LOG8715- Architecture de jeu vidéo

Rapport TP2 :

Optimisation

Présenté à

Samuel Bellomo

&

Keven Chaussé

Réalisé par

Équipe 4

Emmanuel Doré-Walsh	1954113
Alex Hua	1994253
Vincent Beiglig	2161304

Exercices 1

Pour optimiser la disposition en mémoire des structures UnoptimizedStruct1 et UnoptimizedStruct2, nous avons réordonné les déclarations des variables de la manière présentée dans la figure 1.

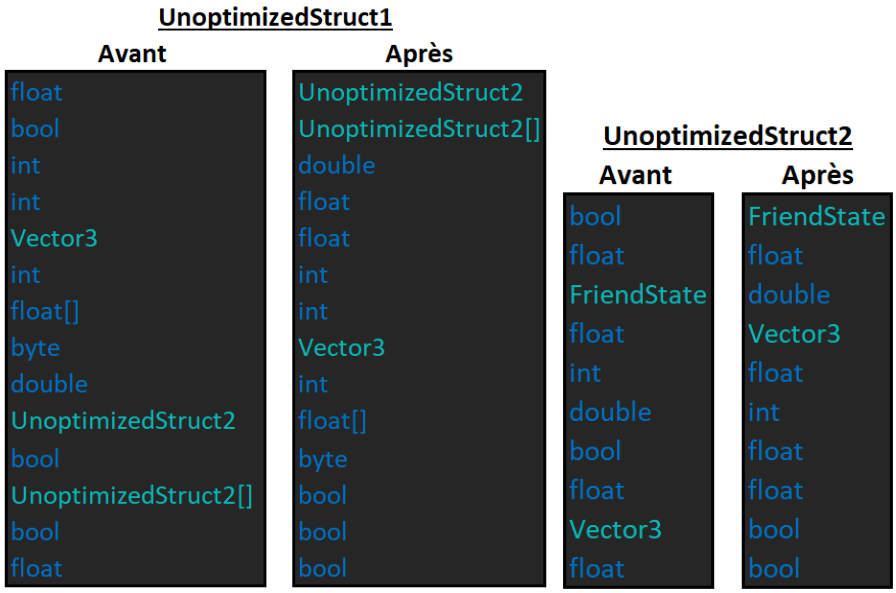


Figure 1 : modification de l'ordre de déclaration des variables de UnoptimizedStruct1 et UnoptimizedStruct2

En regroupant les variables bool et byte à la fin des déclarations, on réduit la perte de mémoire dû au padding. Le réordonnement a permis de passer de 32.4 MB à 28.2 MB pour les 100 000 de UnoptimizedStruct2 et 13.0 MB à 10.7 MB pour UnoptimizedStruct1 tel qu'illustré à la figure 2 et 3.

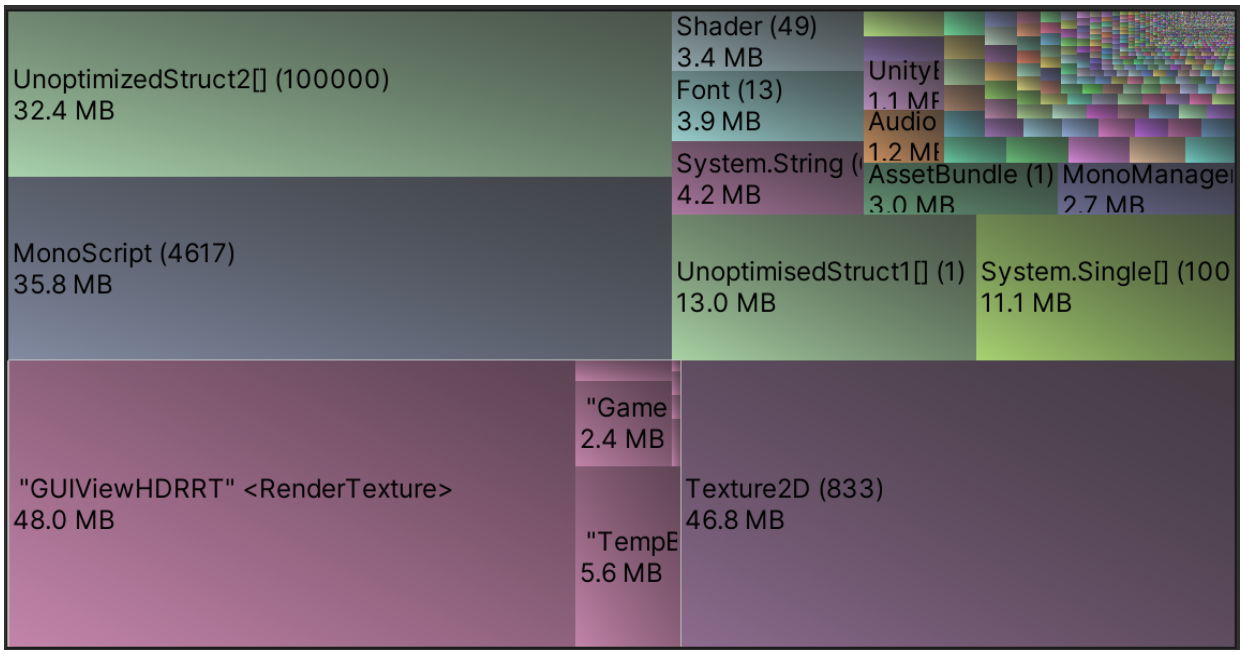


Figure 2 : treemap de la disposition en mémoire avant les modifications

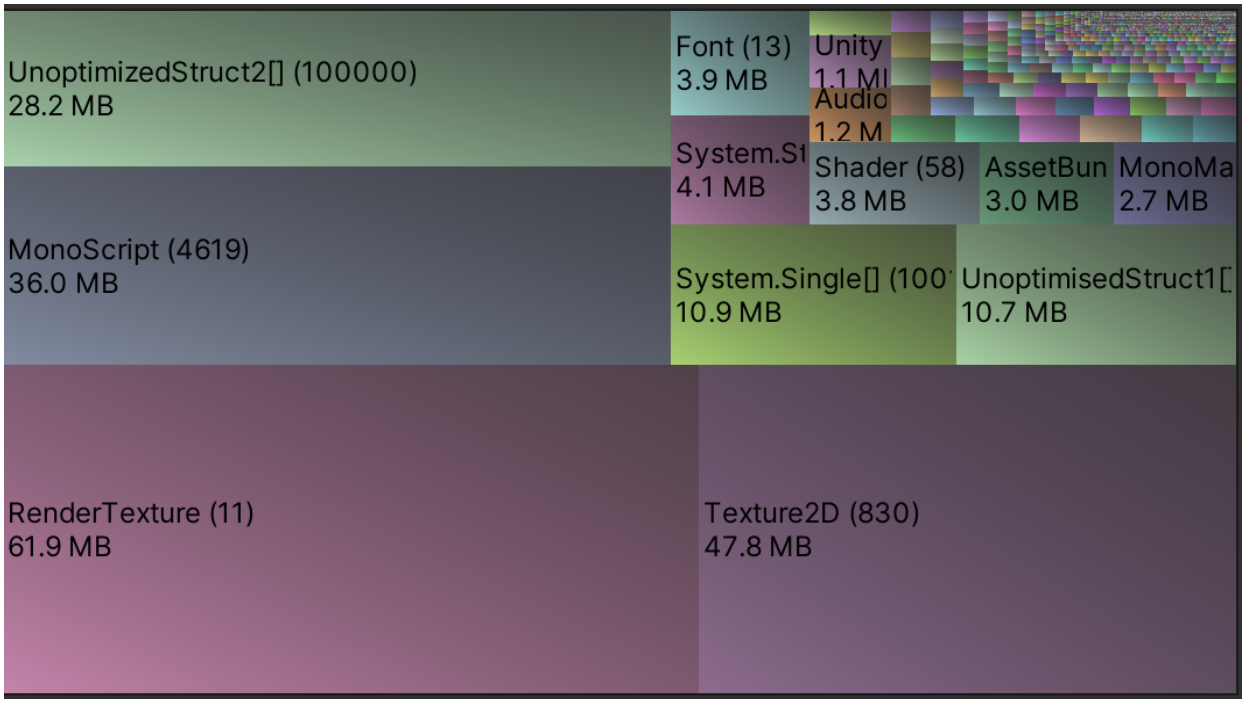
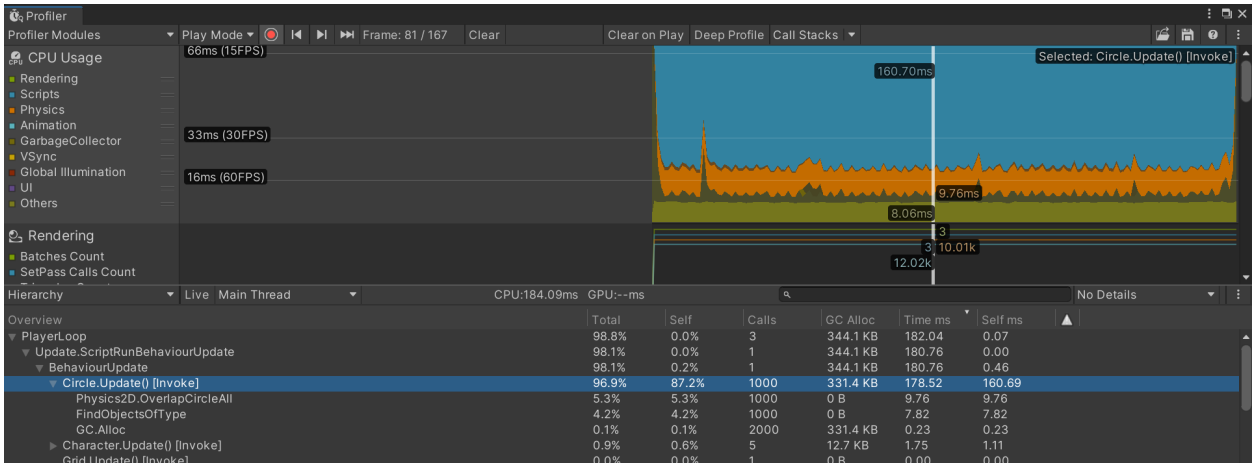
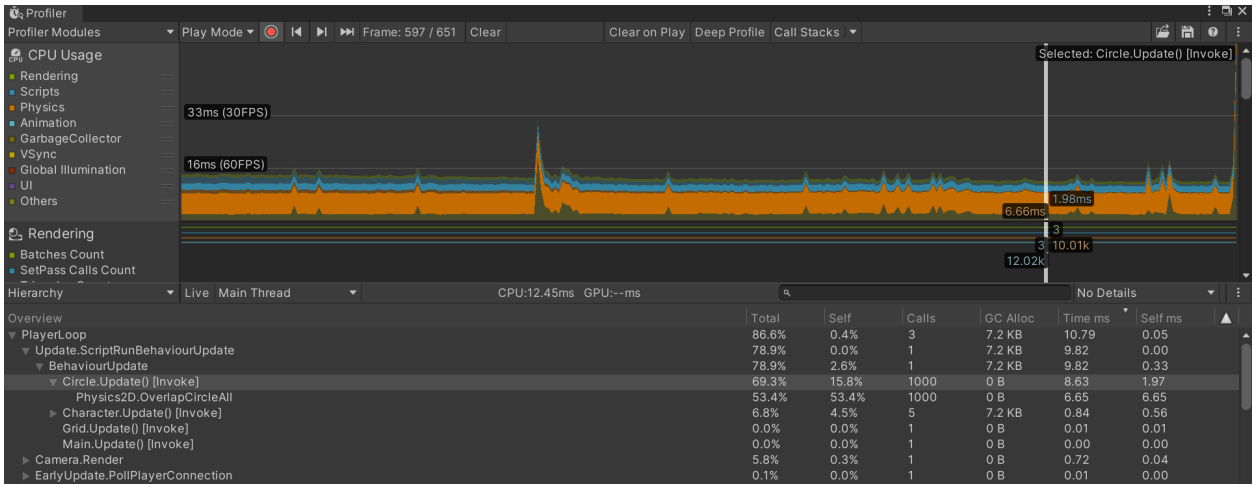


Figure 3 : treemap de la disposition en mémoire après les modifications

Exercices 2



Lors du profilage, on se rend compte que c'est la méthode Update de la classe Circle qui est responsable de la grande majorité de la latence. En creusant un peu plus profond, on constate que c'est la méthode HealNearbyShapes qui est la plus coûteuse et qu'il faut optimiser. Les deux principales optimisations que nous avons effectué sont: changer la façon dont on récupère les cercles proches et limiter les appels à getComponent qui sont coûteux. De plus, les comparaisons d'objet avec null comme condition sont aussi coûteuses. Comme solution nous avons simplement vérifié si les objets existent ou pas selon le contexte (if(object) ou if(!object)).



Exercices 3

Pour optimiser les conteneurs de composantes, nous avons opté pour conserver le dictionnaire qui contient les types de composantes, car il n'y a très peu de types différents et on itère seulement sur un type à donc le fait d'optimiser ce conteneur risque de ne pas apporter d'amélioration de performance. Pour le conteneur de composante, nous avons remplacé le dictionnaire de composantes par une structure sur mesure comme suit:

```
using InnerType = ComponentContainer<IComponent>; // TODO CHANGEZ MOI, UTILISEZ VOTRE PROPRE TYPE ICI
using AllComponents = System.Collections.Generic.Dictionary<uint,ComponentContainer<IComponent>>;
```

Ce nouveau conteneur contient les composantes dans un array de IComponent. L'array a été choisi, car contrairement à un dictionnaire, ses éléments sont placés de manière contiguës dans la mémoire. Cela permet de réduire les cache miss qui font en sorte que l'on doit plus souvent regarder dans la mémoire vive qui est significativement plus lente que la mémoire cache. Bien que le code avec cette solution compile, nous avons été incapable de réaliser l'implementation sans modifier le CRUD et sans perdre certaines fonctionnalité du jeu. Nous n'avons donc pas été en mesure d'améliorer le FPS.

Exercices 4

Pour optimiser la simulation de prédateurs et de proies, il fallait faire une parallélisation en utilisant le job system de Unity. Le job system de Unity permet d'écrire le code sur plusieurs threads pour que le code puisse utiliser tous les cœurs du CPU. L'implémentation se fait sur plusieurs étapes.

Étape 1: On crée un struct de type IJobParallelFor avec un NativeArray pour l'array qu'on veut itérer pour faire une compilation en BurstCompile.

Étape 2: On ajoute la fonctionnalité qu'on veut appliquer à tous les éléments de l'array en assignant un index en paramètre pour les éléments.

Étape 3: On ajoute les attributs nécessaires qui sont utiles pour garder la fonctionnalité.

```
[BurstCompile]
2 references | unknown, 21 minutes ago | 1 author, 2 changes
struct ChangePlantLifeTimeJob : IJobParallelFor
{
    public NativeArray<Vector3> preyTransformsPosition;
    public Vector3 transformPosition;
    public float touchingDistance;
    public float decreasingFactor;

    7 references | unknown, 21 minutes ago | 1 author, 2 changes
    public void Execute(int index)
    {
        if (Vector3.Distance(preyTransformsPosition[index], transformPosition) < touchingDistance)
        {
            decreasingFactor *= 2f;
        }
    }
}
```

Étape 4: On remplit un nouveau NativeArray des éléments sur laquelle on veut itéreré.

```
NativeArray<Vector3> preyTranformPositionArray = new NativeArray<Vector3>(Ex4Spawner.PreyTransforms.Length, Allocator.TempJob);
for (int i = 0; i < Ex4Spawner.PreyTransforms.Length; i++)
{
    preyTranformPositionArray[i] = Ex4Spawner.PreyTransforms[i].position;
}
```

Étape 5: On construit le nouveau Job en attribuant les valeurs aux attributs.

```
ChangePlantLifeTimeJob changePlantLifeTimeJob = new ChangePlantLifeTimeJob
{
    preyTransformsPosition = preyTranformPositionArray,
    transformPosition = transform.position,
    touchingDistance = Ex4Config.TouchingDistance,
    decreasingFactor = _lifetime.decreasingFactor
};
```

Étape 6: On fait la planification des jobs avec un jobHandle pour permettre aux jobs de se faire en parallèle sur tous les cœurs du CPU .

Étape 7: On exécute le job jusqu'à ce qui est complété.

Étape 8: On enlève les éléments du NativeArray qu'on a rempli à l'étape 2.

```
JobHandle jobHandle = changePlantLifeTimeJob.Schedule(Ex4Spawner.PreyTransforms.Length, 64);
jobHandle.Complete();
preyTranformPositionArray.Dispose();
```