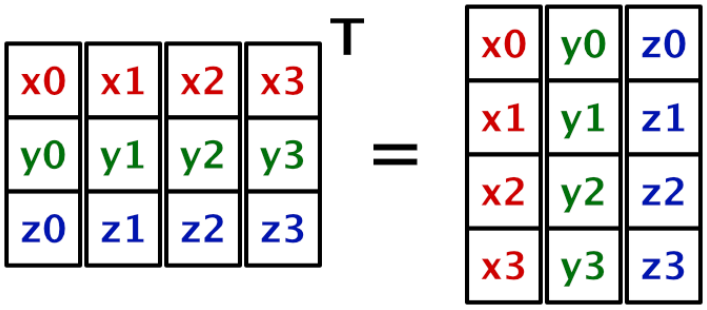

Figure 8: 4 poziții reinterpretate ca matrice de 3×4.

În cazul nostru nu este nevoie de transpunere deoarece nu lucrăm pe componente individuale ale vectorilor, deci putem actualiza Job-ul direct:

```
partial struct ApplyVelocities : IJobFor
{
        public NativeArray<float3x4> Positions;

        [ReadOnly]
        public NativeArray<float3x4> Velocities;

        public float                 DeltaTime;

        public void Execute(int i) =>
                Positions[i] = float3x4(
                        Positions[i].c0 + Velocities[i].c0 * DeltaTime,
                        Positions[i].c1 + Velocities[i].c1 * DeltaTime,
                        Positions[i].c2 + Velocities[i].c2 * DeltaTime,
                        Positions[i].c3 + Velocities[i].c3 * DeltaTime);
}
```

Pentru a planifica Job-ul este nevoie să modificăm numărul de iterații:

```
new ApplyVelocities
        {
                Positions  = _positions,
                Velocities = _velocities,
                DeltaTime  = Time.deltaTime
        }
        .ScheduleParallel(_positionsCount / 4, innerLoopBatchCount, default)
        .Complete();
```

Analizând codul rezultat, putem observa că acum se efectuează mai multe operații pe vectori pentru o singură iterație. Presupunând că există porturi disponibile pentru instrucțiuni, acestea se pot executa în paralel 9.

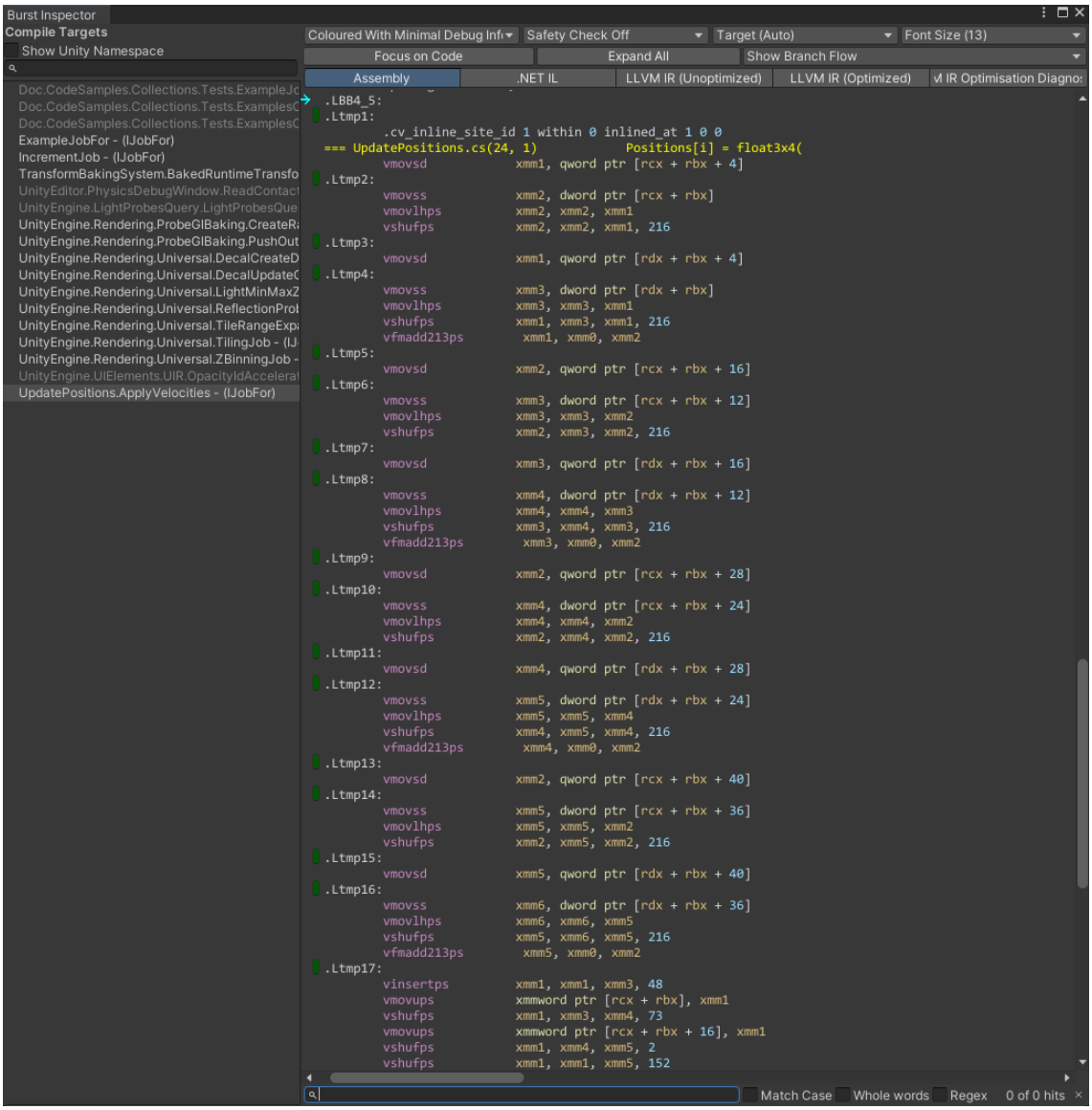

Figure 9: Codul generat după vectorizarea job-ului de actualizare a pozițiilor.

2.2 Măsurarea performanței

Pentru a măsura timpul de executare al unui Job este nevoie să folosim Profiler-ul din Unity `Window > Analysis > Profiler` 10.

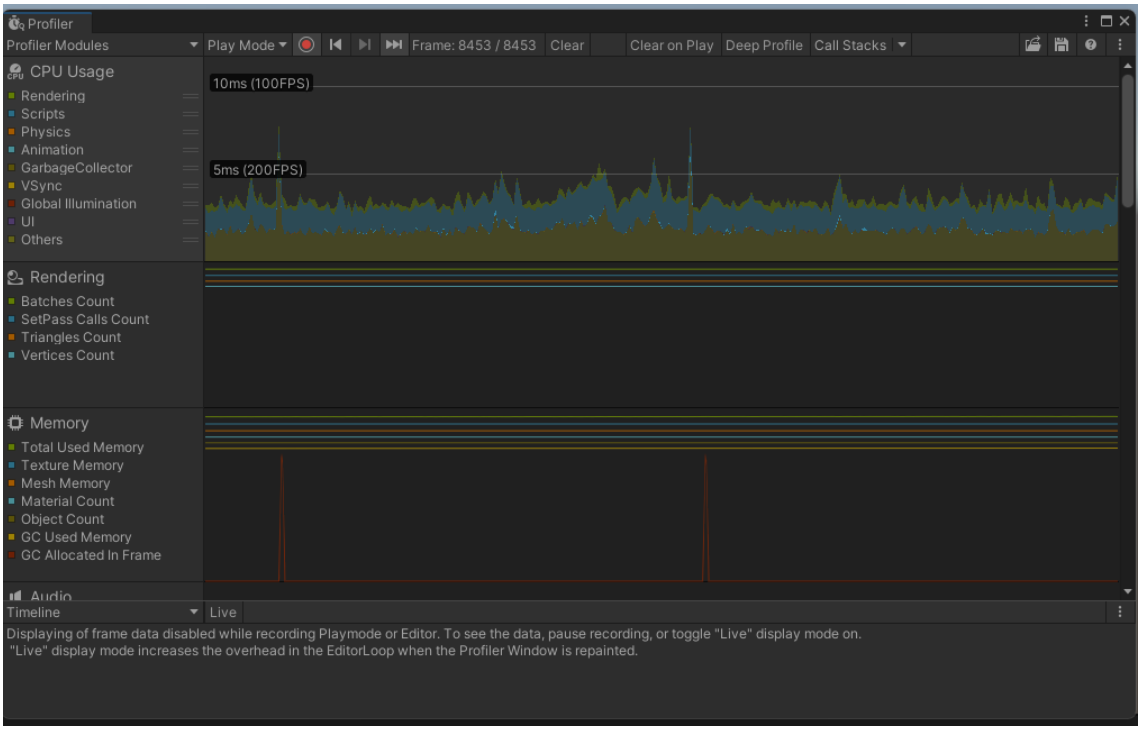

Figure 10: Unity Profiler.

În partea de sus a profiler-ului avem un grafic al timpului scurs pentru fiecare cadru. Dacă apăsăm click pe o poziție a acelui grafic, profiler-ul ne va prezenta în partea de jos o cronologie a evenimentelor care au avut loc în acel cadru. Astfel, putem vedea timpii fiecărui script care rulează la un anumit moment de timp. Putem vedea și timpul de executare al job-urilor definite de către noi 11.

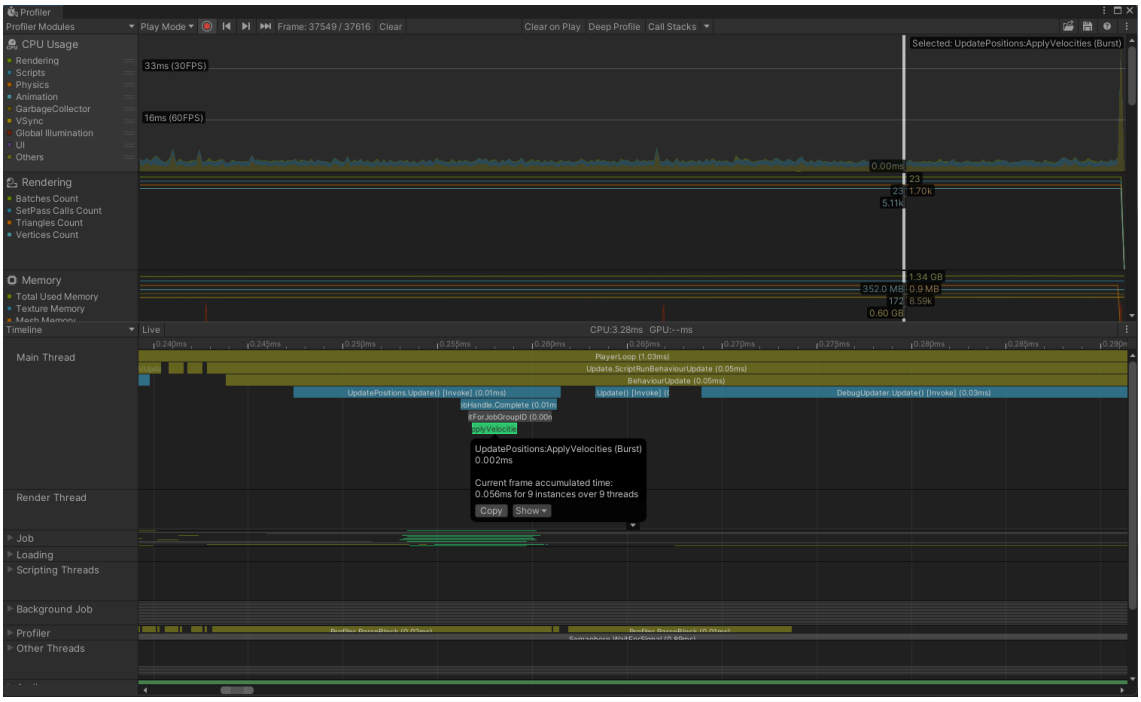

Figure 11: Timpul de executare al unui job.

Când optimizăm o aplicație, de regulă ne dorim să aflăm care sunt cele mai ineficiente părți ale codului nostru. Putem seta Profiler-ul ca în partea de jos să ne arate o ierarhie a lucrurilor care se execută în timpul unui cadru în loc să ne arate cronologia lor. Facem asta prin schimbarea din modul `Timeline` în modul `Hierarchy` 12.

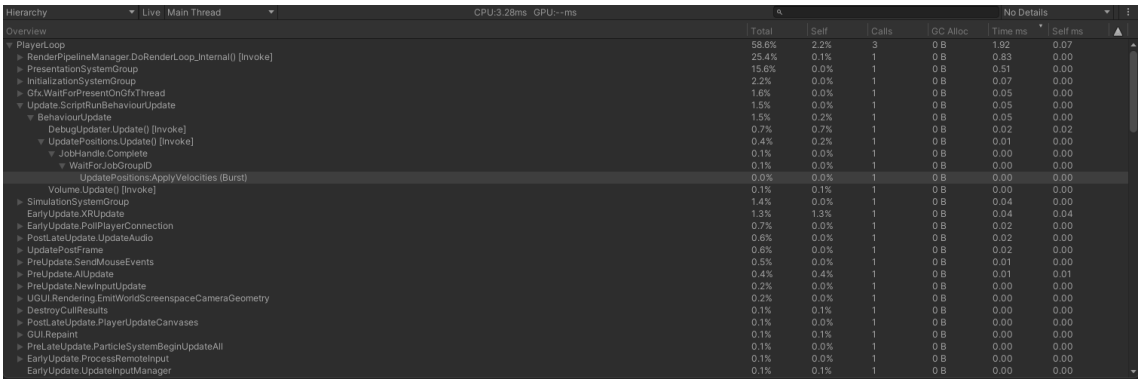

Figure 12: Profiler Hierarchy.

Multe din lucrurile pe care le vedem în profiler sunt datorate pachetelor folosite, pipeline-ului de randare (URP) dar și modului în care funcționează Unity. Când rulăm jocul din editor, în profiler vedem și detalii despre performanța editorului.

Când vrem să măsurăm corect performanța aplicației, trebuie să folosim profiler-ul în afara editorului, direct pe executabilul generat. Doar așa avem garanția că numerele pe care le vedem în Profiler sunt relevante. Pentru a genera un build asupra căruia să putem atașa Profiler-ul, trebuie să facem următoarele setări atunci când facem build 13:

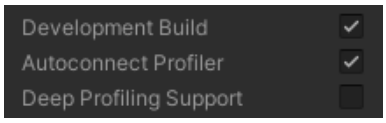

Figure 13: Build pe care putem atașa Profiler-ul.

Există mult mai multe moduri de a măsura performanța în Unity, dar acestea nu reprezintă scopul acestui laborator.

2.3 Cele mai apropiate puncte

Se dau n puncte albastre și n puncte roșii în plan. Pentru fiecare punct albastru se dorește să se găsească indexul celui mai apropiat punct roșu.

Această problemă se poate implementa folosind job-uri în mai multe moduri. Ca exemplu vom implementa modul direct, în care se pargurg toate punctele roșii pentru fiecare punct albastru (o soluție mai eficientă este folosirea unei căutari binare).

Până acum, pentru fiecare `i` al job-urilor, am accesat array-uri la acel index. Dacă am fi accesat alți indecși am fi primit o eroare. Pentru a putea accesa orice element al unui array (chiar și din afara limitelor, deci trebuie să avem grijă) în interiorul job-urilor folosim:

```
[NativeDisableContainerSafetyRestriction]
```

Exemplu de job pentru această problemă:

```
partial struct UpdateMinimums : IJobFor
{
        [ReadOnly]
        public NativeArray<float2> BluePositions;

        [ReadOnly]
        // We specify this so we can access the whole
        // Array, not just the i-th element.
        [NativeDisableContainerSafetyRestriction]
        public NativeArray<float2> RedPositions;

        [WriteOnly]
        public NativeArray<int>    MinIndices;

        public void Execute(int i)
        {
                var result = -1;
                var minDistanceSq = float.MaxValue;

                for (var j = 0; j < RedPositions.Length; j++)
                {
                        var redPos = RedPositions[j];
                        // We don't have to calculate the actual distance,
                        // the square of the distance is enough for our
                        // comparison and it's cheaper to compute.
                        var distanceSq = distancesq(redPos, BluePositions[i]);

                        if (distanceSq < minDistanceSq)
                        {
                                minDistanceSq = distanceSq;
                                result = j;
                        }
                }

                MinIndices[i] = result;
        }
}
```

3 Entity Component Systems

Până acum am folosit Job-uri pentru a optimiza algoritmi generali care nu interacționează cu jocul.

Pentru a aplica astfel de optimizări întregului joc, Unity a introdus un Entity Component System (vezi curs 2).

În mod clasic, în Unity există Obiecte și Componente. Un obiect are mai multe componente, iar fiecare componentă a acestuia determină un comportament al obiectului.

Într-un Entity Component System, există Entități, Componente și Sisteme. O entitate o putem asemăna cu un obiect, în sensul că aceasta conține mai multe componente. În cazul aceste, componentele nu mai definesc un comportament, ci sunt folosite doar pentru a salva date. Sistemele preloucrează toate componentele ale tuturor entităților. Practic, în loc de a avea comportamentul definit de o componentă, acesta este definit de un sistem.

În Unity, putem converti obiectele în entități, componentele în componente ce conțin numai date și putem defini sisteme care lucrează cu acestea.

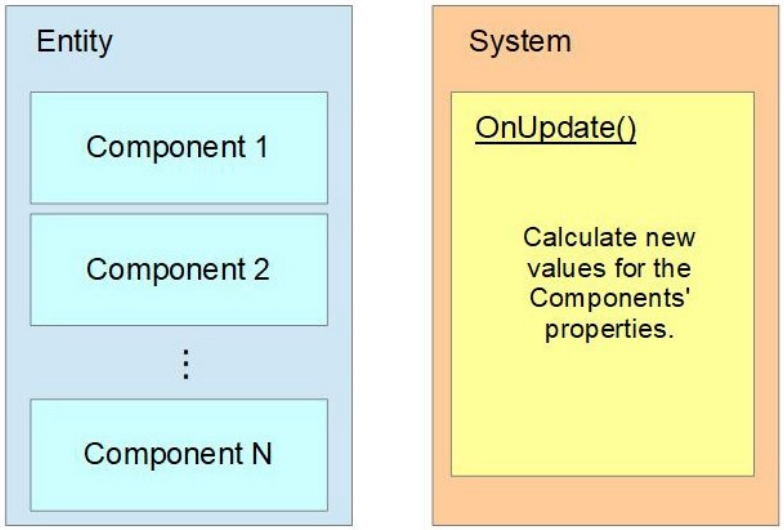

Figure 14: Entități, componente, sisteme.

3.1 Demo

Vom realiza o aplicație în care există un obiect `Spawner` care adaugă constant bile în scenă. Bilele vor avea o viteză care este decrementată atunci când ating podeaua și vor fi eliminate din scenă atunci când viteza acestora este prea mică.

Pentru a folosi Unity ECS (Entity Component System), este necesar să creăm o sub-scenă în care vom plasa obiectele `Hierarchy > Cele 3 puncte din dreptul scenei curente > New Empty Sub Scene...`15.

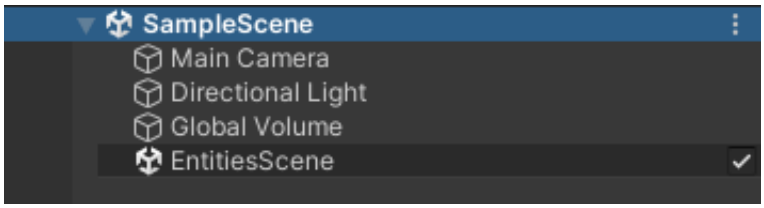

Figure 15: Empty Sub Scene.

În sub-scenă vom crea o sferă 16 căreia îi vom elimina componenta `Sphere Collider` deoarece aceasta nu poate fi convertită în componentă cu care poate lucra ECS-ul. În inspectorul pentru acest obiect putem observa în partea de jos componentele pe care aceasta le va avea, după ce componentele normale vor fi convertite la componente de ECS 17.

Figure 16: Sferă adăugată în sub-scenă.

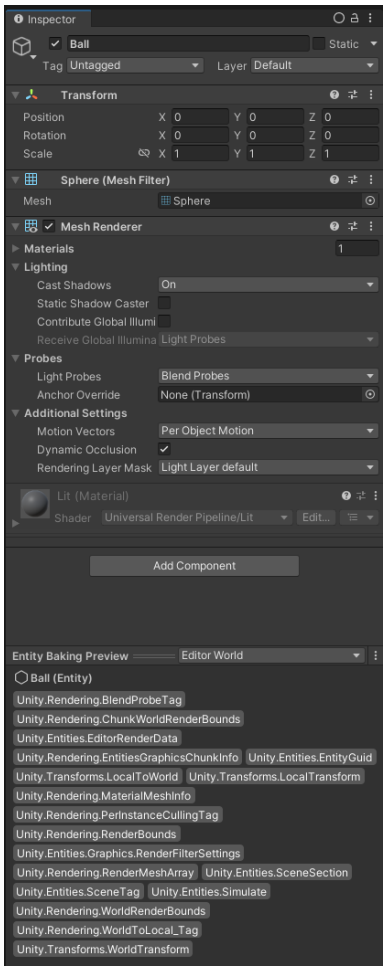

Figure 17: Inspectorul unui obiect care va fi transformat în entitate.

Vom defini o componentă normală pentru sfera noastră, iar apoi vom specifica cum trebuie Unity să convertească această componentă de la o componentă normală la una pentru ECS. Componenta care urmează să fie convertită se numește `Authoring`. Deci vom crea un script numit *BallAuthoring.cs*.

```
using Unity.Entities;
using Unity.Mathematics;
using UnityEngine;

// The regular component that we want to convert
class BallAuthoring : MonoBehaviour
{
        // We need to define a baker that specifies how the component
        // should be converted to an ECS component.
        class BallBaker : Baker<BallAuthoring>
        {
                // A baker must define the Bake function
                public override void Bake(BallAuthoring authoring) =>
                        AddComponent<Ball>();
        }
}

// The new component must be a structure
// that only contains data.
// The data it contains must be unmanaged
struct Ball : IComponentData
{
        // We only need to know the speed of the ball
        // to update its position.
        public float3 Speed;
}
```

Vom adăuga această componentă sferei create anterior. Putem observa în inspector noua componentă adăugată și datele acesteia 18. Vom crea un *Prefab* pentru această sferă, apoi o vom elimina din sub-scenă.

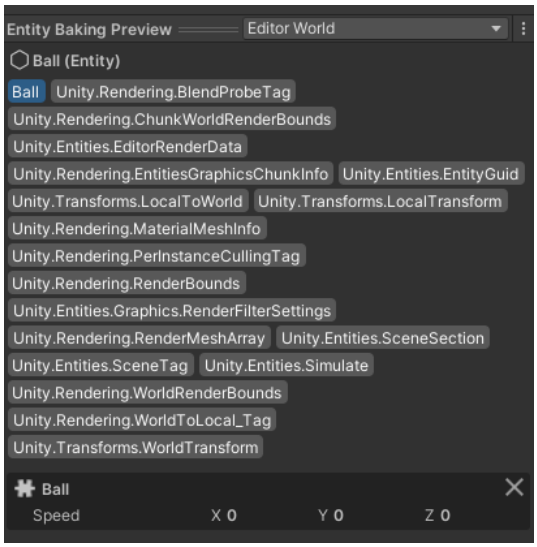

Figure 18: Componenta definită de noi.

Pentru a adăuga constant obiecte în scenă este nevoie de un *Spawner*. Vom crea în sub scenă un obiect gol numit *Spawner* pentru care vom defini o componentă similară cu cea definită în cazul bilei. Vom numi acest script *SpawnerAuthoring.cs*. Această componentă va fi folosită de un sistem care operează pe acest tip de componente și instanțiază bile la poziția spawner-ului. Deci, este nevoie să avem o referință către prefab în interiorul acestei componente. Pentru a avea o referință către un obiect folosind ECS se folosește tipul `Entity` și funcția `GetEntity` pentru a lua o astfel de referniță de la un obiect normal.

```
using Unity.Entities;
using UnityEngine;

public class SpawnerAuthoring : MonoBehaviour
{
        // Regular reference to the Ball prefab.
        public GameObject BallPrefab;

        class SpawnerBaker : Baker<SpawnerAuthoring>
        {
                public override void Bake(SpawnerAuthoring authoring)
                {
                        AddComponent(new Spawner
                        {
                                // Convert the object reference to an entity reference
                                BallPrefab = GetEntity(authoring.BallPrefab)
                        });
                }
        }
}

public struct Spawner : IComponentData
{
        public Entity BallPrefab;
}
```

Vom adăuga această componentă obiectului *Spawner* și vom seta referința către prefab folosind Drag and Drop.

Pentru a defini comportamentul spawner-ului este necesar să definim un sistem care să lucreze cu această componentă. Sunt mai multe moduri de a defini sisteme în Unity, fiecare cu avantaje și dezavantaje. În acest laborator vom prezenta un singur tip de sistem, și anume, sistemul în care toate componentele sunt parcurse folosind un Job. Folosim această metodă pentru că este cea mai versatilă și mai eficientă metodă (deși implică mai mult boilerplate).

Un job nu se poate executa direct pe pentru fiecare componentă *Spawner* din scenă, avem nevoie de un wrapper prin intermediul căruia job-urile să poată accesa componenta. Acest tip de wrapper se numește `Aspect`. Un aspect este o structură care implementează interfața `IAspect` și care definește o referință către componenta pe care o reprezintă. Această referință poate să fie `read-only` sau `read-write`. Vom defini un `Aspect` pentru componenta Spawner, numit *SpawnerAspect.cs*.

```
using Unity.Entities;

readonly partial struct SpawnerAspect : IAspect
{
        // We add this property so we can easily access the reference to the
        // ball prefab inside a job. This way we don't have to explicitly
        // access the Spawner reference every time we want to access the ball
        // prefab reference.
        public          Entity         BallPrefab  => Spawner.ValueRO.BallPrefab;

        public readonly Entity         Self;    // We have a reference to the spawner
                                                // entity.
                                                // It is set automatically by Unity.

        public readonly RefRO<Spawner> Spawner; // We have a read-only
                                                // reference to the Spawner
                                                // It is set automatically by Unity.
}
```

Având acest aspect, putem defini sistemul care operează pe *Spawnere*. Un sistem are următoarea structură:

```
// It must be a partial structure that implements the ISystem interface
[BurstCompile]
partial struct ExampleSystem : ISystem
{
        // Called when the system is created.
        [BurstCompile]
        public void OnCreate(ref SystemState state)
        {
        }

        // Called when the system is destroyed.
        [BurstCompile]
        public void OnDestroy(ref SystemState state)
        {
        }

        // Called on every frame, similar to Update
        // from Monobehaviour
        [BurstCompile]
        public void OnUpdate(ref SystemState state)
        {
        }
}
```

Vom defini sistemul *SpawnSystem.cs*:

```
using Unity.Burst;
using Unity.Collections;
using Unity.Entities;
using Unity.Transforms;

// We use an IJobEntity instead of an IJobFor
[BurstCompile]
partial struct BallSpawn : IJobEntity
{
        [ReadOnly]
        // This variable helps us get the transform of a given entity.
        public ComponentLookup<WorldTransform> WorldTransformLookup;
        // The ECB object helps us do operations on Entities (such as
        // adding components, removing components, instantiating
        // entities, and so on...)
        public EntityCommandBuffer             ECB;

        // The argument we recive is now an Aspect though which
        // we operate on the component.
        public void Execute(in SpawnerAspect spawner)
        {
                // Instantiate the prefab.
                var instance = ECB.Instantiate(spawner.BallPrefab);

                // Get the World Transform of the spawner.
                var spawnerWorldTransform = WorldTransformLookup[spawner.Self];



                // Create the Local transform of the ball based on the position of
                // the spawner
                var ballTransform =
                    LocalTransform.FromPosition(spawnerWorldTransform.Position);

                ballTransform.Scale = WorldTransformLookup[spawner.BallPrefab].Scale;

                // Update the components of the new ball, setting
                // its transform and its speed.
                ECB.SetComponent(instance, ballTransform);
                ECB.SetComponent(instance, new Ball
                {
                        Speed = spawnerWorldTransform.Forward() * 20.0f
                });
        }
}

[BurstCompile]
partial struct SpawnSystem : ISystem
{
        // This variable helps us get the transform of a given entity.
        private ComponentLookup<WorldTransform> _worldTransformLookup;

        [BurstCompile]
```

```
        public void OnCreate(ref SystemState state) =>
                _worldTransformLookup = state.GetComponentLookup<WorldTransform>(true);

        [BurstCompile]
        public void OnDestroy(ref SystemState state)
        {
        }

        [BurstCompile]
        public void OnUpdate(ref SystemState state)
        {
                // World Transform lookups must be updated before using them
                // in a system.
                _worldTransformLookup.Update(ref state);

                // The ECB object helps us do operations on Entities (such as
                // adding components, removing components, instantiating
                // entities, and so on...)
                var ecbSingleton =
                        SystemAPI.GetSingleton<BeginSimulationEntityCommandBufferSystem.Singleton>();
                var ecb = ecbSingleton.CreateCommandBuffer(state.WorldUnmanaged);

                // Create the job that will run for each Spawner in the scene.
                var spawnerJob = new BallSpawn
                {
                        WorldTransformLookup = _worldTransformLookup,
                        ECB = ecb
                };

                // Schedule the spawner to run on a different thread
                // without blocking the main thread.
                spawnerJob.Schedule();
        }
}
```

Dacă rulăm, aplicația ar trebui ca acum să se spawneze bile constant la poziția unde se află obiectul *Spawner*20. Vom dori ca bilele să se deplaseze pe o direcție de sus-înainte, așa că vom seta rotația spawner-ului pe axa X la 45 de grade.

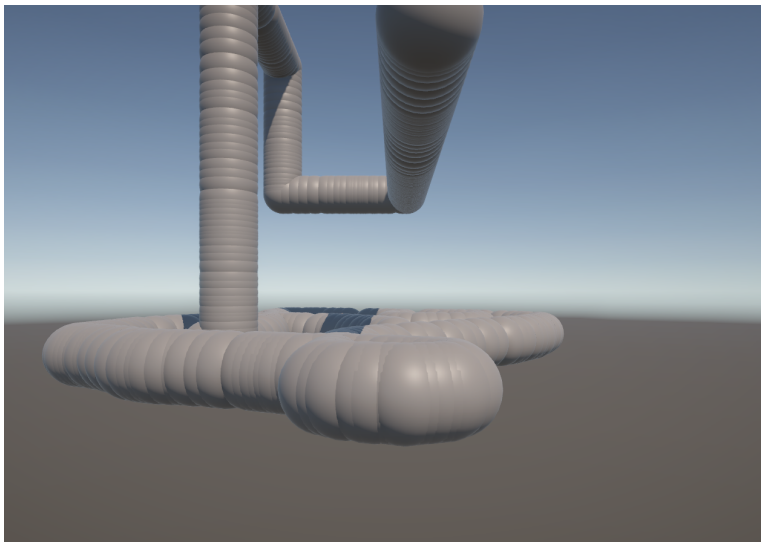

Figure 19: Bile spawnate la poziția unui spawner care este deplasat în scenă.

Pentru a face bilele să se deplaseze și să fie eliminate când viteza lor este prea mică trebuie să definim un sistem pentru bile. Din nou, vom avea nevoie de un `Aspect` pentru acest sistem. Îl vom numi *BallAspect.cs* și ne va pune la dispoziție informații precum poziția și viteza bilei.

```
using Unity.Entities;
using Unity.Mathematics;
using Unity.Transforms;

readonly partial struct BallAspect : IAspect
{
        public          float3          Position
        {
                get => Transform.LocalPosition;
                set => Transform.LocalPosition = value;
        }

        public          float3          Speed
        {
                get => Ball.ValueRO.Speed;
                set => Ball.ValueRW.Speed = value;
        }

        public readonly Entity          Self;
        public readonly TransformAspect Transform;

        // We use a read-write reference now because we want to
        // modify the position of the ball
        public readonly RefRW<Ball>     Ball;
}
```

Acum se poate defini sistemul pentru deplasarea bilelor *BallsUpdate.cs*:

```
using Unity.Burst;
using Unity.Entities;
using Unity.Mathematics;

using static Unity.Mathematics.math;

[BurstCompile]
partial struct BallUpdate : IJobEntity
{
        public EntityCommandBuffer.ParallelWriter ECB;
        // We cannot get the time between the frames inside a job
        // so we get it from the system instead.
        public float                              DeltaTime;

        // We get the chunkIndex because we want to run this job
        // in paralel, and our paralel ECB expects the chunk id when
        // doing operations.
        public void Execute([ChunkIndexInQuery] int chunkIndex, BallAspect ball)
        {
                var gravity = new float3(0.0f, -9.82f, 0.0f);
                var invertY = new float3(1.0f, -1.0f, 1.0f);

                // Update the position of the ball accordingly
                ball.Position += ball.Speed * DeltaTime;

                // Change the velocity when hitting the floor.
                if (ball.Position.y < 0.0f)
                {
                        ball.Position *= invertY;
                        ball.Speed *= invertY * 0.8f;
                }

                // Apply gravity.
                ball.Speed += gravity * DeltaTime;

                // Destroy the entity when its speed is too low.
                var speed = lengthsq(ball.Speed);
                if (speed < 0.1f)
                        ECB.DestroyEntity(chunkIndex, ball.Self);
        }
}

[BurstCompile]
partial struct BallsUpdateSystem : ISystem
{
        [BurstCompile]
        public void OnCreate(ref SystemState state)
        {
        }

        [BurstCompile]
        public void OnDestroy(ref SystemState state)
        {
```

```
        }

        [BurstCompile]
        public void OnUpdate(ref SystemState state)
        {
                var ecbSingleton =
                        SystemAPI.GetSingleton<EndSimulationEntityCommandBufferSystem.Singleton>();
                var ecb = ecbSingleton.CreateCommandBuffer(state.WorldUnmanaged);

                var ballJob = new BallUpdate
                {
                        ECB = ecb.AsParallelWriter(),
                        DeltaTime = SystemAPI.Time.DeltaTime // instead of
                                                             // Time.deltaTime
                };

                ballJob.ScheduleParallel();
        }
}
```

Dacă rulăm ar trebui să avem următorul rezultat:

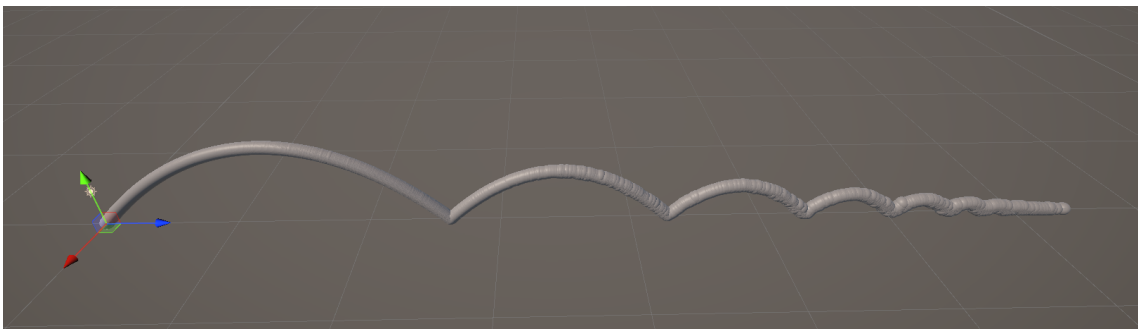

Figure 20: Bile care se deplasează în scenă până într-un punct.

4 Cerințe de laborator

4.1 Înmulțirea matricelor (0.2p bonus)

Se dau două matrice A și B, amândouă având dimensiune 1000×1000. Să se calculeze matricea $C = AB$ folosind sistemul de job-uri din Unity, și optimizările de caching prezentate în primul curs.

Hint: Se transpune matricea B în memorie, și se calculează pe rând rezultatele câte unui block de din matricea C (preferabil să fie block-uri de dimensiune 100×100). O iterație a unui job va fi responsabilă pentru `block_size` linii și va calcula toate block-urile de pe acele linii succesiv. Job-ul este programat să releze iterațiile în paralel. 21

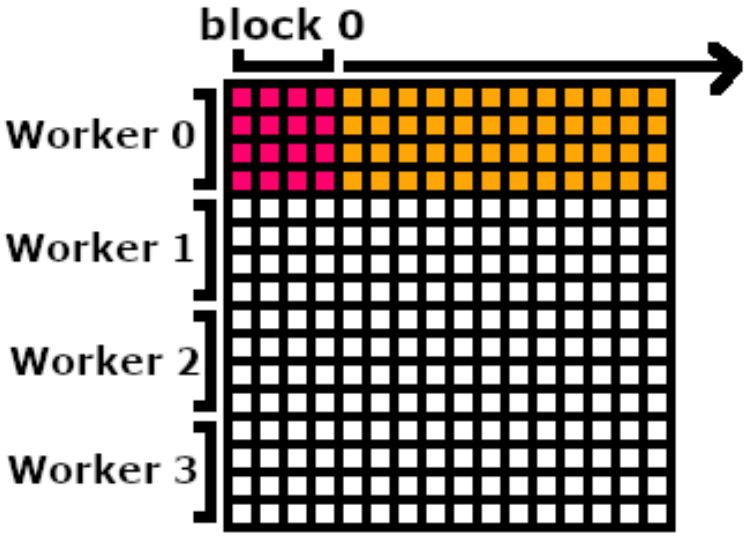

Figure 21: Distribuirea thread-urilor pentru înmulțirea matricelor.

4.2 Burst (0.1p bonus)

Asigurați-vă că job-ul definit anterior a fost compilat de către Burst și analizați codul assembly generat.

4.3 Profiler (0.1p bonus)

Măsurați timpul de executare al algoritmului implementat anterior folosind Profiler-ul în timp ce aplicația rulează (în afara editorului).

5 Bibliografie/ Resurse

- Catlikecoing Unity Tutorials - Basics
- Unity Entity Component System Samples