コンポーネントシステム

ノーマル（最初から）

コンポーネントてっ何？から
簡単な実装まで

ハード（応用）

より実践的な実装へ
ECSの解説

ベリーハード（より発展的）

更に進化する
ECS実装!!

コンポーネントってなに？

コンポーネントとは？

何かの部品のこと。

車でいえば「タイヤ」「エンジン」等のことを、一つのコンポーネントと呼ぶ。

使うとどんなメリットがあるの？

- 部品ごとで配置しているので使い回し（再利用）が出来る。
- 独立性を高く出来るので、バグも発生しずらく、発見しやすい

シンプルなコンポーネントシステム１（実装例）

#簡単な実装例

```cpp
#include <iostream>
#include <assert.h>
#include "GameObject.h"

class Player : public GameObject
{
public:

};

class Transform : public Component
{
public:

    Transform(GameObject* owner) : Component(owner)
    {
        printf("Transform\n");
    }
};

int main()
{
    auto player = Player();

    // コンポーネントの追加＆取得（正常な値かをチェックしている）
    assert(!player.AddComponent<Transform>().expired());
    assert(!player.GetComponent<Transform>().expired());

    // コンポーネントの消去（実際には消去されていない）
    player.RemoveComponent<Transform>();

    // コンポーネントは消去されてないが、取得不可
    assert(player.GetComponent<Transform>().expired());

    // ここで初めてコンポーネントが消去される
    player.Update();

    // 取得不可
    assert(player.GetComponent<Transform>().expired());
}
```

仮コードは、皆さんが知ってるシンプルな実装です。

AddCompoent関数でコンポーネントを追加。

GetComponent関数でコンポーネントの取得。

RemoveComponent関数でコンポーネントの消去。

後で実装部分で解説しますが、

変わった実装として、遅延消去するようにしています。

#プラスα

遅延消去する理由は、

「別スレッドで使っている」等の何らかの理由で、

消去を遅らせたい時に有効だからです。

シンプルなコンポーネントシステム２（実装例）

#簡単な実装例

```cpp
#pragma once

class GameObject;

class Component
{
public:

    Component() = delete;
    Component(GameObject* owner) { m_owner = owner; }

    virtual ~Component() {}

public:

    virtual void Init() {}

    virtual void UnInit() {}

    virtual void Update() {}

    // 消去申請を受付る関数で消去待ち状態に入ります。
    virtual void Destory()
    {
        m_requestDestroy = true;
    }

    // このコンポーネントが現在消去可能なのかを返します。
    virtual bool Erasable()
    {
        return true;
    }

    bool RequestDestroy()
    {
        return m_requestDestroy;
    }

protected:

    // 所属ゲームオブジェクト
    GameObject* m_owner = nullptr;

    // このコンポーネントが消去前であるか
    bool m_requestDestroy = false;
};
```

シンプルな実装なので大事な所だけ解説します。

今回は遅延消去を実装するため、

Destory関数は消去待ちフラグを立てるだけにしています。

Erasable関数では派生クラスの実装方法によって、

呼び出し時に消去出来るかを返す関数です。

#プラスα

Erasable関数の使用例としては、

レンダリングスレッドで使われているModelコンポーネント等、

描画クラスが描画が終了していると真を返すとかですね。

シンプルなコンポーネントシステム３（実装例）

#簡単な実装例

```cpp
#pragma once
#include <vector>
#include <memory>
#include "Component.h"

class GameObject
{
public:

    virtual ~GameObject() {}

public:

    virtual void Init() {}

    virtual void Uninit() {}

    virtual void Update()
    {
        // 実際にコンポーネントの消去を行う
        std::erase_if(m_components, [](auto& component)
        {
            // 消去申請がされたコンポーネントで && 消去可能なやつのみ
            return component->RequestDestroy() && component->Erasable();
        });
    }
```

ゲームオブジェクトも非常にシンプルです。

大事なところは、

Update関数内で遅延消去を実装しているぐらいです。

シンプルなコンポーネントシステム 4（実装例）

#簡単な実装例

```
public:

    template<class T>
    std::weak_ptr<T> AddComponent()
    {
        auto component = std::make_shared<T>(this);
        m_components.emplace_back(component);
        return component;
    }

    template<class T>
    std::weak_ptr<T> GetComponent()
    {
        for (auto& component : m_components)
        {
            if (typeid(*component.get()) == typeid(T))
            {
                // 消去待ちのコンポーネントは取得不可にする。
                if (component->RequestDestroy())
                {
                    continue;
                }

                // shared_ptr のキャスト関数
                return std::dynamic_pointer_cast<T>(component);
            }
        }

        return std::weak_ptr<T>();
    }

    template<class T>
    void RemoveComponent()
    {
        for (auto& component : m_components)
        {
            if (typeid(*component.get()) == typeid(T))
            {
                // ここでは実際には消去しない
                component.get()->Destory();
            }
        }
    }

protected:

    // ゲームオブジェクトが持つ部品たち
    std::vector<std::shared_ptr<Component>> m_components;
```

コンポーネントを操作している部分です。

コンポーネントを実装してみて

簡単なサンプルでしたが、

コンポーネントシステムを実装してみました。

今回、遅延消去にしたお陰で、

別スレッドでの実行等出来ることも沢山増えたと思います。

またゲームオブジェクト等のUpdate関数の呼び出し部分に、

ジョブスケジューラーなんかを使うともっと使いやすくなると思います。

今回のような実装はコンポーネント指向と呼ばれるものでした。

しかし次の応用以降では、データ指向のコンポーネント管理について解説＆実装していきます。

コンポーネントステム（応用）

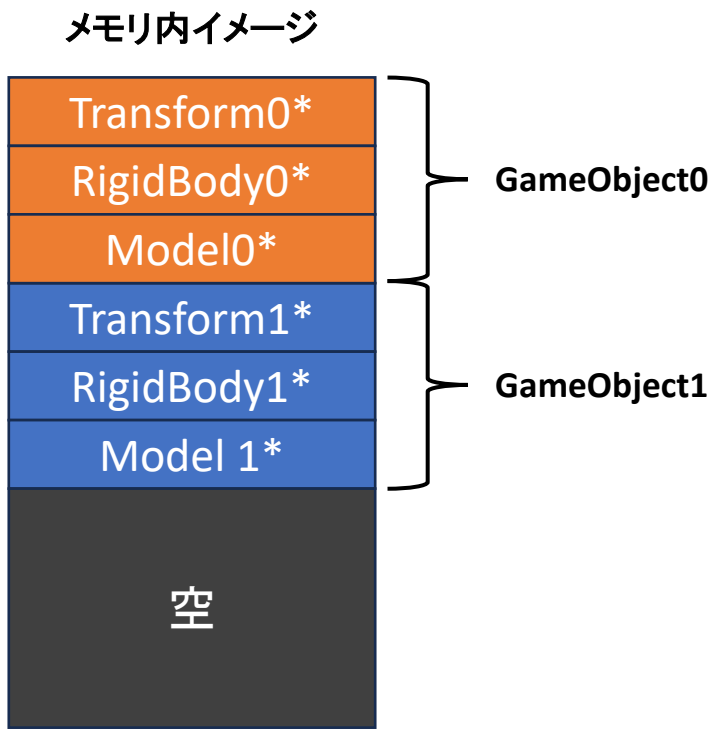
1. EntityComponentSystemについて解説

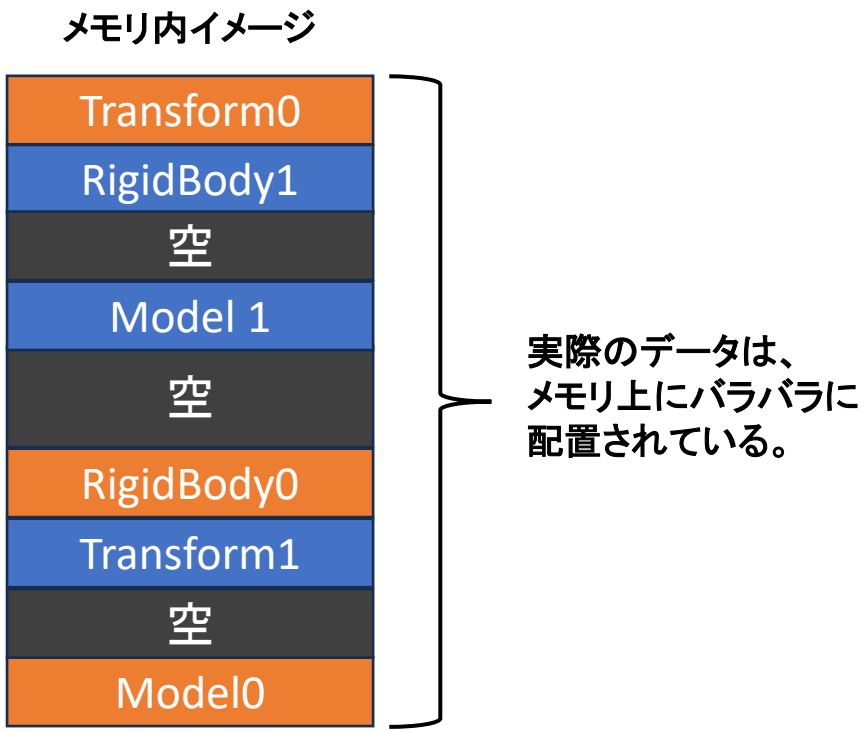
※難しい内容かもしれませんが、順に説明しますので頑張ってみてください。

EntityComponentSystem（ECS）って何？１（解説）

まずECSを解説する前に、

最初のサンプルのようなオブジェクト指向による管理方法の問題について解説します。



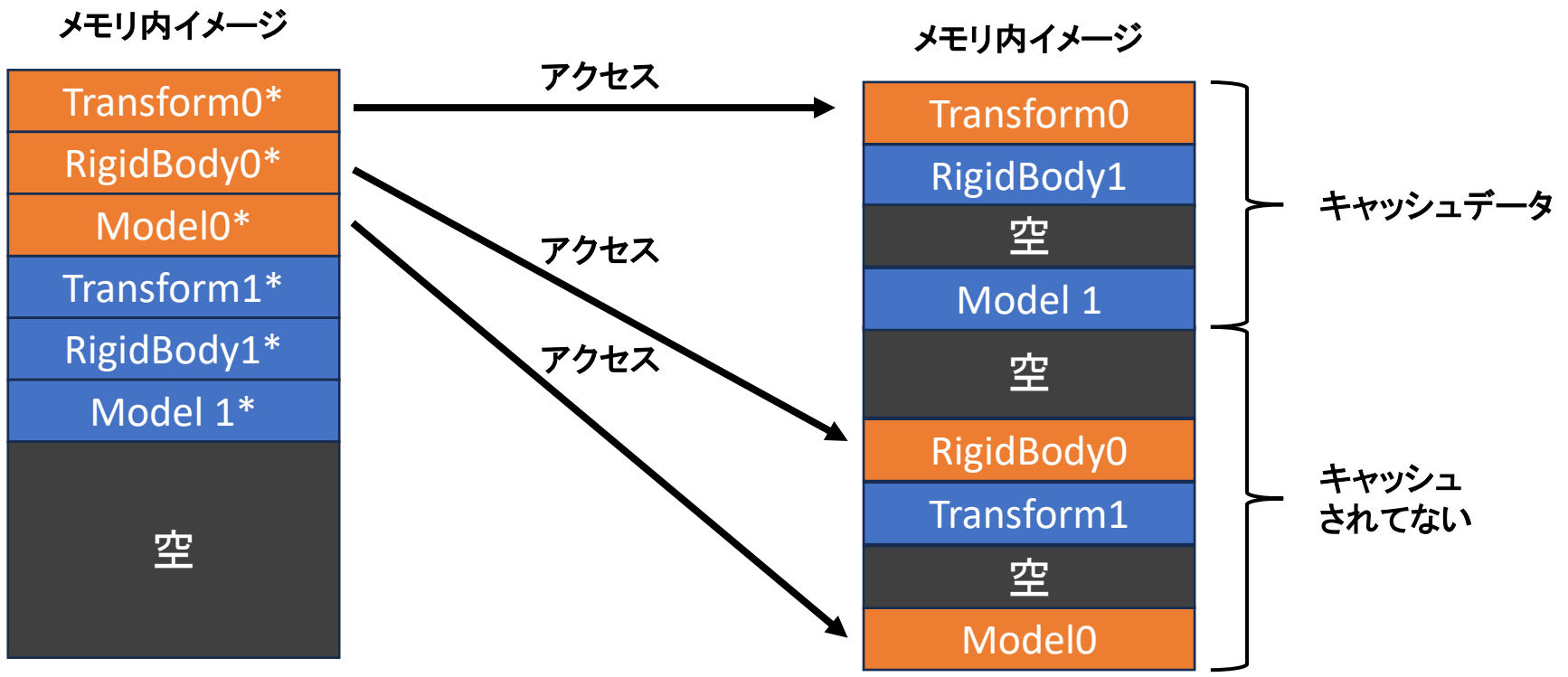


詳しくはこちら

EntityComponentSystem（ECS）って何？２（解説）

まずは、GameObject0の各コンポーネントにアクセスする場合を考えてみます。

実データへアクセスするためには、メモリ内を飛び回る必要があるのが分かると思います。



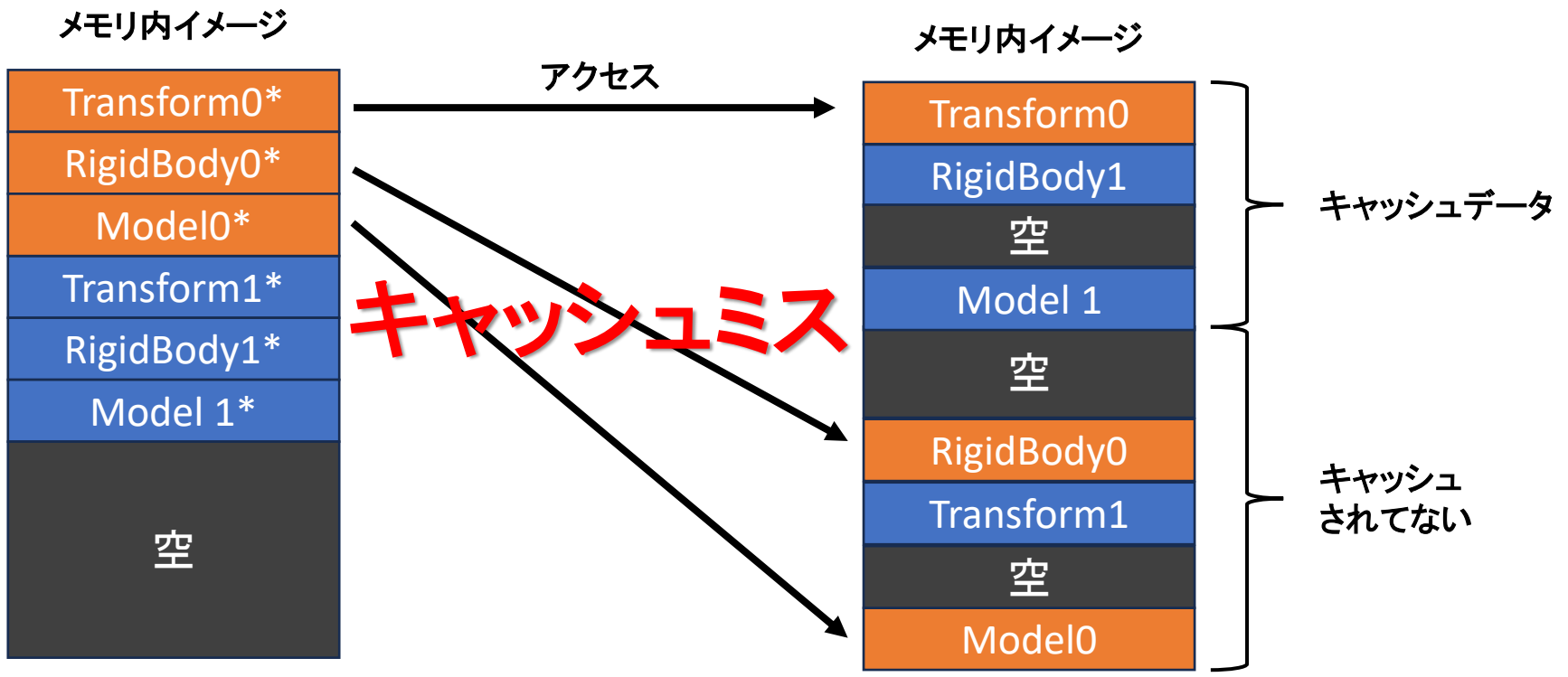
詳しくはこちら

EntityComponentSystem（ECS）って何？３（解説）

今回の場合は、メモリ内のキャッシュデータ には、現在Transform1しかありません。

そのため、次のRigidBody1にアクセスする前にキャッシュを再取得する必要があります。

※キャッシュの再取得は非常に遅く、ボトルネックになりやすいです。



詳しくはこちら

EntityComponentSystem（ECS）って何？４（解説）

先ほどのキャッシュミスによる、キャッシュの再取得の話は、
毎回起こることではなく、不連続的なメモリアクセスを沢山すると起こりやすくなります。

そのため、
一つのGameObject内だけの話であれば可能性は少なく、問題になりにくいです。

しかし大量のオブジェクトを処理するシステムクラスを想像してみるとどうでしょう？
キャッシュミスが起こる可能性が増え、ボトルネックになりやすいと思います。

詳しくはこちら

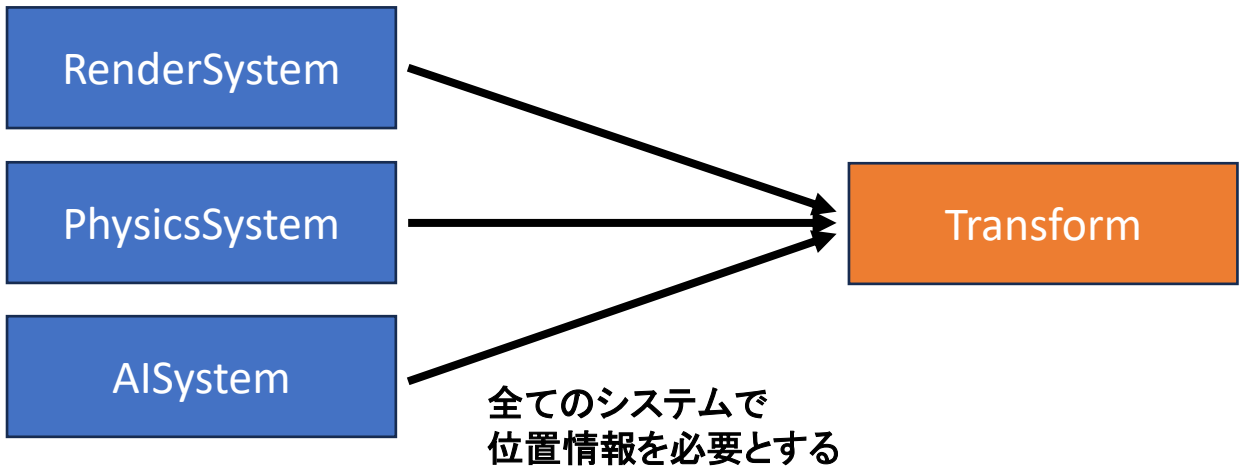
EntityComponentSystem（ECS）って何？５（解説）

そして、もう一つはオブジェクト指向プログラミングにおいて重要な考え方は、

「データはそのデータを操作するオブジェクトにカプセル化される」ということです。

しかし、ゲームでは沢山のシステムやクラスが同じデータを必要とすることがあります。

これらはカプセル化を壊し、コードを複雑にする可能性があります。



詳しくはこちら

EntityComponentSystem（ECS）って何？６（解説）

ECSは、先ほどまでの解説で出てきた二つの問題を解決してくれます。

そのため、

大量のオブジェクトを必要とするゲーム（AAAタイトルや3D弾幕シューティング等）も作成できます。

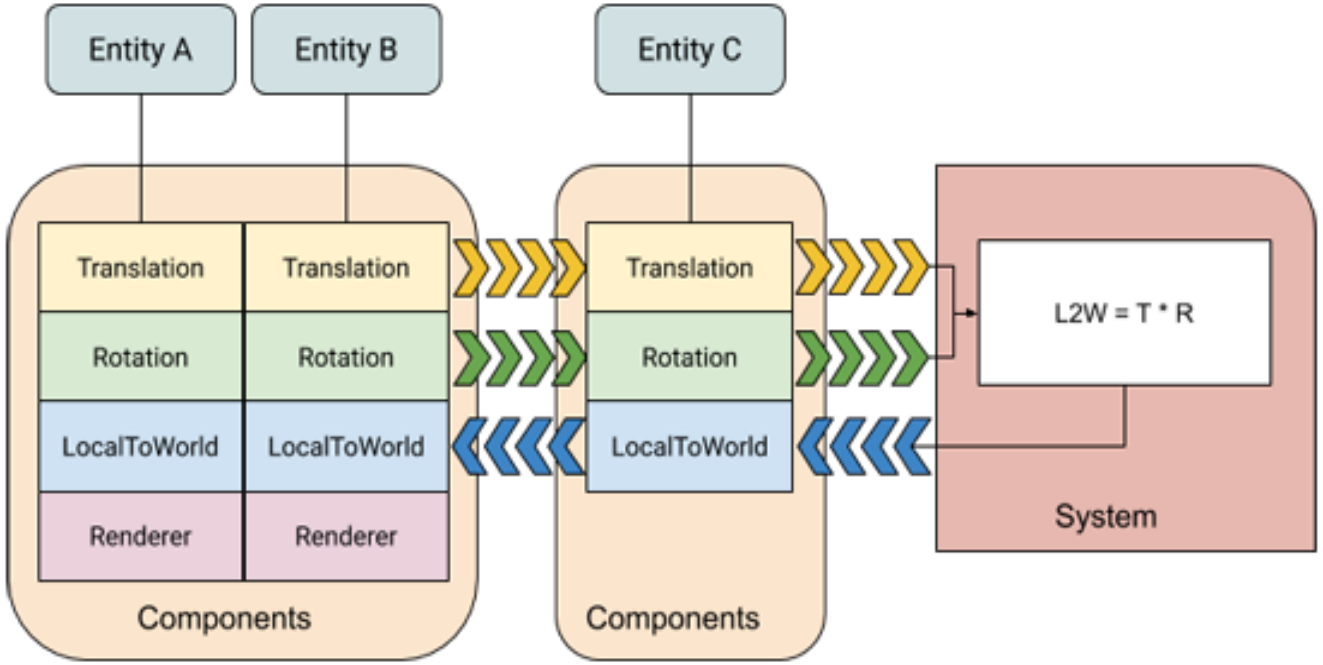
また、データと動作(システム)を分離しているため先ほどのような問題も起きません。

EntityComponentSystem（ECS）って何？7（解説）

そしてECSは名前の通り、

Entity（一意のID ）、Component（データ）、System（動作）、の三つの考えで作成されます。

下記は、その3っつの基本部分がどのように動作するかを表しています。



UnityのECS解説記事の画像

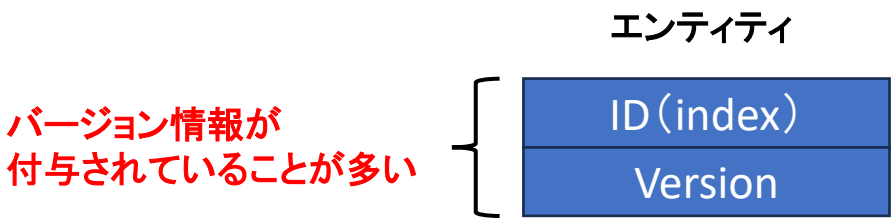
EntityComponentSystem（ECS）って何？８（解説）

ここからは先ほどの話をより詳しく解説していきます。

最初に、Entityは、

一意のIDを保持するクラスで、基本的にはIDを使用して各コンポーネントにアクセスしていきます。

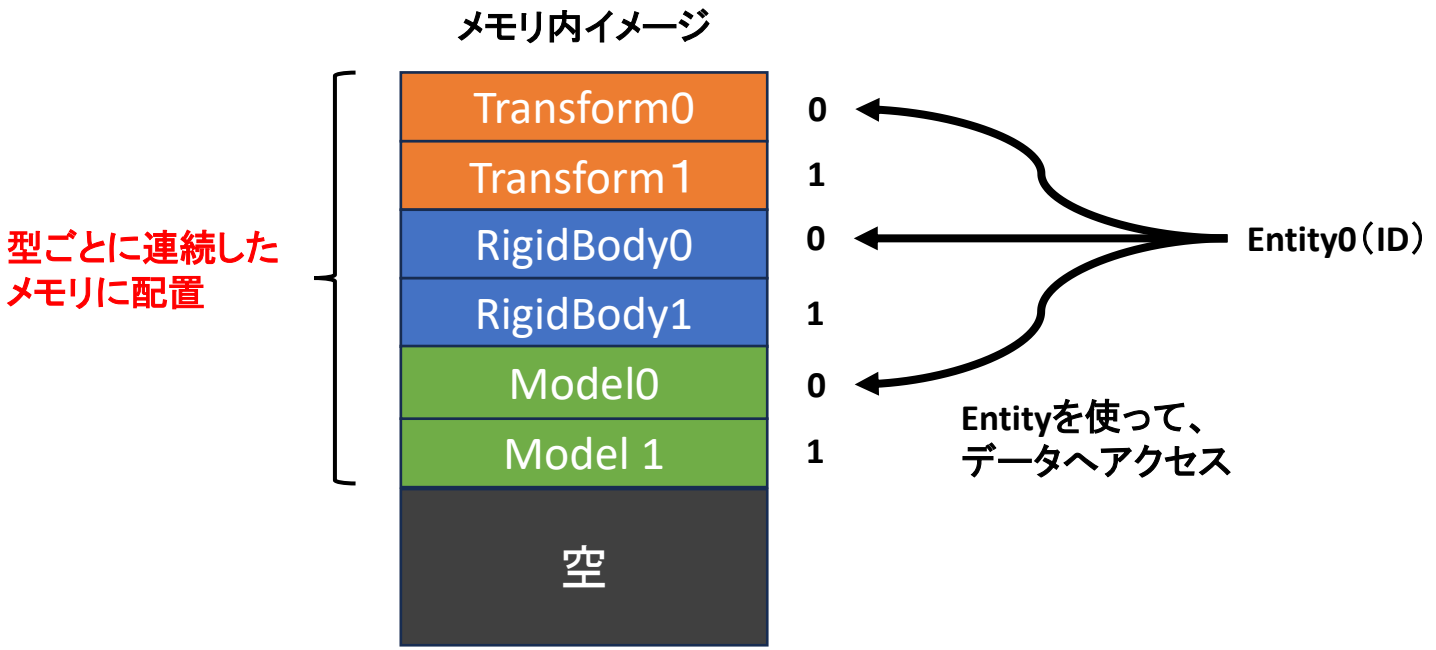
※UnityでいうGameObjectと似たようなものです。



EntityComponentSystem（ECS）って何？９（解説）

次に、コンポーネントは、

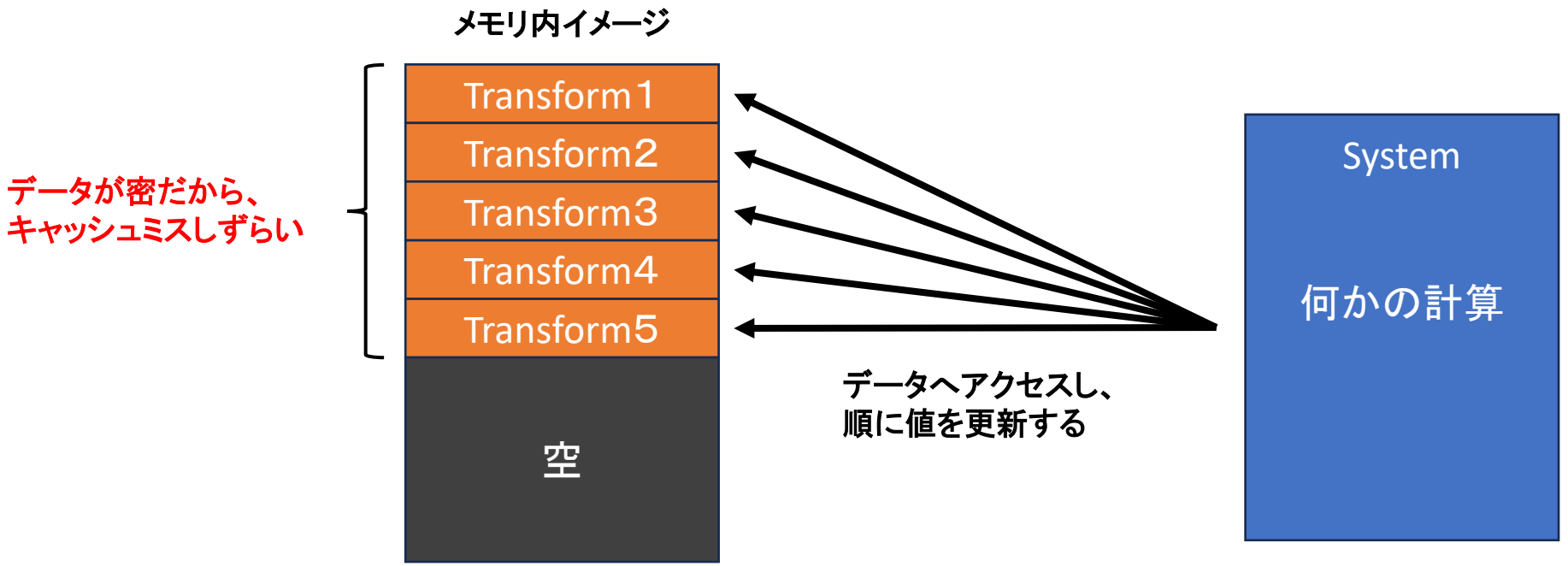
データだけを保持するクラスで、各型の連続したメモリに配置され、Entity（ID）を使うことでアクセス出来ます。



EntityComponentSystem（ECS）って何？10（解説）

そして、システムは、Entity（ID）を使用して各コンポーネントにアクセスし、処理を行います。

この時、データが連続して配置されているため、キャッシュミスが劇的に減ります。

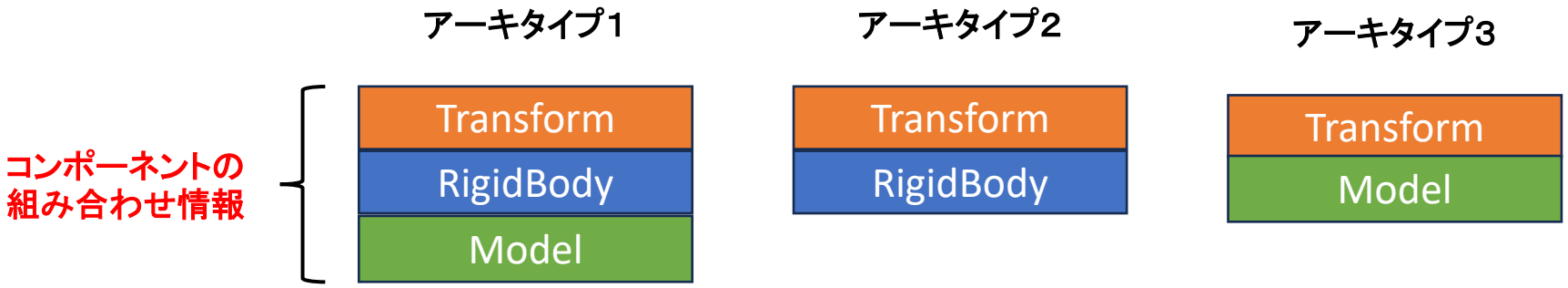


EntityComponentSystem（ECS）って何？１１（解説）

コンポーネントの一意の組み合わせは、アーキタイプ（Archetype）と呼ばれます。

同じコンポーネントのセットを共有するため、アーキタイプとして分類します。

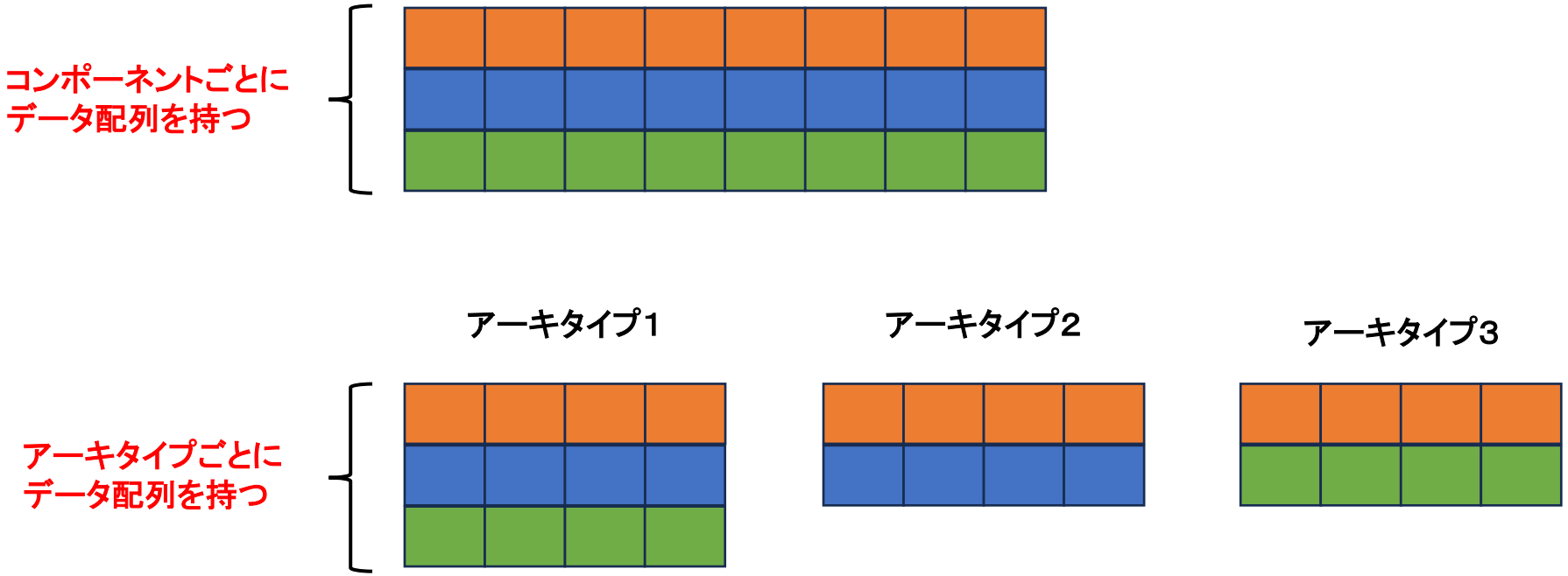
下記は3っつのアーキタイプを表しています。



EntityComponentSystem（ECS）って何？12（解説）

ECSは、アーキタイプの管理方法に、大きく二つアプローチ方法があります。

今回は下の方を実装します。（現在最も人気の方法なので）



EntityComponentSystem（ECS）って何？１３（解説）

さて、長々とした解説でしたが、

「ECSとは何ぞや？」ということは、分かっていただけたのではないでしょうか。

次はいよいよ実装にいこうと思います。

コンポーネントステム（より発展的）

1. EntityComponentSystemの実装

※難しい内容かもしれませんが、順に説明しますので頑張ってみてください。

ECSを実装する前の注意点

EntityComponentSystem（ECS）のアプローチ方法は本当に沢山あります。

なので今回、

私が実装した方法が良い方法というわけではないことを理解しておいてください。

またECSは元々、メリット操作やなんやらで処理が非常に複雑です。

今回はシンプルさを重視したため、

メモリプール等のめっちゃ細かい実装の所まではあまり触っていません。

しかし、最後に比較もしていますが、ECSの恩恵は十分受けられるモノにはしています。

ECSの実装で重要なコードだけを解説１（実装例）

#簡単な実装例

```
3
4   struct Transform
5   {
6       float x;
7       float y;
8       float z;
9   };
10
11  struct RigidBody
12  {
13      float speed;
14  };
15
```

コンポーネント（データのみ）を定義しています。

何も継承していませんがこれで大丈夫です。

ECSの実装で重要なコードだけを解説２（実装例）

#簡単な実装例

```
25    void Init() override
26    {
27        auto entity = m_entityManager->CreateEntity();
28        m_entityManager->AddComponent<Transform>(entity);
29        m_entityManager->AddComponent<RigidBody>(entity);
30
```

Entity（GameObjectの様なもの）を生成し、

先ほどのコンポーネントを追加しています。

ECSの実装で重要なコードだけを解説３（実装例）

#簡単な実装例

```
111
112     int main()
113     {
114         World world;
115
116         // システムを登録
117         world.RegisterSystem<TestSystem>();
118         world.RegisterSystem<PhysicsSystem>();
119
120         // システムの初期化
121         world.Init();
122
123         // ワールドの更新
124         for (auto i = 0; i < 10; i++)
125         {
126             world.Update(0.0166666f);
127         }
128     }
```

Worldに独自実装のシステムを登録することで、

独自の更新処理を追加していきます。

ECSの実装で重要なコードだけを解説 4（実装例）

#簡単な実装例

```
void UnInit() override {}

void Update(float deltaTime) override
{
    m_entityManager->ForEach<Transform, RigidBody>([&](Transform& trans, RigidBody& rb)
    {
        trans.x += rb.speed * deltaTime;
        printf("x = %f\n", trans.x);
    });
}
```

先ほどの物理システムの更新処理です。

ワールド内のTransformとRigidBodyだけを取得し、

速度に応じて、Transformの位置情報を更新しています。

ECSの実装で重要なコードだけを解説５（実装例）

#簡単な実装例

```
// ただのIDだけを保持（Unityでいうゲームオブジェクトに相当）
using Entity = size_t;
```

Entityは、IDだけを保持しています。（version情報×）

#簡単な実装例

```
Entity GenerateID()
{
    auto entity = Entity(m_entities.size());
    m_entities.emplace_back(entity);
    return entity;
}
```

下の画像は、Entityを生成するコードです。

今回はIndex番号をそのままIDとしています。

ECSの実装で重要なコードだけを解説６（実装例）

#簡単な実装例

```
3
4     using TypeID = size_t;
5
6     class TypeIDGenerator
7     {
8     public:
9
10        template<class T>
11        static TypeID GetID()
12        {
13            static TypeID typeId = m_typeCounter++;
14            return typeId;
15        }
16
17    private:
18
19        static inline size_t m_typeCounter = 0;
20    };
21
22    // コンポーネントタイプの組み合わせ情報を保持
23    using Archetype = std::vector<TypeID>;
24
```

アーキタイプの定義部分です。

std::vectorでタイプの組み合わせを保持しているのと、

そのTypeIDを生成するコードを作っています。

（typeid（T）でも良いです、、、）

ECSの実装で重要なコードだけを解説 7（実装例）

#簡単な実装例

```
// 指定型を保持した、ArchetypeIDを生成＆取得（ArchetypeIDの初期値を設定出来るようにしてる）
template<typename... Args>
Archetype GetArchetype(const Archetype& archetype = {})
{
    Archetype newArchetype { TypeIDGenerator::GetID<Args>()... };

    // 初期値が指定されていれば追加
    if (!archetype.empty())
    {
        for (auto& typeId : archetype)
        {
            newArchetype.emplace_back(typeId);
        }
    }

    // 昇順に並べ替え（比較処理を簡単にするため）
    std::sort(newArchetype.begin(), newArchetype.end());

    return newArchetype;
}
```

そして、アーキタイプの生成関数です。

可変長テンプレートで指定された、
全ての型を持つアーキタイプを生成しています。

初期値として設定出来るように引き数を追加しています。

ECSの実装で重要なコードだけを解説８（実装例）

#簡単な実装例

```cpp
// ComponentArrayがテンプレートクラスであるため、
// 通常は、静的タイミングでしかクラスを定義出来ない。（ComponentArray<T> test; みたいな）
// しかし、動的にComponentArrayの型を変更出来る用にしたかったので、このクラスを作成。
struct IComponentArray
{
    virtual ~IComponentArray() = default;

    virtual std::unique_ptr<IComponentArray> Create() = 0;

    virtual void Remove(size_t id) = 0;

    virtual bool Has(size_t id) = 0;

    virtual void Move(size_t id, IComponentArray* components) = 0;
};

template<class T>
struct ComponentArray : public IComponentArray
{
private:
    static inline size_t nullPosition { std::numeric_limits<Entity>::max() };
```

```cpp
    auto begin() { return m_datas.begin(); }
    auto end() { return m_datas.end(); }

private:

    // コンポーネントデータ配列（DenseIndexでアクセスする）
    std::vector<T> m_datas;

    // DenseIndex -> ID に変換
    std::vector<size_t> m_toSparse;

    // ID -> DenseIndex に変換（これをMapで管理するとビックリするくらい重くなるので試してみてください）
    std::vector<size_t> m_toDense;
};
```

ComponentArrayは、

実際にコンポーネントデータを保持し、操作する人達です。

IComponentArrayとComponentArray<T>を使って、

型を気にすることなく、操作出来るようにしています。

ECSの実装で重要なコードだけを解説９（実装例）

#簡単な実装例

```
#include "ComponentArray.h"

// 同じArchetypeを持つEntityのコンポーネントデータを保持
struct ArchetypeChunk
{
    using ComponentDataMap = std::map<TypeID, std::unique_ptr<IComponentArray>>;
public:

    ArchetypeChunk(const Archetype& archetype) { ... }

public:

    //====================================
    // Component Methods
    //====================================

    // entityのコンポーネントデータを追加
    template<class T, typename... Args>
    T& Add(Entity entity, Args&& ...args)
    {
        return GetContainer<T>()->Add(entity, std::forward<Args>(args)...);
    }

    // entityのコンポーネントデータを取得
    template<class T>
    T& Get(Entity entity) { ... }
```

そしてこちらがアーキタイプチャンククラスです。

アーキタイプと、コンポーネントデータ配列を保持し、
データ配列とクライアントの橋渡しのようなことをします。

ECSの実装で重要なコードだけを解説 10（実装例）

#簡単な実装例

```
//===================================

template<class T, typename... Args>
T& AddComponent(Entity entity, Args&& ...args)
{
    assert(IsValid(entity));
    auto oldChunk = m_entityChunkMap[entity];
    auto archetype = GetArchetype<T>(oldChunk);

    // 新しいデータ格納用チャンクを取得
    auto newChunk = GetOrAddChunk(archetype);
    assert(!newChunk.expired());

    // 追加された直後の場合はセットアップ
    auto initialized = newChunk.lock()->HasComponentMap();
    if (!initialized)
    {
        newChunk.lock()->SetUpComponentMap<T>(archetype, oldChunk);
    }

    // データ保存先が変わる時の共通処理を実行
    MoveChunkData(entity, newChunk, oldChunk);

    // 実際に AddComponent を実行
    return newChunk.lock()->Add<T>(entity, std::forward<Args>(args)...);
}
```

これが実際のコンポーネント追加関数です。

非常に難しそうですがやっていることはシンプルです。

過去のアーキタイプのデータを消去し、

Add後の新しいアーキタイプにデータを移しています。

それが完了してから実際に新しいデータを追加します。

ECSを実装してみて

実装で重要な部分をいくつか解説しましたが、
正直、ここで色々言っても分からないと思います。

なのでGitにあるサンプルプログラムを実際に見てほしいと思います。

勿論、全部理解する必要はありませんが「どういうものなのか？」というのは、
知っておいて損はないと思います。

最後に各コンポーネントシステムを比較してみた（実装結果）

#ノーマル

```
finish 388.858800 milli
```

メモリ使用量：2.6GB

#ECS(サンプル)

```
finish 104.813800 milli
```

メモリ使用量：1.8GB

#ECS（別）

```
finish 43.470200 milli
```

メモリ使用量：1.4GB

1000万体のオブジェクトの更新を比較した結果です。

コンパイルツールにはv142(VS2019)を使用しています。

通常のコンポーネントシステム（最初のサンプル）、

サンプルと別（これも自作）のECS2種類を比較しました。

サンプルのECSでも通常のコンポーネントシステムより、

約3．8倍ほど処理速度が上がっていることからも、

ECSの強力さが実感できますよね。

コンポーネントのまとめ

今回は結構難しくなってしまいましたが、

簡単なコンポーネントシステムと、ECSの実装が出来たと思います。

これから、ECSはデフォルトスタンダードになっていくので、

知っておいて損はないと思います。

ですが正直、ECSは、

「どんなものなのか？」ということ分かっていれば十分です。

ですがもし、3D弾幕などのゲームを作ろうとしているなら

作ってみる価値は十分ありますので挑戦してみてください。（多分一番効果が出る）