Entity Componenet System for Unity

Unity®

Codron Julien – Master 2 ILJ 2021-2022

Table des matières

Introduction

Unity est connu pour avoir parfois des problèmes de performance. Notamment lorsqu'un grand nombre de gameObject sont présent dans une scène. Il a souvent été comparé sur ce point, à l'Unreal Engine, son principal concurrent. En effet, sur Unity, tous les GameObject héritent du même objet qu'est le monoBehaviours, permettant de leurs donné toute les fonctionnalités de base Comme par exemple son rendu, sa physique ou encore ses mouvements. Ce qui fait que chaque objet est composé d'élément dont parfois il ne se servira jamais.

Une équipe d'Unity a donc commencé à travailler sur le modèle Entity Component Système en 2018 afin de palier à ce problème de performance. Ce système. Il est pour le moment toujours en développement et il n'est donc pas conseiller de l'utiliser en production, mais plutôt d'attendre alors la sortie d'une version stable, la 1.0. Le développement du modèle

Le modèle ECS manière alternative de développer avec le moteur de jeu Unity. Celui permet de grandement augmenter les performances des jeux développé avec ce moteur. Avec un code bien plus performent, notamment pour le multithreading, mais également plus simple à lire et très facile à réutiliser. Il permet également l'utilisation du Burst Compiler. Un compilateur C# produisant du code extrêmement optimisé.

Son design est orienté vers les datas, et non directement vers l'objet. La manière de coder un objet n'est donc pas la même, et il nécessaire d'apprendre afin de pouvoir utilisé le modèle ECS. Il permet donc d'alléger les GameObject en les construisant d'une manière différente, et en évitant qu'il possède des données et fonction inutile.

Développer avec ce modèle permet donc d'être beaucoup plus libre par la suite dans le développement d'un jeu et son gameplay. D'être plus libre et moins limité par les contraintes techniques du moteur de jeu. Je vais donc procéder à une présentation du modèle et de son fonctionnement. Puis réaliser une preuve de concept avec un projet Unity afin de comparer les performances entre des objets construit de manière « traditionnel » avec Unity. Et ces mêmes objets, mais cette fois avec le modèle ECS.

Modèle ECS

Organisation du code

Commençons par un rappelle sur la façon traditionnelle de créer un objet avec Unity, ici un Player. Pour cela on va créer un GameObjet que l'on nommera player auquel on va ajouter des components (des MonoBehaviours) permettant de lui donner des fonctionnalités tel que le rendu, la physique ou encore les mouvements.

Avec modèle ECS nous allons devoir séparer notre code en 3 éléments, 3 parties :

- **Entities** : Leurs fonctionnement va se rapprocher de celui des GameObject, mais sera bien plus léger en terme de ressources. On va donc y regrouper les **Components** de l'objet.

- **Components** : c'est la partie du code qui contient les données de l'objet. Il n'y a aucune logique entre les data dans cette partie. Il s'agit juste de container pour les données que l'on va lier à une **Entité**. Si 2 **Entités** possèdent exactement les mêmes components on dit qu'elles ont le même **Archetype**.

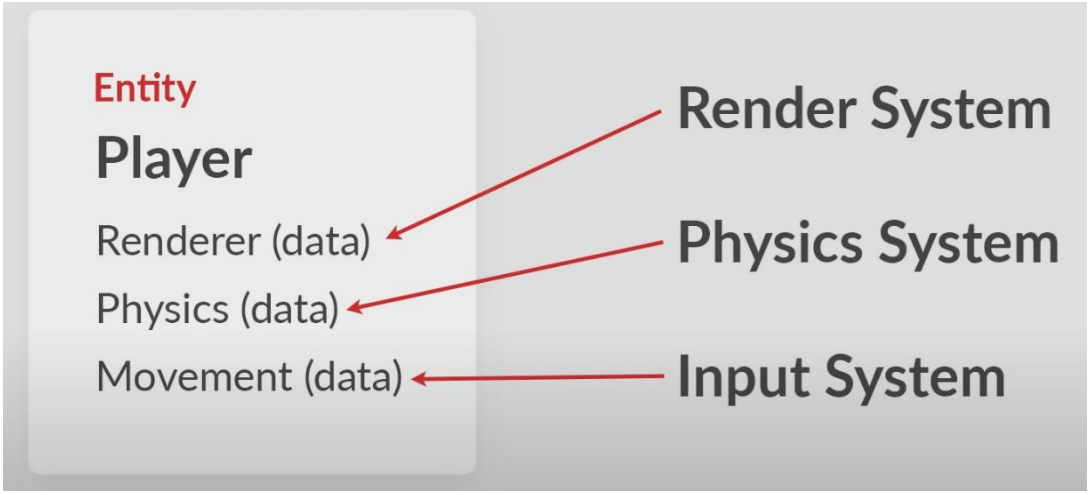
- **Systems** : C'est ici qu'est gérée la partie logique de l'objet, son comportement. Un système va être appliqué à toute les **Entités** qui possèdent les **Composents** spécifique désigné par le système.

Reprenons notre exemple précédent. Ici pour créer un player avec le model ECS.

Nous allons tout d'abord créer une **Entité** Player.

Puis ses **Composants** sous forme de donnée, ici pour son rendu, sa physique et ses mouvements. L'on rappelle que la seule fonction de ces components est de stocker les données et qu'ils n'effectuent aucune action.

Enfin, pour donner des fonctionnalités à notre objet Player, nous allons créer des **Systemes**, ici Render system, Physics system, Input system. Attention, on rappelle que toutes les futurs **Entités** que l'on va créer serons affectée par exemple, par le Render Système, si elle possède le **Composant** correspondant au système, ici le **Composant** pour le rendu.



Voici donc la théorie, nous allons maintenant voir comment implémenter ce système avec Unity. Il existe deux méthode afin d'y parvenir. Une permettant d'exploiter toute la puissance du modèle ECS, mais difficile à mettre en place. Et une seconde bien plus simple à réaliser et permettant de mettre tout de même d'utiliser le modèle ECS.

Pure ECS vs Hybrid ECS

Alors qu'est-ce que tout ça change au niveau code. Pour utiliser les ECS, unity nous propose 2 possibilités.

La première est **Pure ECS**, avec celle-ci, on n'utilise plus du tous les GameObjects, ni les MonoBehaviours. Ce qui peut être très, très contraignant et demande de changer totalement sa façon de développer avec Unity. En effet, on ne peut par exemple plus du tout utilisé de GameObject dans l'éditeur d'unity. En contrepartie bien évidement, on a accès à toute la puissance de ce modèle ECS.

Bien heureusement pour nous Unity a mis à disposition une seconde méthode, **Hybrid ECS**. Elle permet d'utiliser des gameObject et de les convertir en **entités** grâce à un simple scripte. Créer son objet comme un gameObject habituel, et y ajouter un script -> DOTS -> Convert to entity. Dans conversion mode sélectionnez Convert And Destroy, ce qui permet à en runtime de détruire le gameobject et de le remplacer par une **entité**.

Dans un script standard, on gère les données (datas) ainsi que le comportement (Behaviour) de l'objet. Ici les données seront toujours dans ce script. Mais les comportements eux seront gérée au niveau du **Systeme** (qui l'appliquera donc à tous les GameObject possédants ces **Components**, c'est-à-dire ce script) comme vue précédemment.

Bien évidement on ne bénéficie pas de toute la puissance de Pure ECS avec cette méthode mais elle permet une transition beaucoup plus simple pour passer au ECS.

Evolutions du modèle ECS

Création

Comme je vous l'ai introduit le modèle ECS fut créer par une équipe d'Unity en 2018. Dont développement a connu plusieurs rebondissements. Depuis sa création, des mis à jour régulière était faite chaque mois, jusqu'en novembre 2020 avec version 0.17. Puis plus aucune nouvelle. Certains pensaient que le projet était abandonné, mais un développeur à confirmer qu'il était toujours en cours de développement. Et qu'ils ont très mal communiquer dessus.

Le 16 mars dernier la version 0.50 sort. Cette maj expérimentale met beaucoup de chose à jours, en effet elle est sortie 1ans ½ après la dernière. On peut y trouver dans les grandes lignes, une amélioration de l'interface de débug EntityManager remplacée par différentes fenêtres, une meilleure intégration avec l'éditeur Unity. Une Hierarchy window pour les entités, Archétypes window, Component, Systeme et Entities window. Beaucoup d'amélioration dans l'éditeur Unity.

Pour tous les projets utilisant la version 0.17, il est nécessaire d'en faire un upgrade manuel pour qu'ils soit compatible avec la 0.50. Les développeurs ont également indiqué que ce serait surement le cas lors de la release de la 1.0. Beaucoup de signatures/noms de fonctions changent ou de fonctionnalités entre chaque mise à jour.

Pendant cette période le modèle ECS à rejoins un ensemble de technologie visant à l'augmentation des performances des logiciels/jeux développer avec Unity.
Le Data-Oriented technology stack ou DOTS.

Data-Oriented Technology Stack_DOTS

L'objectif de Unity's Data-Oriented Technology Stack (DOTS) de Unity est de vous permettre de créer des jeux plus ambitieux avec Unity, afin de répondre à vos besoins dans le sens de jeux multijoueur complexes, de crossplay et de monde ouvert.

DOTS inclut tous les packages basés sur l'architecture Entity-Component-System (ECS). La notion de DOTS est souvent étendue pour inclure le compilateur Burst et le C# Job System, deux technologies qui ont été créées pour soutenir le développement d'ECS mais qui en sont indépendantes.

Le compilateur Burst et C# Job System sont déjà recommandés pour la production et peuvent être utilisés dans n'importe quel projet Unity, qu'ECS soit utilisé ou non. Les API permettant d'implémenter le modèle d'architecture logicielle ECS dans un projet Unity sont livrées dans le package nommé Entities. Pour bénéficier des avantages de l'architecture orientée données, le code et les packages du jeu doivent être basés sur des entités.

Le compilateur Burst traduit du bytecode .Net en code natif hautement optimisé à l'aide de LLVM (module de compilation), il est publié en tant que package Unity et accessible dans celui-ci via le gestionnaire de package.

Quant à lui le C# Job System permet d'écrire simplement, et de manière sécurisé du code multithreadé avec unity, afin d'utiliser toute les ressources disponibles et d'optimiser les performances.

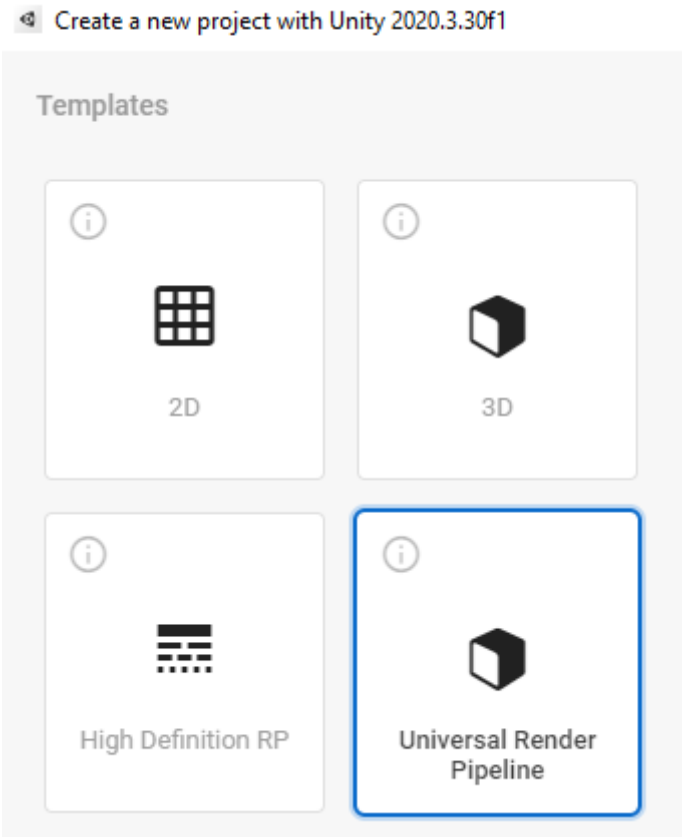
Preuve de concept

Objectifs

Dans un projet Unity, j'ai pour objectif de générer un nombre important de gameObject, de simple cube. De leurs donner des comportements tel que se mouvoir ou encore de changer leur couleur. Puis de faire la même chose mais en utilisant le modèle ECS. Pour ainsi observer les différences de performance entre les deux systèmes.

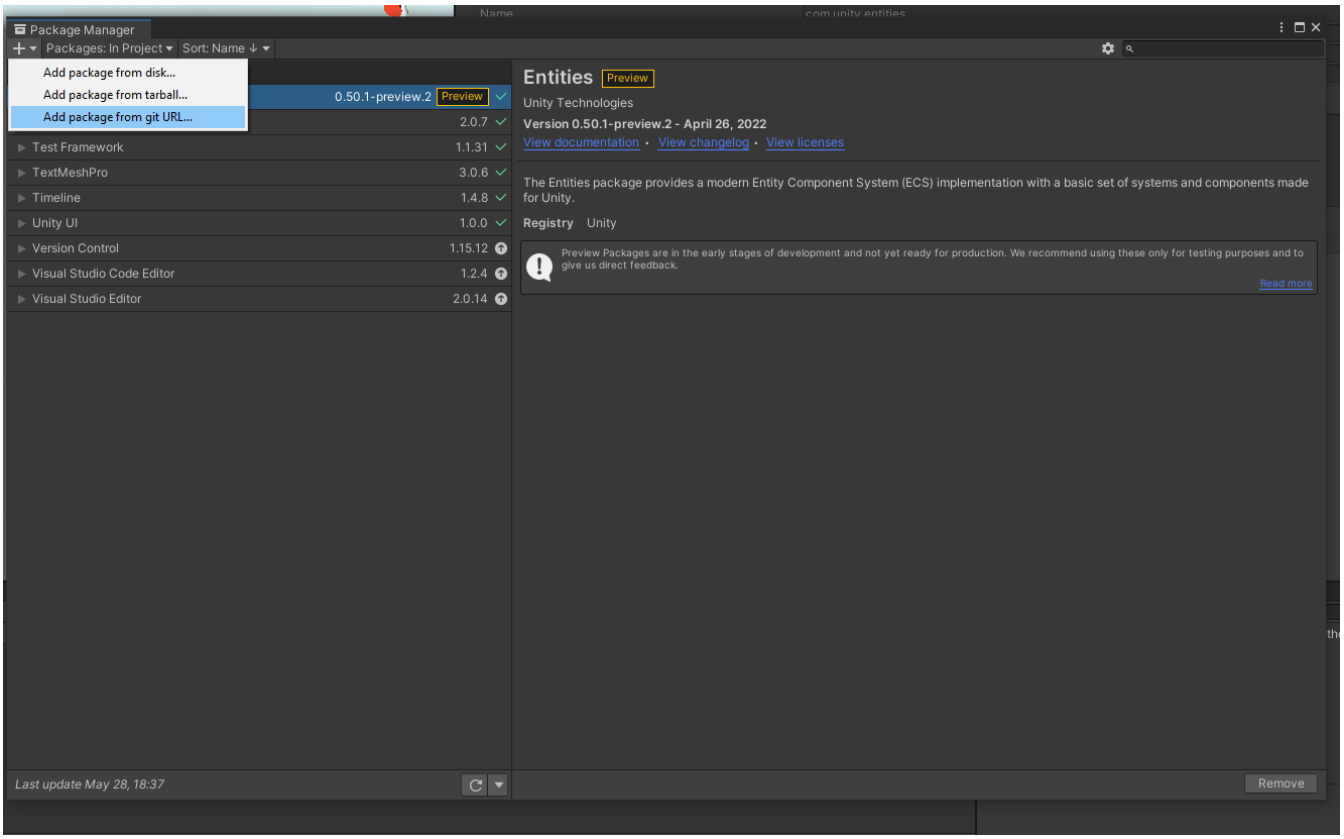
Installation

Il est conseillé pour utiliser ECS 0.50 d'utilisé la version 2020.3.30 d'unity, ce que nous allons alors faire. Il faut donc créer un project Unity, attention pas en 3D mais avec l'URP, Universal Render Pipeline.



Ensuite télécharger les packages nécessaire grâce au package manager :

Pour installer le package « entities » qui est en preview, il faut faire « add package from git URL puis entrer « com.unity.rendering.hybrid ». Celui ajoutera tous les packages nécessaire pour l'utilisation du modèle ECS .



Il est également nécessaire, pour coder avec la dernière version du modèle ECS d'utiliser l'un des IDE suivant, qui eux seul le supportent :

- Visual Studio 2022 +

-Rider 2021.3.3+

Création des objets :

Nous allons dans un premier temps créer 2 scène. L'une pour faire apparaitre des gameObject de manière traditionnel. Et l'autre avec le modèle ECS.
L'objectif est de faire massivement spawn un objet et de le mettre à jour avec différente action. J'ai décidé d'effectuer deux interactions simple qui sont le changement de couleurs de l'objet, et son changement de position. Nous utiliserons ici l'implémentation hybride pour des raisons de compréhension du code. Il nous faut alors pour cela créer 2 prefabs, sur lequel nous ajoutons une mesh(d'un cube), une mesh renderer, un rigidbody, un boxCollider, et pour finir bien évidement une material pour pouvoir changer la couleur du cube.

Pour le premier préfab fait de manière « traditionnel » nous ajoutons 2 script permettant de changer la couleur et la position du cube celons un « ModeType » que l'on sélectionnera. Et un dernier script permettant de sauvegarder ses informations.

Pour le second préfab réalisé avec le modèle ECS hybrid, nous allons également ajouter 2 scripts, qui seront les **Composants** de notre **Entité**. Un premier pour la position et un second pour la couleur. Voici à quoi ressemble l'un d'en eu.

```
using Unity.Entities;
using UnityEngine;

3 références
struct ColorAnimated : IComponentData
{
}

namespace Authoring
{
    [DisallowMultipleComponent]
    Script Unity (1 référence de ressource) | 0 références
    public class ColorAnimated : MonoBehaviour, IConvertGameObjectToEntity
    {
        0 références
        public void Convert(Entity entity, EntityManager dstManager, GameObjectConversionSystem conversionSystem)
        {
            dstManager.AddComponent(entity, ComponentType.ReadWrite<global::ColorAnimated>());
        }
    }
}
```

Nous ajoutons également un script Material Color, permettant de changer la couleur du matériel avec le model ECS. De manière traditionnel, nous avons dans les scripts correspondant la fonction Update qui met à jour la couleur et position de nos cubes. Ici nous avons seulement créer les **Composants**. Il nous faut alors Comme vous le savez maintenant pour gérer les actions de notre object, avoir besoin de système. Nous pouvons créer un système que nous appellerons StressTestA nimationSystem.

Qui mettra à jour la position et la couleur de nos cubes. Nous l'implémentons alors de la manière suivante :

```
using Unity.Entities;
using Unity.Mathematics;
using Unity.Rendering;
using Unity.Transforms;
using static Unity.Mathematics.math;

0 références
public partial class StressTestAnimationSystem : SystemBase
{
    2 références
    void AnimateColors(SimulationMode.Mode mode)
    {
        Entities.WithAll<ColorAnimated>()
            .ForEach((ref MaterialColor color, in SpawnIndex index) =>
            {
                float indexAdd = 0.123f * (float)index.Value;
                float time = mode.time * 5.0f;

                var v = new float3(math.cos(time + indexAdd), math.sin(time + indexAdd), 0.0f);
                color.Value = float4(v * 0.4f + 0.5f, 0);
            }).ScheduleParallel();

        // Color animation of LOD children, they don't have SpawnIndex
        Entities.WithAll<ColorAnimated>()
            .WithNone<SpawnIndex>()
            .ForEach((ref MaterialColor color) =>
            {
                float indexAdd = 4.56f;
                float time = mode.time * 5.0f;

                var v = new float3(math.cos(time + indexAdd), math.sin(time + indexAdd), 0.0f);
                color.Value = float4(v * 0.4f + 0.5f, 0);
            }).ScheduleParallel();
    }

    2 références
    void AnimatePositions(SimulationMode.Mode mode)...

    0 références
    protected override void OnUpdate()
    {
        var mode = SimulationMode.getCurrentMode();

        switch (mode.type)
        {
            case SimulationMode.ModeType.Color:
                AnimateColors(mode);
                break;
            case SimulationMode.ModeType.Position:
                AnimatePositions(mode);
                break;
            case SimulationMode.ModeType.PositionAndColor:
                AnimateColors(mode);
                AnimatePositions(mode);
                break;
            default:
                break;
        }
    }
}
```

On hérite de la classe SystemBase, permettant dans lancer ce système en RunTime. Pas besoin qu'il soit présent sur un gameObject. On peut voir ici avec le « Entities.WithAll<ColorAnimated>() qu'on applique cette fonction, ce système à tous les objets possédant le **Composant** ColorAnimated. On fait de meme pour la position.

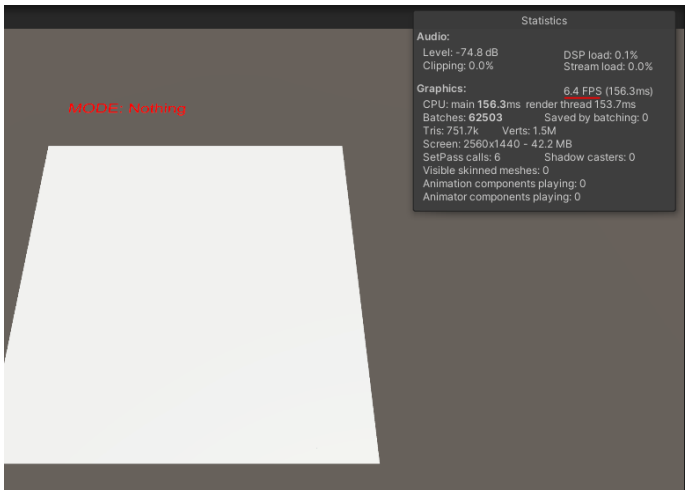
Nous allons maintenant créer les objet/entité « spawner » permettant de faire apparaitre nombre de ces prefabs. Il suffit donc pour la manière habituelle de faire un script avec une double boucle qui fait spawn les GameObject à une distance égale à leur taille l'une de l'autre. Ce script se nomme SpawnGameObject et est attaché au GameObject Spawner.

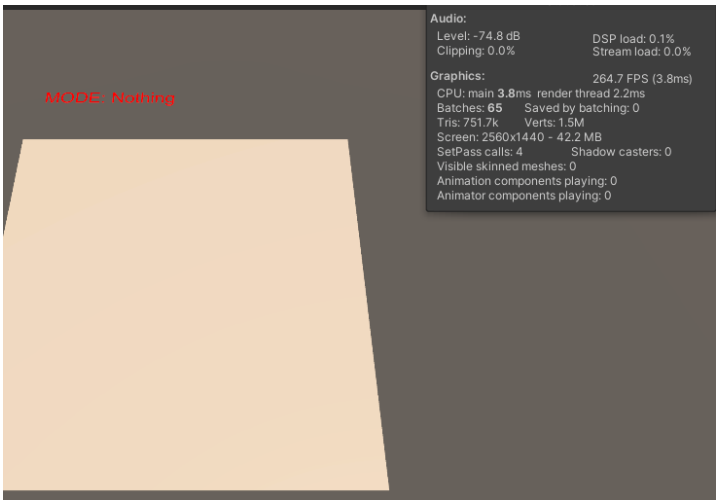
Pour le modèle ECS nous allons pouvoir faire de la même manière, car il est possible de transformer un prefab en **Entité**, et de le faire apparaitre. Nous allons donc créer un **composant** à notre spawner auquel nous allons ajouter un **système** permettant de faire apparaître les **entités**. Ce système est similaire au script SpawnGameObject, mais à la place on spawn des **entités**.

On ajoutes dernier script contentant les différent mode (changement de couelurs, position, aucun ou les deux) que l'on met à jour toute les x seconde. Afin de réaliser les actions souhaiter pour le test.

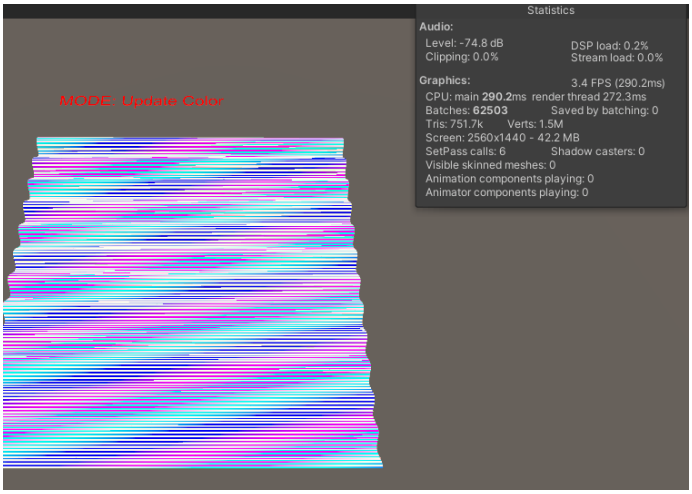
Observation :

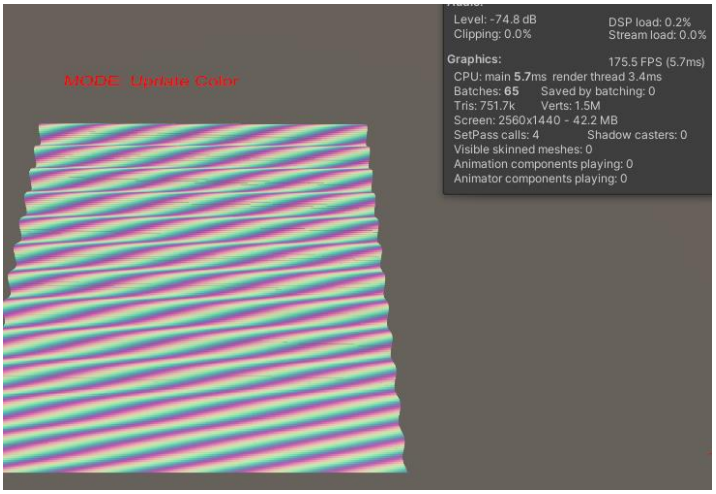
Aucune action :



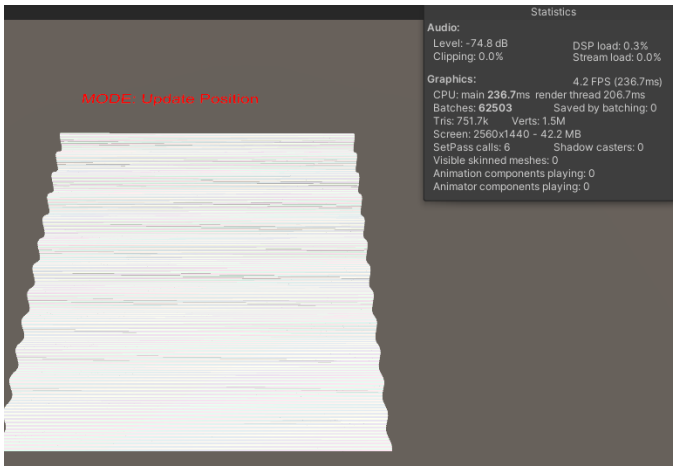


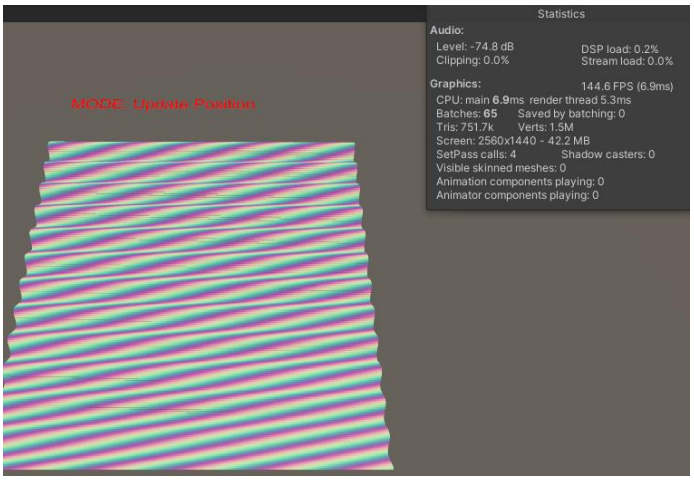
Changement de couleur :



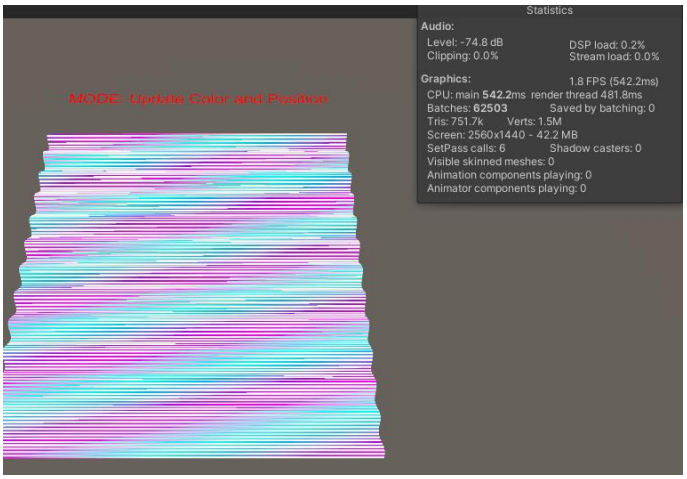


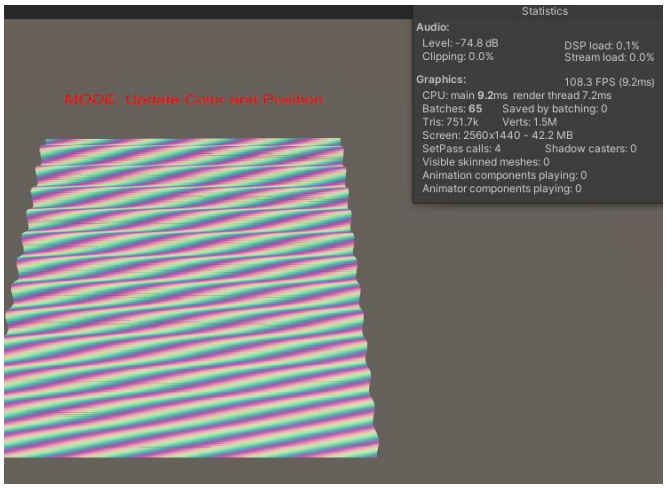
Changement de position :





Changement de couleur et de position :





Conclusion

Nous avons pu observer la flagrante différence de performance avec le système traditionnel. Grace au système ECS, mais également au C# job système permettant un système de multithreading. Bien qu'il soit difficile à manipuler pour le moment, et loin d'être aboutie. Une roadmap a été créer pour le système DOTS avec la dernière mis à jour. Il est prévu pour les développeurs d'implémenter plus d'outils pour l'éditeur, d'outils de débug. Mais également une meilleure compatibilité avec les gameObjects, un code natif performent, une simulation de la physique à grande échelle. Les mis à jour planifier pour le moment sont des serveur DOTS ainsi que des animations basées sur le système ECS. Et bien d'autre features. L'évolution de ce modèle est à suivre, il s'agit surement de l'avenir d'Unity. Grâce à lui, des projets bien plus ambitieux pourrons voir le jours grâce au moteur de jeu Unity.

Sources :

Status DOTS : Unity - DOTS Development Status And Next Milestones - March 2022 - Unity Forum

roadmap DOTS : DOTS Roadmap | Unity |

git ECS : EntityComponentSystemSamples/HybridHDRPSamples at master · Unity-Technologies/EntityComponentSystemSamples · GitHub

guide installation : Entities installation and setup | Entities | 0.50.0-preview.24 (unity3d.com)

doc unity : Entities overview | Entities | 0.50.0-preview.24 (unity3d.com)

good Youtube channel about ECS : Turbo Makes Games - YouTube