EntityComponentSystem(ECS)についての簡易説明

目次

1. ECS とは

主にゲーム開発で使用されているソフトウェアアーキテクチャパターンである。ECS は継承よりコンポジションの原則に従うことで、より柔軟にエンティティを定義することを可能にする。エンティティとは、ゲームのシーンの中のすべての実体であるオブジェクトのことである（例えば、敵、銃弾、乗り物など）。すべてのエンティティは、付加的な振る舞いや機能を追加するものである 1 つ以上のコンポーネントから構成される。したがって、エンティティの振る舞いは、実行時にコンポーネントを追加あるいは削除することで変更可能である。これは、深く幅広い継承階層を除去し、その理解・保守・拡張が難しくなりあいまいになるという問題を取り除く。ECS の一般的なアプローチはデータ指向設計の手法と高い互換性を持ち、よく組み合わせられる。

※Wikipedia より引用

今回実装した ECS は cppconference2015 にて行われたものを参考にしています。

https://www.youtube.com/watch?v=NTWSeQtHZ9M

※この文書内では GameObject を Entity で統一します

2.通常のオブジェクト指向との違い

通常の場合

```
struct Entity
{
	virtual ~Entity() {}
	virtual void update(){}
	virtual void draw() {}
};
struct Player : public Entity
{
	void update() override { /*Playerの処理*/}
	void draw() override {}
};
```

ECS の場合

```
struct Component
{
	virtual ~Component() {}
	virtual void update(){}
	virtual void draw() {}
};
struct Entity
{
	std::vector<std::unique_ptr<Component>> components;
	void update() { for (auto& c : components) c->update(); }
	void draw() { for (auto& c : components) c->draw(); }
};
```

通常は継承ベースで実装していますが、ECS は委譲で機能の追加をしています

ECS で組む場合、Entity はコンポーネントを格納するためのコンテナにすぎません。

3.プログラム例

```
//データはメソッドを持たない
//データはそれ自体が明確な意味を持つ
//Component がそれらのデータに対する処理を行う
struct HP : public ComponentData{
        int val;
};
struct Pos : public ComponentData{
        Vec2 val;
        Pos() = default;
        Pos(float xx, float yy) :
                val(xx, yy)
        {}
};
class Move : public Component{
private:
        Pos* pos_;
public:
        void Initialize() override      {
                pos_ = &entity->GetComponent<Pos>();
        }
        void Update() override  {
                ++pos_->val.x;
        }
};
//簡単なファクトリ関数を用意すれば複製も容易にできる
Entity* MakePlayer(){
//Entityは自分自身だけではインスタンス化できないため、マネージャーが生成する
        Entity* entity = entityManager.AddEntity();
        entity->AddComponent<Pos>();
        entity->AddComponent<HP>();
        entity->AddComponent<Move>();
        return entity;
}
```

4．イベント処理はどうする？

イベント処理とは、例えばプレイヤーが入力によって弾 Entity を生成する等の処理です。

まだ明確な解決策は出ていませんが、現在私は以下の方法で行っています。

まずシーンとシーン内で実行する関数を格納し、それらを処理するシングルトンクラスを作成します。

```
class Singleton final
{
private:
using SceneManager = Scene::SceneManager;
std::vector<std::pair<SceneManager::State, std::function<void()>>> events;
public:
	void Add(const SceneManager::State state, std::function<void()> func)
	{
		events.emplace_back(std::make_pair(state, func));
	}
	void Update()
	{
		for (auto& it : events)
		{
			if (SceneManager::Get().GetCurrentScene() == it.first)
			{
				it.second();
			}
		}
	}
};
```

イベントの登録はこのようにしてます。

```
Event::EventManager().Get().Add(
Scene::SceneManager::State::Game,
Event::CollisionEvent::AttackCollisionToEnemy);
```

呼び出しはこのようにします

```
//更新処理内
Event::EventManager::Get().Update();
```

あまり良い方法だとは思っていませんが、各自の作業の粒度を下げることができます。

5.既知の問題点

・同名のコンポーネントを重複して追加できない
例えば同一機能の矩形のコリジョンが2つほしい場合、名前だけ異なるクラスが複製される。(BoxColliderとBoxCollider2のようになる)
・データの通信が生々しい
・エンティティマネージャーへの参照が面倒、しかし、シングルトンにするとメモリ管理的な面で問題が発生しやすく、バグの原因になりやすい(1敗)。
・親子関係がentity側で実装されていない
・オブジェクト指向ベースなため処理負荷が高い
・似たようなデータ、コンポーネントが増える。
・