DOTS in Unity

Mikesch Severin
BIT19
Fachhochschule Wiener Neustadt
Wiener Neustadt, Lower Austria
Severin.Mikesch@fhwn.ac.at

Giessrigl Christian
BIT19
Fachhochschule Wiener Neustadt
Wiener Neustadt, Lower Austria
Christian.Giessrigl@fhwn.ac.at

I. Introduction

Unity is a game engine which comes with a built-in Integrated Development Environment (IDE). The difficulty of building 2D and 3D games will be drastically reduced by working with a pre-build engine like unity. Also unity is a freeware if you do not earn $100.000 a year with it and the game application program code is mostly written in C#.

II. Unity´s Past: The Component System

When unity was first launched, everything was constructed around GameObjects and Components. You want to add gravity to a GameObject? Just add a Rigidbody component! Want to emit light? Add a light component! Want to add a custom script to a GameObject? MonoBehaviour component!

Although it is a very natural way to think of objects in a game world, it doesn't scale very well when growing the game worlds.For example, each GameObject was put into a global list of GameObjects which had to be completely locked in order to add to or remove from it. Furthermore, this was a massive hit in performance as this list spread randomly in memory which lead to an increased amount of cache misses.

Unity´s old component system called periodic updates on it´s components (also called the Update loop). As the global list of GameObjects was unordered, there was no predictable pattern these components would be called and so the code had to be loaded and dumped from the cache over and over again.

This (unordered) data layout leads to many problems in the mobile oriented and multi-core driven gaming market. Introducing Unity DOTS...

III. What is DOTS?

With unity's new initiative of "performance by default" comes the Data-Oriented Technology Stack that offers developers new possibilities when working with unity. DOTS is a new system that adds the concept of data oriented design in form of multiple add-in packages to unity. The 3 main packages are:

1) The Entity Component System (ECS)
2) The C# Job System
3) The Burst Compiler

These three packages work together to make unity an even better workspace for developers and will be explained further in this document. There are also ambitions to improve loading times and shrink file sizes with "Projekt Tiny" - a unity package collection built on DOTS. This will lead to faster loading times especially in mobile and web applications.

A. Entity Component System

The entity component system of unity has three important parts to understand the big picture of this architecture.

1) The Entity: The first part is the entity itself. An entity is a reference to an object in your game, no matter if it can interact with something or can be interacted with or is just decoration - so it is comparable to an GameObject. But the difference is that those entities are grouped together in blocks of so called "Archetypes". Archetypes are groups of entities that share the same combination of components, which are explained in the next paragraph.

2) The Component: This data is stored in the so called components, the second part of this system. A Component is a named container of a small part of an object's data. Every object can hold multiple components and all objects can use the same named component but storing it´s own values, but you only have to create a component once and distribute it to the objects of your choice.

For example an entity "bird" has the component "wings" and the value of the amount of wings would be 2. Also another entity "butterfly" could have the same component "wings" but the value of the amount of wings would be 4.

3) The System: The third and last part is named a system. You can imagine a system as the outsourced Update-method(s) of a MonoBehaviour. The different systems transform or work with the components of an object. A specific system can only handle an object's data if the object has the right set of components.

For example the entity "bird" has the component "fledged" and therefore the system "fly" can take the bird's component and modify it's z-axis position. Also an entity "cow" does not have the component "fledged" and in the following the system "fly" will not be working with "cow".

Every component / system - process will be handled as a "job" in the run-time and be called every frame. A job is a small unit of work that does one specific task. A job receives parameters and operates on data, similar to how a method call behaves. Jobs can be self-contained, or they can depend on other jobs to complete before they can run.

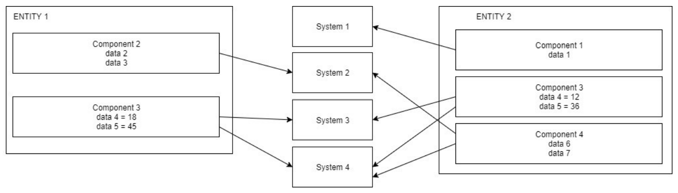

Fig. 1. The Entity Component System

ECS can run alongside the existing system of individually called MonoBehaviour-Components (Hybrid ECS) or can be used exclusively (Pure ECS).

B. The C# Job System

The C# job system manages a group of worker threads across multiple cores. It usually has one worker thread per logical CPU core, to avoid context switching. User-written code and Unity share those worker threads. This cooperation avoids creating more threads than CPU cores, which would cause conflicts for CPU resources.

The C# job system is designed to make unity a multi-threaded environment. Not only does it boost the performance of a game, it can also work off complicated scenes with a ton of moving game objects fluently.

Multi-threaded code offers better performance to the developer but can be difficult to write. Although situations demanding the C# job system may differ, the performance boost someone can get is extremely high. The Entity Component System, can also be utilized to further increase performance. Many tasks with a heavy load on the CPU can be lightened thanks to C# jobs.

To get an overview of how the ECS and the Job System work together and how their architecture looks like, look at Figure 1.

C. The Burst Compiler

The Unity Burst Compiler converts your written code into highly optimized machine code. It is especially efficient when it comes to optimizing C# Jobs. The Burst Compiler gives small teams and indie developers the opportunity to achieve a high level of performance for their unity games. Normally you would need to work in a wealthy company to get to this level of performance. All you need to do is import the Unity Burst Package to your project, leverage the Unity Job System, and Unity takes care of the rest.

IV. Difference Old System vs New System

Not only the multi threading is a huge advantage compared to the old system but also the managing of the systems mentioned before. Instead of having an update loop, there are now archetypes of entities. An archetype is a specific combination of components the entities have. Now a loop of operations runs over all entities of the same archetype to read and write the component's data. Adding or removing components results in the entity being transferred to a different archetype's collection of entities. Also you declare in systems if the data of a component is used read only or read/write. Therefore all jobs that only want to read do not have to wait on other read only jobs and the ECS takes a huge advantage out of that by sticking them together in the runtime.

V. Classic Unity Code

A. Script that moves it´s GameObject towards a random location

Needs to be attached to a GameObject that should move towards its destination.

```
using UnityEngine;

// every script that needs to be a component
// inherits from MonoBehavior
public class MoverBehaviour : MonoBehaviour
{
    // the position in the 3D space
    // the GameObject moves towards.
    public Vector3 destination;
    // the velocity of the Gameobjects movement
    // towards the destination
    public float speed = 5;

    // will be called automatically
    // in every frame iteration
    void Update()
    {
        // the distance between the Gameobjects
        // position and the destination position.
        float distance = Vector3.Distance(
        destination, transform.position);

        // if the distance is short enough it
        // has practically reached the destination
        if (distance < 0.1f)
        {
            // choose a new destination point in a
            // sphere of 50 units around the origin (0, 0, 0)
            destination = Random.insideUnitSphere * 50;
        }
        // if the gameobject has not reached
        // the destination yet
        else
        {
            // move the gameobject closer to the
            // destination based on the time difference
            // of now and the last update
            transform.position +=
            (destination - transform.position).normalized
            * Time.deltaTime * speed;
}}}
```

B. A script that spawns a GameObject multiple times

Needs to be attached to a GameObject in order to be executed.

```
using UnityEngine;

// creates the amount of GameObjects
public class Spawner : MonoBehaviour
{
    // the game object which will be spawned
    // must be set in the inspector
    public GameObject spawnableObject;

    // the amount of gameobjects that should
    // be spawned by the Spawner
    public int amount = 50000;

    // the range in which the objects can
    // spawn from the spawners position
    public float range;

    // will be called automatically when
    // the game starts
    void Start()
    {
        for (int i = 0; i < amount; i++)
        {
            // the spawn position of a gameobject
            Vector3 pos = Random.insideUnitSphere * range;

            // creates a gameobject at the specified
            // position with a specified rotation
            Instantiate(
            spawnableObject, pos, Quaternion.identity);
}}}
```

VI. DOTS Unity Code

A. The Entity´s Data

Needs to be attached to the GameObject prefab that should be moved.

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

// Tells unity to generate a MonoBehaviour with these fields
// ... in order to add it to the inspector.
[GenerateAuthoringComponent]
// contains a part of the data an entity holds
public struct MovableDots : IComponentData
{
    // the destination of the entity
    public float3 destination;
    // the speed the entity moves towards the detination
    public float speed;
}
```

B. Creating a manager that creates an entity from a prefab and multiplies this entity

This MonoBehaviour needs to be attached to a GameObject as it will not be executed otherwise.

```
using UnityEngine;
using Unity.Entities;
using Unity.Transforms;

// normal MonoBehavior that creates the entities
public class SpawnerDots : MonoBehaviour
{
    // the prefab that should be converted into an entity
    // must be set in the inspector
    public GameObject movablePrefab;
    // the from the prefavb created entity
    public Entity movableEntity;
    // the entity manager instance used by the spawner
    public EntityManager entityManager;

    void Start()
    {
        // Get the default EntityManager for the world
        entityManager =
        World.DefaultGameObjectInjectionWorld.
        EntityManager;

        // Get settings for converting the
        // Game Objects into Entities
        var settings =
        GameObjectConversionSettings.FromWorld(
        World.DefaultGameObjectInjectionWorld, null);

        // Create the entity
        movableEntity =
        GameObjectConversionUtility.
        ConvertGameObjectHierarchy(
        movablePrefab, settings);

        // Spawn the entities
        for (int i = 0; i < 50000; i++)
        {
            entityManager.Instantiate(movableEntity);
}}}
```

C. Iterating over all entities with the specified Component

This script doesn´t need to be attached to a GameComponent as it will be used automatically.

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

// component system provides multiple virtual methods to use
public class MovableSystem : ComponentSystem
{
    // is called every frame
    protected override void OnUpdate()
    {
        // iterates over all entities that have ...
        // ... a MovableDots component on them.
        Entities.WithAll<MovableDots>().
        ForEach((
            // requires the translation component
            ref Translation trans,
            // requires the MovableDots component
            ref MovableDots movable) =>
        {
            // setting the speed
            movable.speed = 3f;

            // if the distance to the
            // destination is small enough...
            if(Vector3.Distance(
                new Vector3(
                trans.Value.x, trans.Value.y, trans.Value.z),
                new Vector3(
                movable.destination.x,
                movable.destination.y, movable.destination.z)
                ) < 0.1f)
            {
                // ... then set a new random location
                movable.destination = new float3(
                    UnityEngine.Random.Range(-20f, 20f),
                    UnityEngine.Random.Range(-20f, 20f),
                    UnityEngine.Random.Range(-20f, 20f));
            }

            // set the position value accordingly
            trans.Value +=
            (movable.destination - trans.Value)
            * Time.DeltaTime * movable.speed;
        });
}}
```

VII. The Result

Testing both codes with an amount of 50.000 spheres flying around the classic code has spent approximately 200 milliseconds for each frame to update and the program had a time lag. The DOTS code has reduced this amount by half (around 100 milliseconds) and the spheres have been flying more smoothly.