Programmation Gameplay

Architecture performante pour le gameplay

Qui suis-je ?

- Ancien de l'ESGI et de paris Dauphine
- Programmeur Gameplay/IA chez Eugen System
- Enseignant en Jeu vidéo

Objectifs du cours

- Comprendre le processus d'itération dans la conception d'un jeu vidéo
- Aborder les notions d'optimisations liées à l'architecture logicielle
- Comprendre et implémenter certaines de ces structures

Sommaire

- Lundi : Object Pooling + TD
- Mercredi : Correction TD + Théorie ECS
- Jeudi : Projet ECS
- Vendredi : Soutenances

Projet

- PACMAN ECS
- Groupe de 2 maximum
- A rendre sur Git

Des questions ?

Object Pooling

Qu’est ce que c’est ?

- Créer des objets au chargement
- Pour les réutiliser dans le jeu sans allocation mémoire

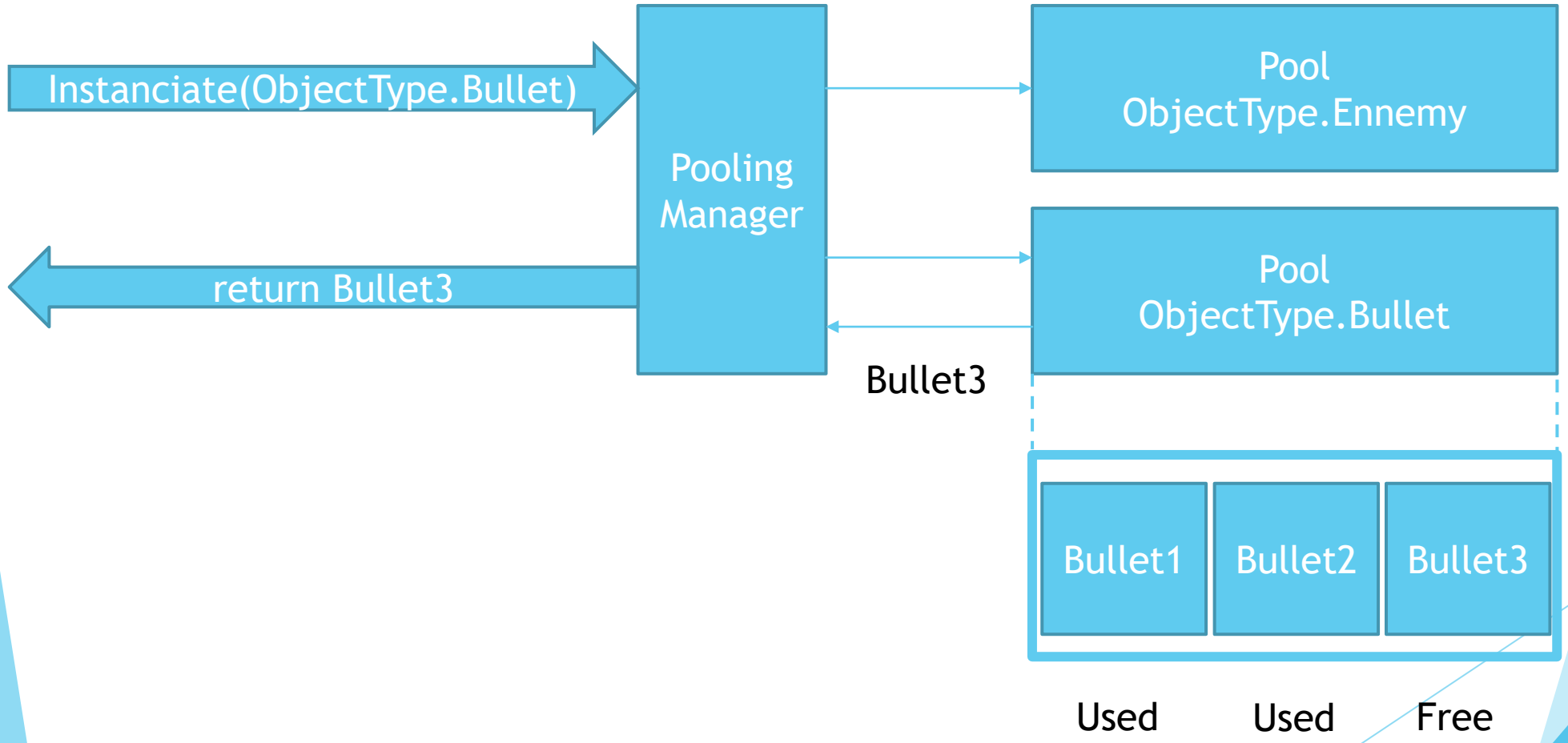
Pourquoi ?

- L’allocation mémoire très couteuse
- Chargement long = pas trop dérangeant
- Lag en jeu = très dérangeant

Comment faire ?

- Sous Unity deux méthodes :
  - Gameobject déjà dans la scène
    - Nombre maximum défini
    - Mais pas adaptable durant le développement
    - Poids du build plus lourd
  - Instancier les Gameobjects au chargement
    - Nombre maximum adaptable
    - Poids du build plus léger
    - Chargement plus long

Schéma


Instanciate(ObjectType.Bullet)
return Bullet3
Pooling Manager
Pool ObjectType.Ennemy
Pool ObjectType.Bullet
Bullet3
Bullet1
Bullet2
Bullet3
Used
Used
Free

TD – Object Pooling

PoolableObject

- Créez la class PoolableObject
  - Dérivant de monobehaviour
  - Une variable pour savoir si l'objet est dans la pool ou sur la scène
  - Une fonction « Init » (possiblement abstraite)
    - Remplir avec SetActive(true)
  - Une fonction « DeInit » (possiblement abstraite)
    - Remplir avec SetActive(false)

Pool

- Créez l’enum ObjectType (exemple : Player, Ennemy etc…)

- Créez la class Pool contenant
  - Une liste de PoolableObject
  - Une fonction d’initialisation qui instancie les PoolableObject
    - public void Initialize(PoolableObject parPrefabObject, int parNumber)
  - Une fonction pour récupérer un objet dans la pool

ObjectForPool

- Il s’agit d’une petite struct qui va nous permettre de définir les objets à instancier dans le PoolManager

```
using System;
[Serializable]
public struct ObjectForPool
{
    public ObjectType ObjectType;
    public PoolableObject Prefab;
    public int Number;
}
```

PoolManager

```
public class PoolManager : MonoBehaviour
{
    static public PoolManager Instance() { return _singleton; }
    static private PoolManager _singleton;

    private void Awake()
    {
        _singleton = this;
    }
}
```

- Créez la class pool manager
  - Sous forme de singleton – mais avec une fonction d'initialisation
  - Contenant un dictionnaire privé de [ObjectType, Pool]
  - Contenant une List public de ObjectForPool
  - Une fonction pour récupérer un objet dans la pool
    - Exemple : PoolManager.Instance().GetPooledObject(ObjectType.Player)
  - Une fonction pour rendre un objet dans la pool
    - Exemple : PoolManager.Instance().ReleasePooledObject(myObject)

TestScript

- Faire un fichier de test pour get tout les éléments dans une pool et tout remettre.
- Testez

Petit plus : Le scriptable Object

- Créez un Scriptable Objet
  - Contenant un dictionnaire de ObjectType vers [int, PoolableObject]
  - Ajouter ce scriptableObject dans PoolManager et l'utiliser pour instancier les pools.
  - Bonus : Ajouter un système qui fait grandir les pool lorsque le nombre est dépassé et ne pas oublier de throw un warning pour avertir l'utilisateur
    - Pour instancier le ScriptableObject :
      - Assets / Create / ScriptableObjects / ObjectPoolGeneratorData

```
[CreateAssetMenu(fileName = "ObjectPoolGeneratorData", menuName = "ScriptableObjects/ObjectPoolGeneratorData")]
public class ObjectPoolGenerator : ScriptableObject {
```

TD Correction

Poolable Object

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PoolableObject : MonoBehaviour
{
    public bool IsActive()
    {
        return gameObject.activeInHierarchy;
    }

    public void Init()
    {
        gameObject.SetActive(true);
    }

    public void DeInit()
    {
        gameObject.SetActive(false);
    }
}
```

Pools

```
public enum ObjectType
{
    Player,
    Ennemy,
    Length
}

public class Pool
{
    List<PoolableObject> ObjectInPool;

    public void Initialize(PoolableObject parPrefabObject, int parNumber)
    {
        ObjectInPool = new List<PoolableObject>();
        for (int i = 0; i < parNumber; ++i)
        {
            PoolableObject go = MonoBehaviour.Instantiate(parPrefabObject);
            go.DeInit();
            ObjectInPool.Add(go);
        }
    }

    public PoolableObject PullObject()
    {
        int numberObjectInPool = ObjectInPool.Count;
        for (int i = 0; i < numberObjectInPool; ++i)
        {
            if (!ObjectInPool[i].IsActive())
                return ObjectInPool[i];
        }
        return null;
    }
}
```

PoolManager

```
public class PoolManager : MonoBehaviour
{
    static public PoolManager Instance() { return _singleton; }
    static private PoolManager _singleton;

    private void Awake()
    {
        _singleton = this;
        Initialize();
    }

    public List<ObjectForPool> ObjectPrefab;
    Dictionary<ObjectType, Pool> Pools;

    public void Initialize()
    {
        Pools = new Dictionary<ObjectType, Pool>();
        foreach (ObjectForPool obj in ObjectPrefab)
        {
            Pool newPool = new Pool();
            newPool.Initialize(obj.Prefab, obj.Number);
            Pools.Add(obj.ObjectType, newPool);
        }
    }

    public PoolableObject GetPooledObject(ObjectType parObjectType)
    {
        return Pools[parObjectType].PullObject();
    }

    public void ReleasePooledObject(PoolableObject parObject)
    {
        parObject.DeInit();
    }
}
```

Test Script

```
void Start()
{
    Objects = new List<PoolableObject>();
    PoolManager poolManager = PoolManager.Instance();

    for (int i = 0; i < 10; ++i)
    {
        PoolableObject obj = poolManager.GetPooledObject(ObjectType.Ennemy);
        if (obj != null)
        {
            obj.Init();
            Objects.Add(obj);
        }
    }

    int objectCount = Objects.Count;
    for (int i = 0; i < objectCount; ++i)
    {
        if (Objects[i] != null)
            poolManager.ReleasePooledObject(Objects[i]);
    }
}
```

Bonus

```
[CreateAssetMenu(fileName = "ObjectPoolGeneratorData", menuName = "ScriptableObjects/ObjectPoolGeneratorData")]
public class ObjectPoolGenerator : ScriptableObject
{
    public List<ObjectForPool> ObjectForPools;
}
```

ECS - Préambule

Singleton Pattern

```
public class PoolManager : MonoBehaviour
{
    static public PoolManager Instance() { return _singleton; }
    static private PoolManager _singleton;

    private void Awake()
    {
        _singleton = this;
    }
}
```

- L'idée du singleton est d'avoir un objet unique à utiliser partout
- Exemple : GameManager
  - Besoin : le gameManager est utilisé souvent et partout, on voudrait pouvoir y accéder facilement
  - GameManager.StartNewGame()
  - Si on utilise le pattern Singleton : GameManager.Instance().StartNewGame()
  - L'idée est que le singleton soit construit au premier appel et existe ensuite tout le long du programme.

Generic (C#)

- List<int> est un generic, il s'agit d'un type qui s'adapte au type spécifier entre « < > »
  - List<float> contiendra des floats, List<int> des int, mais la class est la même « List »

- Public class List<T> pour déclarer une classe generic
  - T étant le type générique
  - On pourra donc faire public void Add(T newValue);

Generic (C#)

- On peut aussi restreindre le type T
  - Public class List<T> where T : Toto (un exemple de list n'acceptant que les T héritant de la classe Toto
- On peut faire des classes generic avec plusieurs types génériques
  - Public class Dictionary<T, U>
- Attention en C# le type (T) doit être connu à la compilation

Entity Component System

Qu’est-ce que l’ECS ?

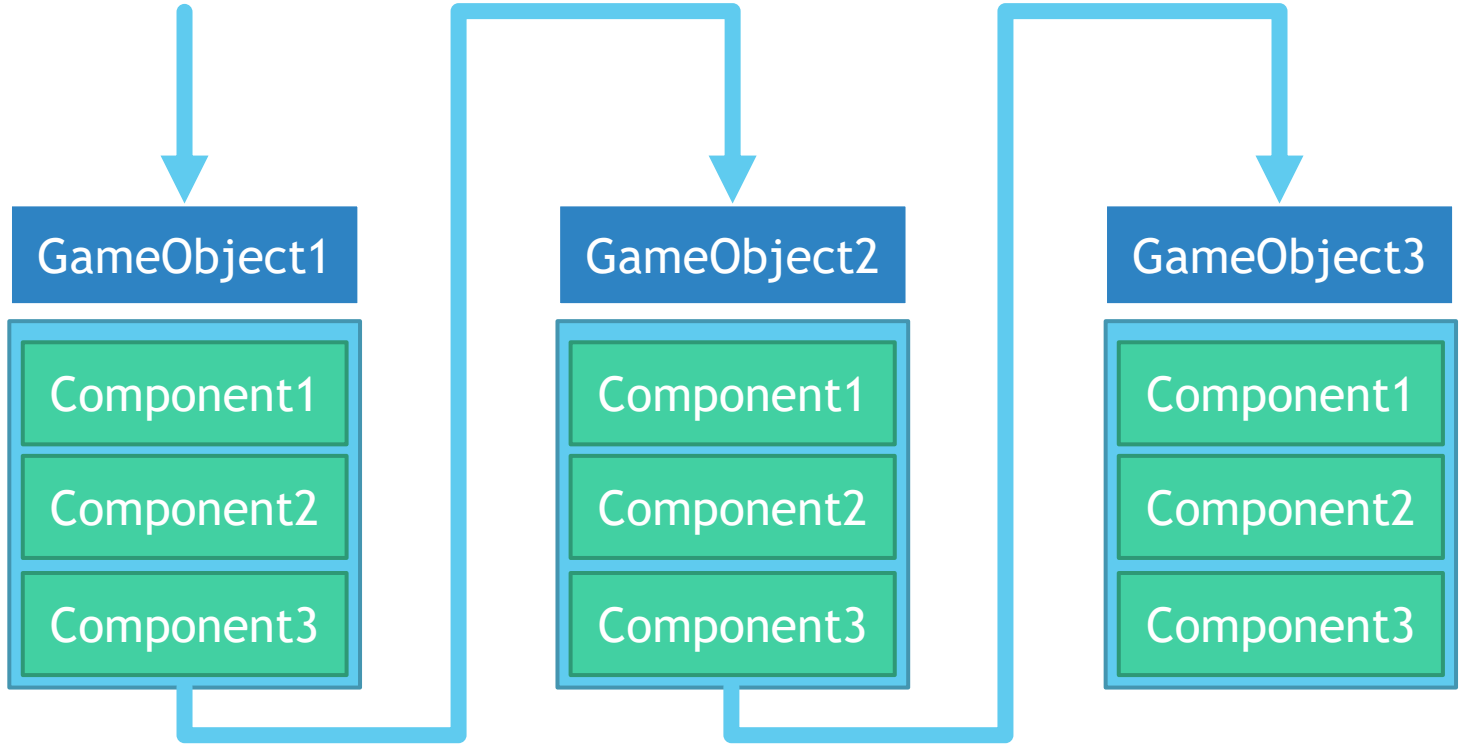
- Un type d’architecture composé de :
  - **Entity :** il s’agit de n’importe quel objet en jeu
  - **Component :** il s’agit des données ajoutant une fonctionnalité en jeu
  - **System :** il s’agit de l’update des components

Pourquoi changer d’architecture ?

- Performance
- Flexibilité
- Maintenabilité
- Interface avec les autres jobs (game design, mais aussi graphistes)

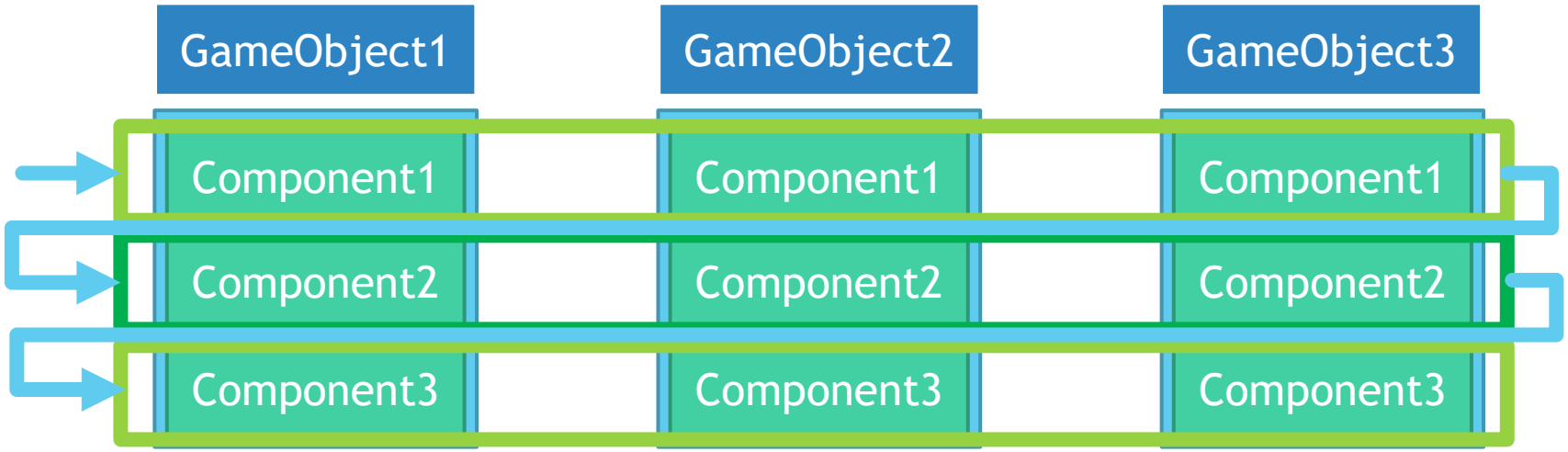
Qu’est-ce que ça change ?

Version Classique Unity



Qu’est-ce que ça change ?

Version ECS



Qu’est-ce que ça change ?

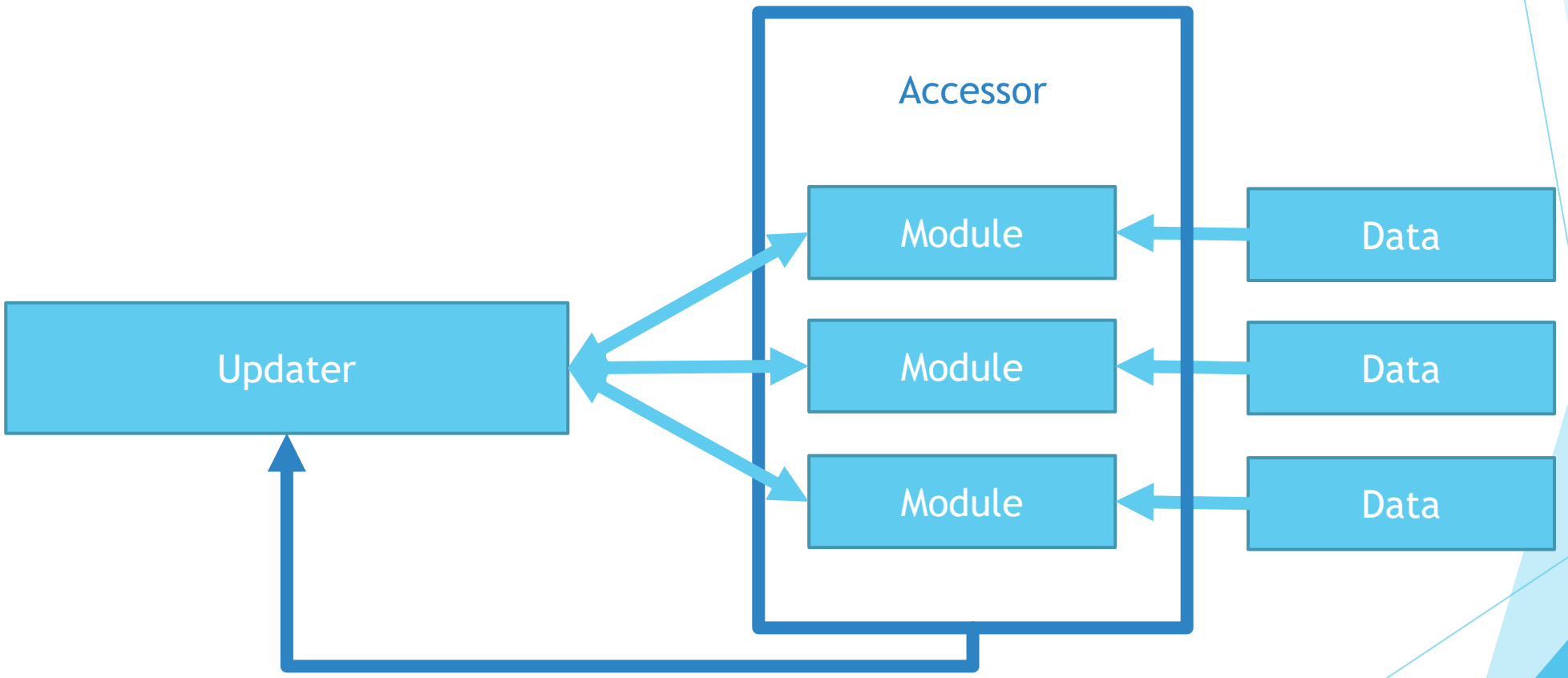
Non, mais concrètement

- Composition au lieu d’héritage
- Dépendance plus simple et plus safe
- Memory Management
  - Data Caches
- Plus orienté fonctionnalité
  - Exemple : On veut ajouter temporairement une fonctionnalité à un objet il suffit de lui rajouter le component et… c’est tout.

En profondeur

- Entité : Un objet du jeu
- Component (aussi appelé Module) : contiens les données du jeu potentiellement modifiées par l'état du jeu
- System (aussi appelé Updater) : process tous les Modules pour traiter les données
- Accessor<ModuleClass> : il s'agit d'une classe qui contient tous les modules d'un type
  - NB: Dans l'ECS Unity il s'agit des NativeArray et dérivé.

Un schéma



Un peu de (pseudo) code

```
Public class MyUpdater : IUpdater
{
Public void SystemUpdate()
{
    TAccessor<MyModule> myModuleAccessor = TAccessor<MyModule>.Instance();
    TAccessor<MyOhterModule> myOtherModuleAccessor = TAccessor<MyOhterModule>.Instance();
    Foreach(var module in myModuleAccessor.GetAllModules())
    {
        GameObject myEntity = module.Value.GameObject;
        MyOtherModule otherModule = myModuleAccessor.TryGetModule(myEntity);
        If(otherModule != null)
            DoSomething();
    }
}
}
```

Consignes du projet

- Réaliser l'ECS sous Unity comme vu en cours
  - System (Updater) : IUpdater
  - Component (Module) : TModule
  - Accessor : TAccessor<T>
  - Entity : simplement un GameObject

Consignes du projet

- Nous allons réaliser un jeu type pacman avec plusieurs fantômes et plusieurs Pacman

- Réaliser l'ECS ainsi que les composants suivants : (Module + Updater si besoin)
  - **FollowTarget :** Composant qui fait que l'entité suit une autre entité (fantômes suit les PacMan) (2 points)
  - **TargetEdible :** Composant qui fait que l'entité cherche les EdibleModule pour les mangers (2 points)
  - **EdibleModule :** Composant qui fait que l'entité peut être mangée par les Pacman (1 point)
  - **ScoreModule :** Composant qui stocke les points de l'entité. (1 point)
  - **KillPlayerScript :** L'interaction physique entre un fantôme et un Pacman (1 point)
  - **EatEdibleScript :** L'interaction physique entre un Pacman et un fruit (1 point)

  - **Attention :** il s'agit d'une liste minimale à avoir, mais vous serez amené à faire d'autres classes en plus

Consignes du projet

- À rendre sur git
- M’envoyer le lien git après avoir formé les groupes
  - emerick.lecomte.pro@gmail.com
- Groupe de 2
- À rendre au plus tard le : 10/07/2020 16h59m59s

Notation du projet

Disclaimer : il s'agit d'un barème à titre indicatif qui peut changer

- Les composants de l'ECS + Unity - 8 points
- Les composants demandés - 8 points
- Les composants aux choix (1 minimum) - 4 points

- Pour avoir la totalité des points dans une section
  - Code propre, commenté si besoin
  - Structure lisible et optimisée
  - Respect de l'E.C.S.

Des questions ?