Basics of DOTS: Jobs and Entities

Basics of DOTS: Jobs and Entities – Unity Learn

Khóa học này bao gồm 3 hướng dẫn cơ bản của package Entities, C# Job System và các phần cơ bản khác của Unity’s Data-Oriented Technology Stack (DOTS). Tạo điều kiện thuận lợi để viết code C# hiệu năng cao.

Hoàn thành khóa học này bạn có thể:

- Viết code đa luồng với C# Job System.
- Tạo entites trong Runtime.
- Load entities từ scenes và sử dụng baking.
- Viết hệ thống để truy cập và sử dụng entities.

- Làm quen với DOTS
- Bắt đầu với Job System

Làm quen với DOTS

Mục tiêu

Kết thúc hướng dẫn này bạn có thể:

- Hiểu các phần của Data-Oriented Technology Stack (DOTS) trong Unity.
- Xác định trường hợp có lợi khi sử dụng DOTS.

Tổng quan

Hiệu năng CPU và bộ nhớ là những thành phần quan trọng cần cân nhắc để đem tới trải nghiệm tốt khi phát triển ứng dụng. Data-Oriented Technology Stack là kiến trúc phần mềm cho phép lập trình viên tận dụng tốt nhất phần cứng của họ.

DOTS là tập hợp công nghệ đem tới một cách khác để làm việc với Unity. Nó là cách tiếp cận khác khi nghĩ về code, dữ liệu sử dụng data-oriented design DOD thay vì object-oriented design (OOP). DOTS cho phép bạn tận dụng CPU đa lõi để xử lý dữ liệu song song. Với DOTS, bạn có thể tạo ra ứng dụng với hiệu năng tốt hơn và sử dụng phần cứng tốt hơn.

Hướng dẫn này dành cho cả người biết và không biết về kỹ thuật, là người mới với DOTS. Cung cấp hiểu biết cơ bản về DOTS và giải đáp các câu hỏi thường gặp khi tìm hiểu về DOTS.

Các package và tính n?ng.

DOTS là tập hợp các gói và tính năng cho phép viết code hiệu suất cao:

- **C# Job System** mang tới cách đơn giản và an toàn để viết code đa luồng tận dụng CPU đa lõi ngày nay.
- **Burst Compiler** là compiler tối ưu hóa cho C# tạo ra code nhanh hơn Mono hoặc ngay cả ILCPP.
- **Unity.Mathematics** mang tới thư viện toán được tối ưu đặc biệt khi sử dụng với code sử dụng Burst-Compiler.
- **Unity.Collections** cung cấp các collection phổ biến như Lists và hashmaps. Không giống như C# collections, collections này không được quản lý nên có thể sử dụng trong code Burst-Compiled. Collections này cũng hỗ trợ kiểm tra an toàn giúp bạn chắc chắn rằng chúng được sử dụng đúng trong jobs.
- **Unity.Entities** cung cấp một dạng kiến trúc Entity Component System (ECS). Ngắn gọn thì entities là một thay thế nhẹ hơn, có hiệu quả hơn cho GameObject, và chúng được xử lý bởi các đơn vị code gọi là systems.

- **Unity.Entities.Graphics** renders entities sử dụng URP (The Universal Rendering Pipeline) hoặc HPRP (The High-Definition Rendering Pipeline).

Một số package khác được xây dựng dựa trên Entities và lõi DOTS:

- **Unity.Physics** đem tới hệ thống vật lý. *(nguyên văn: The Unity.Physics package provides a deterministic rigid body dynamics system and spatial query system.)*
- **Unity.Netcode** (được biết đến là **Netcode for Entities**) cung cấp hệ thống multiplayer qua mạng với một máy chủ có thẩm quyền và một client có thể dự đoán.

Hiện tại, Unity chưa có giải pháp cho *animation*, *audio*, hay *UI* dựa trên Entities vậy nên các dự án dựa trên entities phải trở lại với GameObject hoặc các cách thay thế khác cho những tính năng này. Ví dụ, nến bạn tạo một game dựa trên entities với nhân vật được tạo anim thì bạn có thể xử lý nhân vật dưới dạng entities trong logic nhưng hiển thị nó như một GameObjects. Điều này đòi hỏi đồng bộ trạng thái giữa từng entity nhân vật và GameObject của nó, điều này (được Unity cho rằng) là có thể chấp nhận được với các dự án ở quy mô vừa và nhỏ.

Các khái niệm chính của DOTS.

Các video dưới đây giới thiệu nhanh về các khái niệm cơ bản trong DOTS:

- The C# Job system (11 minutes).
- ECS Entities and components (10 minutes).
- ECS Systems (7 minutes).

Có nên sử dụng DOTS?

Bất kỳ dự án nào bị nghẽn cổ chai CPU trong code đều nên được xem xét triển khai lại với slow code như Burst-Compiled jobs. Burst-Compiled code không chỉ nhanh hơn Mono-Compiled nhiều lần hay thậm chí tương đương IL2CPP-compiled, jobs còn cho phép bạn chia nhỏ công việc cho tất cả các lõi của CPU.

Tin tốt là Burst-Compiled jobs còn thể được tích hợp vào một dự án có sẵn. Ngay cả khi dự án đó không sử dụng DOTS. Bạn có thể sẽ cần copy dữ liệu vào và ra khỏi Unity.Collections nhưng mặt khác thêm Burst-Compiler jobs thường không yêu cầu tái cấu trúc code một cách nghiêm ngặt.

Điều này ít đúng hơn với package Entities. Mặc dù đôi khi có thể tích hợp có chọn lọc các entities để thực hiện các tính năng cụ thể, kiến trúc ECS áp đặt các cấu trúc code riêng của nó và do đó thường tạo nên nền tảng chung cho dự án của bạn.

Dưới đây là 4 lý do để sử dụng Entities cho dự án của bạn:

- Dự án của bạn có nhiều thành phần tĩnh, chẳng hạn như render môi trường rộng và chi tiết. Trong dự án mẫu Megacity thể hiện một môi trường phức tạp được tạo thành từ các entity.
- Dự án có nhiều thành phần động, có thể bao gồm các hành vi tính toán nặng, ví dụ những game RTS thường cần tính toán đường đi cho hàng trăm thậm chí hàng ngàn đơn vị.
- Bạn thích cấu trúc dữ liệu và code của ECS hơn. Cách này được cho là dễ hiểu và dễ bảo trì hơn OOP.
- Bạn đang tạo game multiplayer có tính cạnh tranh với những hành động nhanh như bắn súng, do đó bạn yêu cầu một máy chủ có thẩm quyền và khả năng dự đoán phía máy khách để tạo ra trải nghiệm tốt cho người chơi. (Những tính năng này được hỗ trợ bởi Netcode for Entities còn Netcode for GameObjects thì không).

Hướng dẫn của khoá học này.

Ba hướng dẫn của khoá học này bao gồm các hướng dẫn các sử dụng cơ bản của Entities và job system.

Hướng dẫn đầu tiên, “Bắt đầu với job system của Unity”, trình bày cách sử dụng C# job system, Burst, Unity.Collections và Unity.Mathematics để tạo một giải pháp hiệu năng cao cho một vấn đề liên quan đến tính toán nặng trên CPU.

Hướng dẫn thứ 2, “Học entities với HelloCube”, đi qua các ví dụ vô cùng cơ bản để tạo và sử dụng entities.

Hướng dẫn thứ 3, “Entities hành động: Tanks”, giới thiệu một mô phỏng rất đơn giản về việc di chuyển và bắn đạn của khẩu pháo.

Tài liệu bổ sung.

Để tìm hiểu nhiều hơn về DOTS, tham khảo website.

Bắt đầu với Job System

Mục tiêu

Hướng dẫn này giới thiệu những nguyên tắc cơ bản về cách sử dụng jobs để tận dụng lợi thế của CPU đa nhân.

Kết thúc khóa học này bạn sẽ có thể:

- Tạo, lập lịch và kết thúc jobs “đơn luồng” (single-threaded).
- Tạo, lập lịch và kết thúc jobs song song.
- Lên lịch cho các jobs phụ thuộc các jobs khác.
- Sử dụng NativeArrays.

Tổng quan

Hướng dẫn này giới thiệu những nguyên tắc cơ bản về cách sử dụng jobs để tận dụng lợi thế của CPU đa nhân. Chúng ta sẽ đi từng bước qua mẫu Jobs tutorial trong Entity Component Samples repository trên GitHub.

Trong mẫu này, bạn sẽ bắt đầu với Seekers (cục xanh) và Targets (cục đỏ) mỗi cục di chuyển chậm theo hướng ngẫu nhiên trên mặt phẳng 2D. Công việc của bạn là vẽ đường debug màu trắng từ Seeker tới Target gần nhất theo thời gian thực.

Đầu tiên bạn sẽ được xem cách xử lý vấn đề mà không sử dụng jobs. Sau đó bạn sẽ học cách giải quyết với jobs, và chúng ta sẽ nhìn vào profiles để xem hiệu năng được cải thiện.

Trước khi bắt đầu

Hướng dẫn này giả định rằng bạn có hiểu biết cơ bản về C# và GameObject, và bạn đã hoàn thành hướng dẫn trước đó của khóa học này để làm quen với DOTS và entities.

Set up Project:

1. Tạo project sử dụng mẫu **3D (URP) Template**.
2. Cài package Collections thông qua Package Manager.

Thêm package Collections đồng thời tự động thêm Burst và Mathematics như một ràng buộc

Tạo Seekers và Targets đi lang thang trên mặt phẳng 2D

Đầu tiên, cần định nghĩa các MonoBahaviour đơn giản di chuyển theo hướng ngẫu nhiên. Tham khảo code dưới đây.

1. Tạo script `Obj.cs` và thêm code

```
private Vector3 Direction;
    public float Speed = 100f;
    public float TimeCounter = 0f;

    private void Start()
    {
        Direction = new Vector3(Random.Range(-1f, 1f), 0, Random.Range(-1f, 1f)).normalized;
    }

    public void Update()
    {
        TimeCounter += Time.deltaTime
        if (TimeCounter > 3)
        {
            Direction = new Vector3(Random.Range(-1f, 1f), 0, Random.Range(-1f, 1f)).normalized;
            TimeCounter = 0;
        }
        transform.localPosition += Direction * Time.deltaTime * Speed;
    }
```

2. Tạo script `Seeker.cs` và `Target.cs` kế thừa `Obj.cs`

3. Tạo 2 prefab Seeker và Target chứa component Seeker và Target

```
using UnityEngine;
public class Spawner : MonoBehaviour
{
    public static Transform[] TargetTransforms;
    public static Transform[] SeekerTransforms;
    public GameObject SeekerPrefab;
    public GameObject TargetPrefab;
    public int NumSeekers;
    public int NumTargets;
    public Vector2 Bounds;
    public void Start()
    {
        Random.InitState(123);
        MaterialPropertyBlock seekerPropertyBlock = new();
        seekerPropertyBlock.SetColor("_BaseColor", Color.blue);
        MaterialPropertyBlock targetPropertyBlock = new();
        targetPropertyBlock.SetColor("_BaseColor", Color.red);
        SeekerTransforms = new Transform[NumSeekers];
        for (int i = 0; i < NumSeekers; i++)
        {
            GameObject go = Instantiate(SeekerPrefab);
            SeekerTransforms[i] = go.transform;
            go.transform.localPosition = new Vector3(Random.Range(0, Bounds.x), 0, Random.Range(0,
Bounds.y));
            Renderer renderer = go.GetComponent<Renderer>();
```

```
            renderer.GetPropertyBlock(seekerPropertyBlock);
            renderer.SetPropertyBlock(seekerPropertyBlock);
        }
        TargetTransforms = new Transform[NumTargets];
        for (int i = 0; i < NumTargets; i++)
        {
            GameObject go = Instantiate(TargetPrefab);
            TargetTransforms[i] = go.transform;
            go.transform.localPosition = new Vector3(Random.Range(0, Bounds.x), 0, Random.Range(0,
Bounds.y));
            Renderer renderer = go.GetComponent<Renderer>();
            renderer.SetPropertyBlock(targetPropertyBlock);
        }
    }
}
```

4. Tạo GameObject Spawner. Thêm component Spawner và điền các thông số cần thiết.

Nhấn play và bạn sẽ thấy các khối xanh đỏ đi lang thang nhưng chưa thấy đường debug màu trắng nối Seeker và Target.

Tìm Target gần nhất không sử dụng jobs

Để tìm Target gần nhất cho mỗi Seeker và vẽ đường debug trắng giữa chúng ta cần một MonoBehaviour khác. Để tạo MonoBehaviour này tham khảo phần dưới đây:

1. Tạo script `FindNearest.cs`

```
using UnityEngine;
public class FindNearest : MonoBehaviour
{
    public void Update()
    {
        // Find the nearest Target.
        // When comparing distances, it's cheaper to compare
        // the squares of the distances because doing so
        // avoids computing square roots.
        foreach (var seekerTransform in Spawner.SeekerTransforms)
        {
            Vector3 seekerPos = seekerTransform.localPosition;
            Vector3 nearestTargetPos = default;
            float nearestDistSq = float.MaxValue;
            foreach (var targetTransform in Spawner.TargetTransforms)
            {
                Vector3 offset = targetTransform.localPosition - seekerPos;
                float distSq = offset.sqrMagnitude;
                if (distSq < nearestDistSq)
                {
                    nearestDistSq = distSq;
                    nearestTargetPos = targetTransform.localPosition;
                }
            }
            Debug.DrawLine(seekerPos, nearestTargetPos);
        }
    }
```

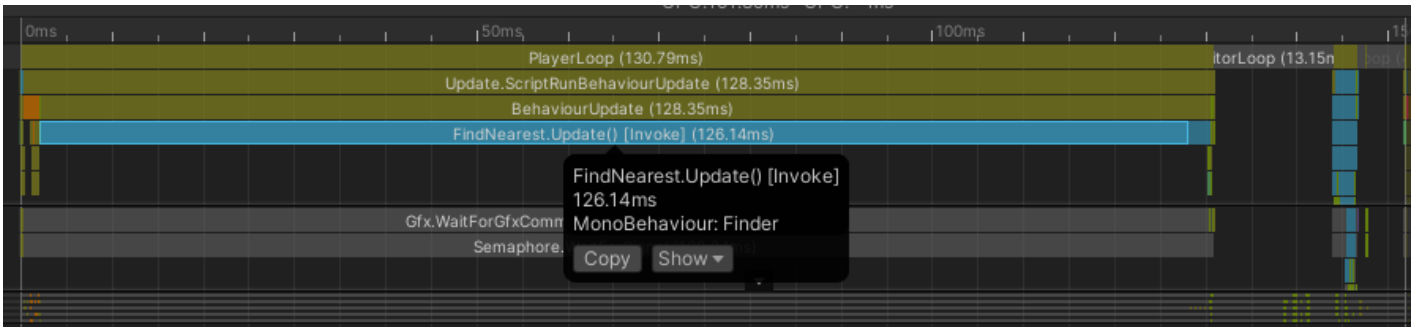
```
}
```

2. Tạo GameObject FindNearest và thêm component FindNearest

Trong Update, với mỗi Seeker. MonoBehaviour kiểm tra khoảng cách với tất cả các Target để tìm ra Target gần nhất và vẽ đường debug.

Note: Để thấy đường debug bạn cần bật **Gizmos** trong **Scene** view hoặc **Game** view

Đây là profile của 1frame chạy với 1000 Seekers và 1000 Targets. Tổng thời gian cho tìm kiếm khoảng **126ms** cho mỗi frame



Rõ ràng đây không phải là một kết quả tốt nhưng k có gì đáng ngạc nhiên khi giải pháp này sử dụng phương pháp N^2 chỉ chạy trên main thread.

Tìm Target gần nhất sử dụng jobs single-threaded

Bằng cách đặt một công việc khó vào job. Bạn có thể di chuyển công việc từ main-thread tới các worker thread. Và bạn có thể Burst-compile code. Jobs và Burst-Compiled code không thể truy cập bất kỳ object được quản lý nào (Bao gồm GameObject và GameObject component), nên trước tiên bạn phải copy tất cả dữ liệu cần xử lý bởi job vào collections không bị quản lý ví dụ như **NativeArrays**.

Note: Nghiêm túc mà nói thì jobs có thể truy cập các object được quản lý nhưng làm như vậy yêu cầu sự quan tâm đặc biệt và đó thường không phải ý tưởng hay. Bên cạnh đó chúng ta muốn Burst-compile job này để tăng tốc và Burst-compile code chắc chắn không thể truy cập object bị quản lý.

Note: Mặc dù chúng ta vẫn có thể sử dụng **Vector3** và **Mathf** nhưng nên thay thế bằng **float3** và **math** từ Mathematics package, đã được tối ưu đặc biệt cho Burst.

Để chuyển sang job, thực hiện các bước sau:

Tạo FindNearestJob.cs

```
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
// We'll use Unity.Mathematics.float3 instead of Vector3,
// and we'll use Unity.Mathematics.math.distancesq instead of Vector3.sqrMagnitude.
using Unity.Mathematics;
// Include the BurstCompile attribute to Burst compile the job.
[BurstCompile]
public struct FindNearestJob : IJob
{
    // All of the data which a job will access should
    // be included in its fields. In this case, the job needs
    // three arrays of float3.
    // Array and collection fields that are only read in
    // the job should be marked with the ReadOnly attribute.
    // Although not strictly necessary in this case, marking data
    // as ReadOnly may allow the job scheduler to safely run
    // more jobs concurrently with each other.
    [ReadOnly] public NativeArray<float3> TargetPositions;
    [ReadOnly] public NativeArray<float3> SeekerPositions;
    // For SeekerPositions[i], we will assign the nearest
    // target position to NearestTargetPositions[i].
    public NativeArray<float3> NearestTargetPositions;
    // 'Execute' is the only method of the IJob interface.
    // When a worker thread executes the job, it calls this method.
    public void Execute()
    {
        // Compute the square distance from each seeker to every target.
        for (int i = 0; i < SeekerPositions.Length; i++)
        {
            float3 seekerPos = SeekerPositions[i];
            float nearestDistSq = float.MaxValue;
            for (int j = 0; j < TargetPositions.Length; j++)
            {
                float3 targetPos = TargetPositions[j];
                float distSq = math.distancesq(seekerPos, targetPos);
                if (distSq < nearestDistSq)
                {
                    nearestDistSq = distSq;
                    NearestTargetPositions[i] = targetPos;
                }
            }
        }
    }
}
```

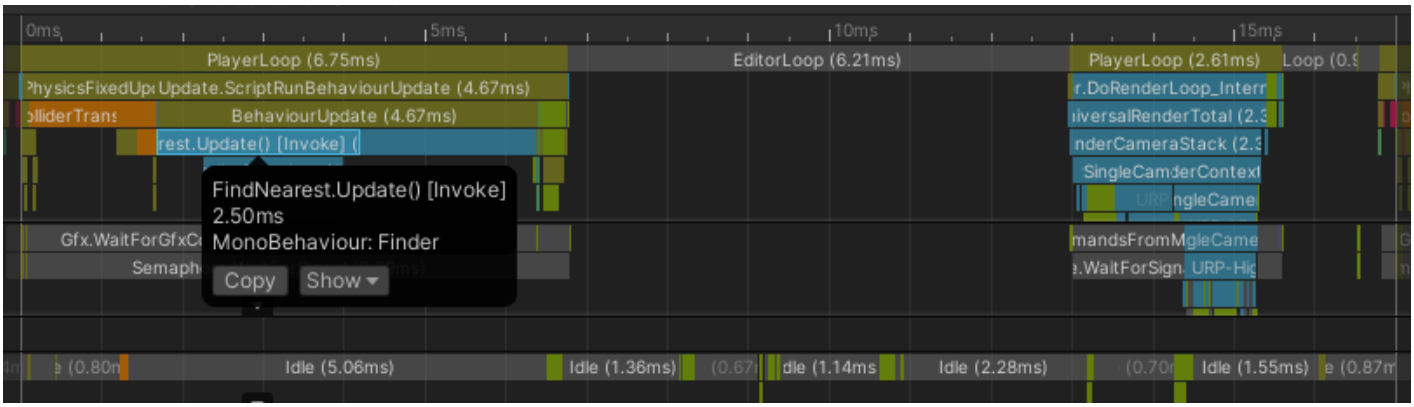
Trong method Execute() vẫn sử dụng logic N2 , nhưng bây giờ vì n chạy trên Burst-compiled job. Job sẽ ăn ít thời gian xử lý của CPU hơn và có thể chạy trên worker thread thay vì main thread.

Để chạy job, khởi tạo và lập lịch cho nó.

```
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;
using UnityEngine;
public class FindNearest : MonoBehaviour
{
    // The size of our arrays does not need to vary, so rather than create
    // new arrays every field, we'll create the arrays in Awake() and store them
    // in these fields.
    NativeArray<float3> TargetPositions;
    NativeArray<float3> SeekerPositions;
    NativeArray<float3> NearestTargetPositions;
```

```
    public void Start()
    {
        Spawner spawner = Object.FindObjectOfType<Spawner>();
        // We use the Persistent allocator because these arrays must
        // exist for the run of the program.
        TargetPositions = new NativeArray<float3>(spawner.NumTargets, Allocator.Persistent);
        SeekerPositions = new NativeArray<float3>(spawner.NumSeekers, Allocator.Persistent);
        NearestTargetPositions = new NativeArray<float3>(spawner.NumSeekers, Allocator.Persistent);
    }
    // We are responsible for disposing of our allocations
    // when we no longer need them.
    public void OnDestroy()
    {
        TargetPositions.Dispose();
        SeekerPositions.Dispose();
        NearestTargetPositions.Dispose();
    }
    public void Update()
    {
        // Copy every target transform to a NativeArray.
        for (int i = 0; i < TargetPositions.Length; i++)
        {
            // Vector3 is implicitly converted to float3
            TargetPositions[i] = Spawner.TargetTransforms[i].localPosition;
        }
        // Copy every seeker transform to a NativeArray.
        for (int i = 0; i < SeekerPositions.Length; i++)
        {
            // Vector3 is implicitly converted to float3
            SeekerPositions[i] = Spawner.SeekerTransforms[i].localPosition;
        }
        // To schedule a job, we first need to create an instance and populate its fields.
        FindNearestJob findJob = new FindNearestJob
        {
            TargetPositions = TargetPositions,
            SeekerPositions = SeekerPositions,
            NearestTargetPositions = NearestTargetPositions,
        };
        // Schedule() puts the job instance on the job queue.
        JobHandle findHandle = findJob.Schedule();
        // The Complete method will not return until the job represented by
        // the handle finishes execution. Effectively, the main thread waits
        // here until the job is done.
        findHandle.Complete();
        // Draw a debug line from each seeker to its nearest target.
        for (int i = 0; i < SeekerPositions.Length; i++)
        {
            // float3 is implicitly converted to Vector3
            Debug.DrawLine(SeekerPositions[i], NearestTargetPositions[i]);
        }
    }
}
```

Nhìn lại profile Update chỉ tốn 2.5ms cho cả quá trình với Burst-Compilation bật.



Lưu ý rằng với cách này các jobs đều chạy trên main thread. Điều này có thể xảy ra khi gọi Complete() một job chưa được rút ra khỏi hàng đợi. Bởi vì main thread sẽ ở trạng thái không hoạt động khi chờ job kết thúc nên main thread có thể chạy job đó.

Thêm một lưu ý là thời gian để update cho Seeker và Target MonoBeviour cũng rất quan trọng, đặc biệt khi tăng số lượng Seeker và Target. Đây chỉ yếu là chi phí để cập nhật riêng lẻ cho từng MonoBehaviour. Giải pháp tốt nhất là cập nhật tất cả trong 1 update (Tốt hơn nữa là sử dụng Entities để thay thế GameObject, mặc dù điều này đi quá mục tiêu của hướng dẫn này).