Entity Component System

Projet: R-Type (Novembre 2021)

Antoine DEGEETER, Eliot MARTIN, Julien DELPHINE, Thomas GILLET

Introduction:

Le but de ce document est d'expliquer plus en détail notre "entity component system" (ECS) que l'on a créé et utilisé pour le projet R-Type de 3ème année d'Epitech.

Bases:

Entités:

- Chaque entité est un int pouvant aller de 0 à MAX_ENTITY (define valant 100). On pourra donc s'en servir comme index dans des tableaux.
- Une classe EntityHandler va gérer ces entités.

Composants:

- Un composant est une struct contenant différents paramètre en fonction de son type (ex: Composant Movable aura un int speed)
- Les classes ComponentArray et ComponentHandler vont gérer ces composants.
-
```
struct Movable {
    float speed;
    std::string direction;
    TypeEnt typeent;
};
```
(Exemple de composant)

Systèmes:

- Un système est une classe effectuant une ou plusieurs actions sur des entités ayant certains composants en particulier (cf. Cartes)
- La classe SystemHandler va gérer les systèmes.

Cartes:

- Une carte est ce qui va permettre aux systèmes de savoir quelles entités ils vont devoir gérer. Celle-ci sait quelle(s) entité(s) a quel(s) composant(s) grâce à un std::bitset<32> qui aura des “1” aux index de chaque composant. (ex: Le composant Movable est le deuxième composant ajouté donc si l’entité n’a que ce composant la carte sera : 01000...)

EntityHandler:

Attributs:

```
private:
    std::queue<int> _availableEntitySpace;
    std::array<std::bitset<MAX_COMPONENT>, MAX_ENTITY> _cards;
    int _currentNbEntity;
```

_availableEntitySpace stock les ids des entités, _cards stock les cartes des entités, _currentNbEntity compte le nombre d’entité actives.

Méthodes:

Constructeur() -> initialise _availableEntitySpace de 0 à MAX_ENTITY pour stocker les Id des entités.

CreateEntity() -> récupère le premier Id de _availableEntitySpace et le supprime de la queue. Augment de 1 _currentNbEntity.

Destroyentity(entity) -> remet l’id de l’entity dans _availableEntitySpace, diminue de 1 _currentNbEntity et supprimes la carte de _cards.

SetCard(entity, card) -> ajoute la carte dans _cards à l’index entity.

GetCard(entity) -> renvoie la carte de l’entity.

template<typename T> ComponentArray:

Attributs:

```
private:
    std::array<T, MAX_ENTITY> _componentArray{};
    std::map<int, int> _entityToIndex;
    std::map<int, int> _indexToEntity;
    int _size{};
```

_componentArray stock un (template typename T) l'entity et le composant donné, _entityToIndex permet de récupérer l'index de l'entité dans _componentArray, _indexToEntity permet de récupérer l'entité dans _componentArray grâce à son index. _size contient le nombre d'entités ayant ce composant au total.

Méthodes:

InsertData(entity, component) -> ajoute le composant à l'entity en trouvant un index disponible dans _componentArray

```
newindex = _size;
_entityToIndex[entity] = newindex;
_indexToEntity[newindex] = entity;
_componentArray[newindex] = component;
_size++;
```

RemoveData(entity) -> supprime l'entité et son composant de _componentArray et met le dernier composant à l'index libéré

```
indexofremoved = _entityToIndex[entity];
indexoflast = _size - 1;
_componentArray[indexofremoved] = _componentArray[indexoflast];

entityOfLastElem = _indexToEntity[indexoflast];
_entityToIndex[entityOfLastElem] = indexofremoved;
_indexToEntity[indexofremoved] = entityOfLastElem;

_entityToIndex.erase(entity);
_indexToEntity.erase(indexoflast);
_size--;
```

GetData(entity) -> renvoie le composant demandé (grâce au template) d'une entité

ComponentHandler:

Attributs:

```
private:
    std::map<const char *, int> _componenttypes;
    std::map<const char *, std::shared_ptr<IComponentArray>> _componentarray;
    int _nextcomtype{};
```

Componenttypes va stocker le nom du composant et son index (si le premier créé alors 0, deuxième 1 etc...) pour pouvoir plus tard créer la carte pour les entités et les systèmes, componentarray va stocker le nom du composant et un pointeur vers un ComponentArray qu'on a vu juste au dessus, nextcomtype compte le nombre de composants qui seront enregistré et servira pour componenttypes

Méthodes:

NewComponent() -> sert à "enregistrer" un nouveau composant dans le moteur de jeu, il sera ajouté à tous les attributs de la classe.

GetComponentType() -> Renvoie le type du composant passé en template en d'autre terme renvoie son index

GetComponentArray() -> Renvoie un pointeur vers un ComponentArray du composant voulu.

AddComponent(entity, component) -> va appeler InsertData du ComponentArray récupéré grâce au composant dans le template ce qui va ajouter le composant à l'entité.

```
template<typename T>
void addComponent(int entity, T component)
{
    getComponentArray<T>()->insertData(entity, component);
}
```

RemoveComponent(entity) ->appelle RemoveData du ComponentArray récupéré grâce au composant dans template. Supprime le composant de l'entité

GetComponent(entity) -> appelle GetData du ComponentArray récupéré grâce au composant dans template. Renvoie le composant voulu de l'entité donné

System:

Contient simplement un set d'entités entities qui auront la même carte que le système.

```
public:
    System() {}
    ~System() {}
    std::set<int> _entities;
private:
```

SystemHandler:

Attributs:

```
private:
    std::map<const char *, std::bitset<MAX_COMPONENT>> _cards;
    std::map<const char *, std::shared_ptr<System>> _systems;
```

cards contient les noms et les cartes de chaque système, systems va stocker les nom des systèmes et des pointeurs vers ceux-ci.

Méthodes:

NewSystem() -> sert à "enregistrer un nouveau système dans le moteur

SetCard(card) -> va ajouter à cards le nom du système et sa carte

EntityDestroyed(entity) -> va retirer l'entity du set entities

EntityCardChanged(entity, entitycard) -> sert à vérifier si l'entité possède la carte qui correspond à celle du système pour soit la supprimer du set entities du système ou bien l'ajouter

```
for (auto &item : this->_systems) {
    auto &type = item.first;
    auto &system = item.second;
    auto &systemcard = this->_cards[type];

    if ((entitycard & systemcard) == systemcard)
        system->_entities.insert(entity);
    else
        system->_entities.erase(entity);
}
```

Engine:

Cette classe sert à coordonner toutes les classes que l'on vient de voir. Cela permet de tout regrouper à un seul endroit pour pouvoir ensuite tout contrôler avec un seul objet.

Attributs:

```
private:
    std::unique_ptr<ComponentHandler> _componenthandler;
    std::unique_ptr<EntityHandler> _entityhandler;
    std::unique_ptr<SystemHandler> _systemhandler;
```

Il s'agit de pointeurs vers chacun des handlers

Méthodes:

Init() -> va initialiser les trois pointeurs de la classe

CreateEntity() -> va créer une entité grâce à son pointeur vers _entityhandler:

```
return (this->_entityhandler->createEntity());
```

DestroyEntity(entity) -> va appeler les fonctions pour détruire les entités.

```
this->_entityhandler->destroyEntity(entity);
this->_systemhandler->entityDestroyed(entity);
this->_componenthandler->entityDestroyed(entity);
```

NewComponent() -> va appeler la fonction NewComponent() du pointeur vers _componenthandler

AddComponent(entity, component) -> va permettre d'ajouter un component à tout le moteur en communiquant avec toutes les classes pour mettre en place les cartes, les componentArray et les sytèmes:

```
this->_componenthandler->addComponent<T>(entity, component);
auto card = this->_entityhandler->getCard(entity);
card.set(this->_componenthandler->getComponentType<T>(), true);
this->_entityhandler->setCard(entity, card);
this->_systemhandler->entityCardChanged(entity, card);
```

RemoveComponent(entity) -> fait l'inverse de AddComponent():

```
this->_componenthandler->removeComponent<T>(entity);
auto card = this->_entityhandler->getCard(entity);
card.set(this->_componenthandler->getComponentType<T>(), false);
this->_entityhandler->setCard(entity, card);
this->_systemhandler->entityCardChanged(entity, card);
```

GetComponent(entity) -> renvoie le composant voulu de l'entité donné

GetComponentType() -> renvoie le type (l'index) du composant voulu.

NewSystem() -> va appeler la fonction NewSystem() du pointeur vers _systemhandler

SetSystemCard(card) -> va appeler la fonction SetSystemCard(card) du pointeur vers _systemhandler

Comment utiliser cet ECS ? (En pratique)

Engine:

Déclaration de l'objet Engine en global:

```
Engine moteur;
```

Initialisation et composants:

Initialisation du moteur et ajout des différents composant à celui-ci:

```
moteur.init();
moteur.newComponent<Drawable>();
moteur.newComponent<Movable>();
moteur.newComponent<Bullet>();
moteur.newComponent<Ennemy>();
moteur.newComponent<Player>();
moteur.newComponent<Powerup>();
```

Enregistrement des systèmes:

Création de variable contenant les fonctions des systèmes et enregistrement de ceux-ci dans le moteur:

```
auto drawSystem = moteur.newSystem<DrawSystem>();
drawSystem->init();
```

Creation d’une carte:

Création d’une carte permettant aux systèmes de savoir quelles entités traiter:

```
std::bitset<MAX_COMPONENT> card;
card.set(moteur.getComponentType<Drawable>());
card.set(moteur.getComponentType<Movable>());
moteur.setSystemCard<DrawSystem>(card);
moteur.setSystemCard<MoveSystem>(card);
```

Creation d’une entité avec composants:

```
int playerentity;
playerentity = moteur.createEntity();
moteur.addComponent(playerentity, Drawable{
    .position = sf::Vector2f(70, 170),
    .size = sf::Vector2f(160, 80),
    .texture = textureplayer,
    .bulletsprite = playsprite
});
moteur.addComponent(playerentity, Movable{
    .speed = 10,
    .direction = "",
    .typeent = TypeEnt::PLAYER
});
```

Exemple de système:

```cpp
#include "movesystem.hpp"

extern Engine moteur;

void MoveSystem::update(float delta)
{
    for (auto &ent : _entities) {
        auto &movable = moteur.getComponent<Movable>(ent);
        auto &drawable = moteur.getComponent<Drawable>(ent);
        if (movable.direction == "right") {
            drawable.position.x += movable.speed;
        }
        if (movable.direction == "down") {
            drawable.position.y += movable.speed;
        }
        if (movable.direction == "left") {
            drawable.position.x -= movable.speed;
        }
        if (movable.direction == "up") {
            drawable.position.y -= movable.speed;
        }
    }
}
```

Ce Système parcours ses entités en boucle et récupère les composants dont il a besoin:

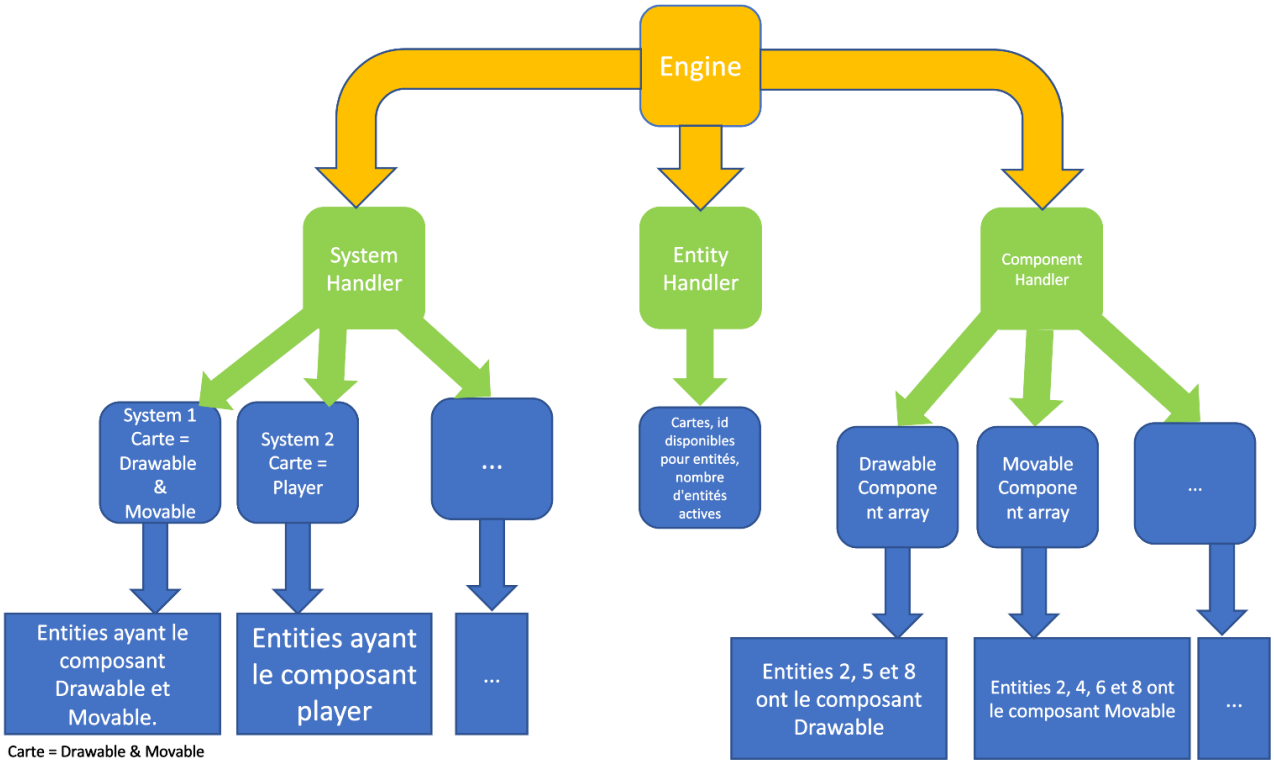
Ici Movable et Drawable. On peut ensuite modifier certaines variables des composants de l'entité pour la mettre à jour.

Ici on récupère la direction qui est stocké dans le composant Movable et on le change la position du composant Drawable grâce à la variable speed située dans le composant Movable.

On fait ça pour chaque entité qui a la carte du système. En l'occurence ici toutes les entités qui ont Movable et Drawable.

Il est donc très facile d'ajouter des systèmes.

Schéma:


Engine
System Handler
Entity Handler
Component Handler
System 1 Carte = Drawable & Movable
System 2 Carte = Player
...
Cartes, id disponibles pour entités, nombre d'entités actives
Drawable Component array
Movable Component array
...
Entities ayant le composant Drawable et Movable.
Entities ayant le composant player
...
Entities 2, 5 et 8 ont le composant Drawable
Entities 2, 4, 6 et 8 ont le composant Movable
...
Carte = Drawable & Movable