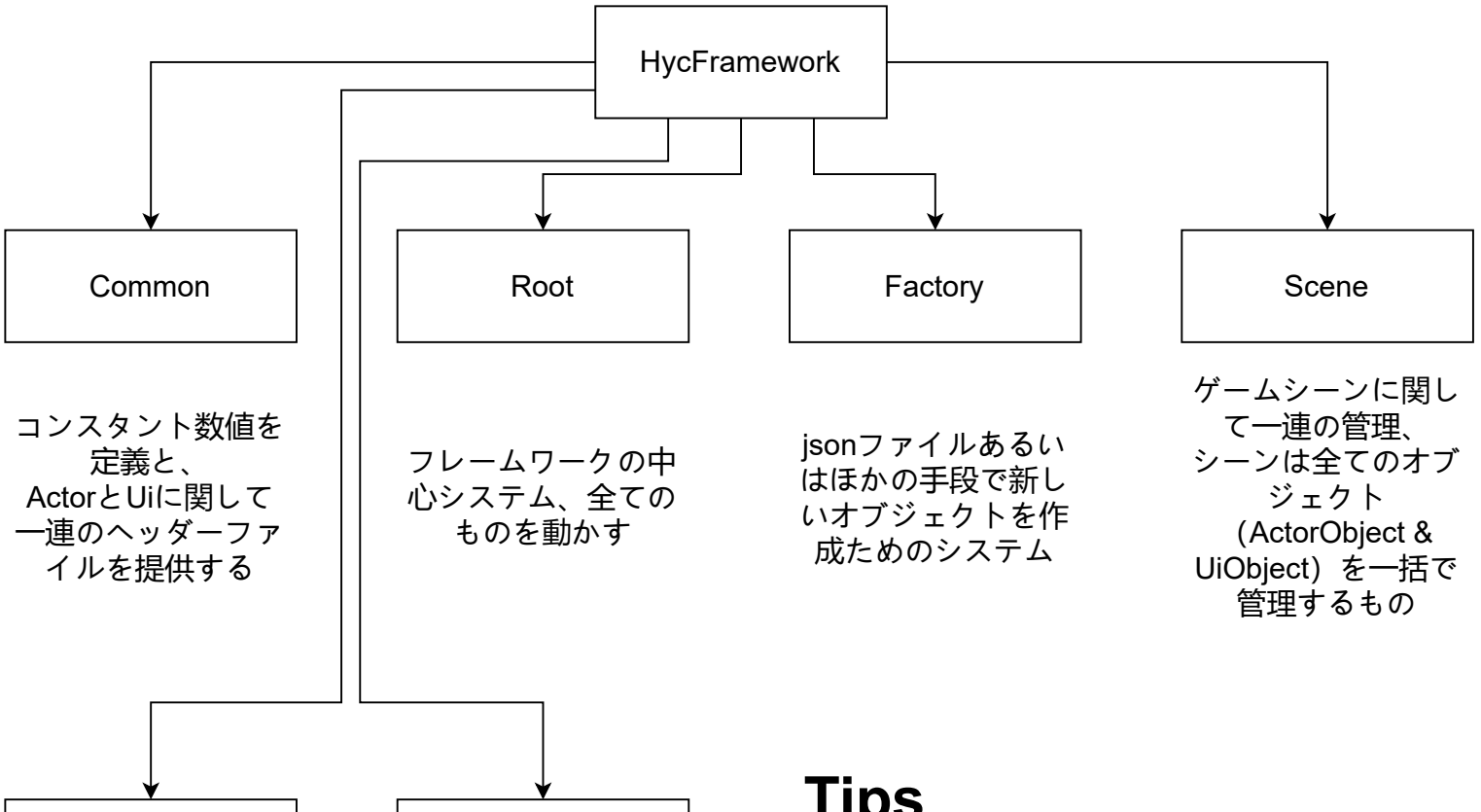
HycFrame2D全体総覧

フレームワーク本体：02_FrameContent	ゲームシステムの管理
中層システム：01_MiddleFunc	コントローラー、サウンド、とほかのツール関数
基本システム：00_BasicFunc	ウィンドウとDirectX関連の管理

便利なところ

1　管理しやすい、チーム開発での役割分担も容易になる
2　初ビルドや再ビルドしない限り、コンパイルが速い
3　JSONファイルを使ってシーンを組み立てる、細かい調整が楽
4　既に用意された豊かな機能

フレームワーク総覧



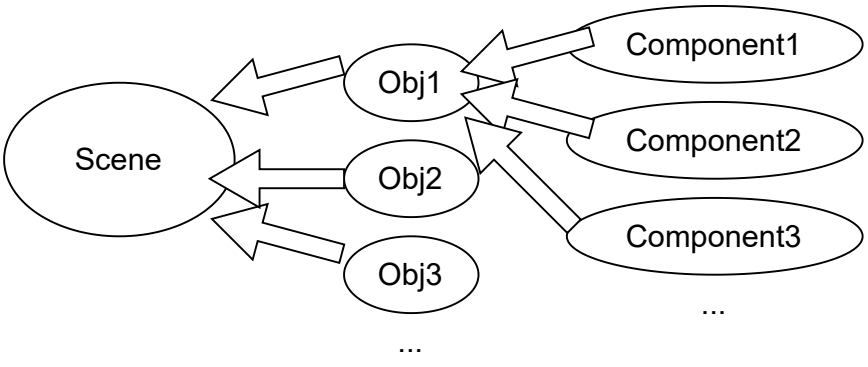
Tips

このプロジェクトのコードはガチなオープンソースプロジェクトよりけっこう読みやすいだと思います、もし何か不具合が出てきたら、できればまず不具合のところのコードをチェックしましょう!

※　重要！！

Scene Object Componentをどう組み立てるのはゲーム世界がどう作動するを決める一番大事な要素。

Unityに代入して認識すれば分かりやすいかもしれません。

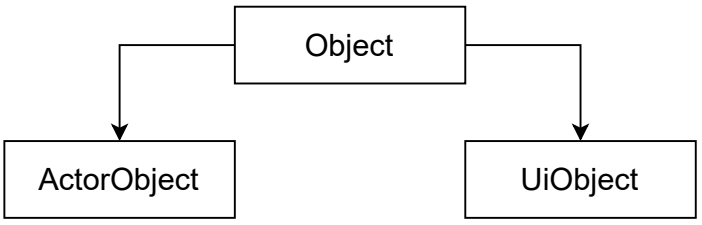


もしComp1がSpriteComponentだとしたら、これよりスプライートを描画できる

もしComp2がTextComponentだとしたら、これより文字を描画できる

もしComp3がTransformComponentだとしたら、これよりオブジェクト所在の位置を変更できる

オブジェクトに関して



全てのObjectが使える関数

std::string GetObjectName(); このオブジェクトの名前を返却する

STATUS IsObjectActive(); & void SetObjectActive(STATUS); このオブジェクトの状態を確定＆設定する（STATUSは02_FrameContent/Common/HFCommon.hの19行を確かめてください）

SceneNode* GetSceneNodePtr(); このオブジェクト所属しているシーンのポインタを取得

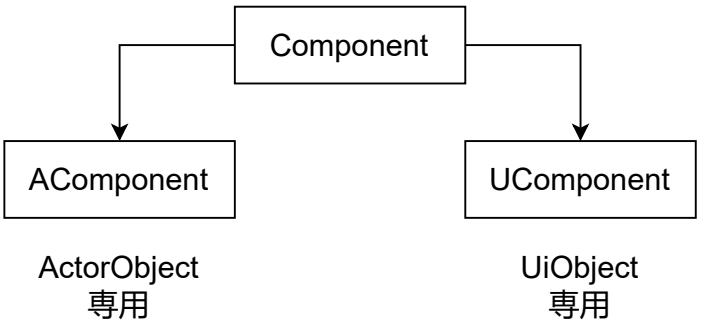
ActorObjectが使える関数（基本）

AComponent* GetAComponent(std::string); & template T* GetAComponet<T>(COMP_TYPE); このオブジェクトが持っている全てのコンポーネントから特定のコンポーネントを取り出す。一番目の関数はコンポーネントの名前より取り出す、でも基本クラスのポインタしか戻されないので、あとで必要なコンポーネント型に変更するが必要。二番目の関数はテンプレートより種類が確定されているコンポーネントを取り出す、テンプレートを利用しているので、型が自動的に変更されています

UiObjectが使える関数（基本）（Actorとほぼ同じ）

UComponent* GetUComponent(std::string); & template T* GetUComponet<T>(COMP_TYPE); このオブジェクトが持っている全てのコンポーネントから特定のコンポーネントを取り出す。一番目の関数はコンポーネントの名前より取り出す、でも親クラスのポインタしか戻されないので、あとで必要なコンポーネント型に変更するが必要。二番目の関数はテンプレートより種類が確定されているコンポーネントを取り出す、テンプレートを利用しているので、型が自動的に変更されています

コンポーネントに関して



全てのComponentが使える関数

std::string GetComponentName(); このコンポーネントの名前を返却する

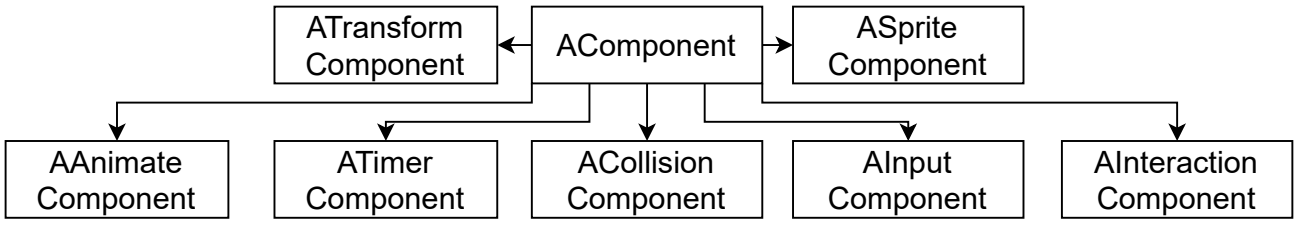
STATUS IsCompActive(); & void SetCompActive(STATUS); このコンポーネントの状態を確定＆設定する（STATUSは02_FrameContent/Common/HFCommon.hの19行を確かめてください

AComponentが使える関数（基本)

ActorObject* GetActorObjOwner(); このコンポーネントが所属してるActorObjectを取得

UComponentが使える関数（基本)

UiObject* GetUiObjOwner(); このコンポーネントが所属してるUiObjectを取得



ATransformComponentが使える関数（基本)

ACTORオブジェクトに動く位置を判定するのコンポーネント

void SetPosition(Float3 _pos); & Float3 GetPosition(); このコンポーネントが所属してるActorObjectの位置をｘｙｚ軸の順で設定、取得する
void SetPosition(Float3 _pos); & Float3 GetPosition(); このコンポーネントが所属してるActorObjectのスピン角度をｘｙｚ軸の順で設定、取得する
void SetPosition(Float3 _pos); & Float3 GetPosition(); このコンポーネントが所属してるActorObjectの大きさをｘｙｚ軸の順で設定、取得する

void Translate(Float3 _pos); & void TranslateXAsix(float _posx); & void TranslateYAsix(float _posy);
現時点の位置を基づいて、更に渡された方向と距離に移動する
void Rotate(Float3 _angle); & void RotateXAsix(float _anglex); & void RotateYAsix(float _angley);
現時点のスピン角度を基づいて、更に渡された方向と距離に移動する
void Scale(Float3 _factor); & void ScaleXAsix(float _factorx); & void ScaleYAsix(float _factory); 現時点の大きさを基づいて、更に渡された方向と距離に移動する

ASpriteComponentが使える関数（基本）

ACTORオブジェクトにスプライト表示に関してのコンポーネント

void SetOffsetColor(Float4 _color); このコンポーネントが保存しているテクスチャを描画する時、各ピクセルの色にこのoffsetColorをかけていく
Float4 GetOffsetColor(); 現時点のoffsetColorを取得

※　もし途中でテクスチャを変更したいなら、AAnimateComponentで実装してください。そこにリピートしないスプライートを追加してAnimateとして表示させたらオッケー。

AAnimateComponentが使える関数（基本）

ACTORオブジェクトにアニメーション表示するためのコンポーネント

void ChangeAnimateTo(std::string _name); スプライートアニメを名前より指定する

ATimerComponentが使える関数（基本）

ACTORオブジェクトに時間測定に関してのコンポーネント

void StartTimer(std::string _name); 名前より応じているタイマーを起動する
void PauseTimer(std::string _name); 名前より応じているタイマーを一時停止する
void ResetTimer(std::string _name); 名前より応じているタイマーをリセットする
Timer* GetTimer(std::string _name); 名前より応じているタイマーを取得する
bool Timer::IsGreaterThan(float _value); 特定のタイマーの時間と与えられた数値と比較する

ACollisionComponentが使える関数（基本）

ACTORオブジェクトにあたる判定に関してのコンポーネント

bool CheckCollisionWith(ActorObject* _obj); 与えられたActorObjectと当たり判定を行う

AInputComponentが使える関数（基本）

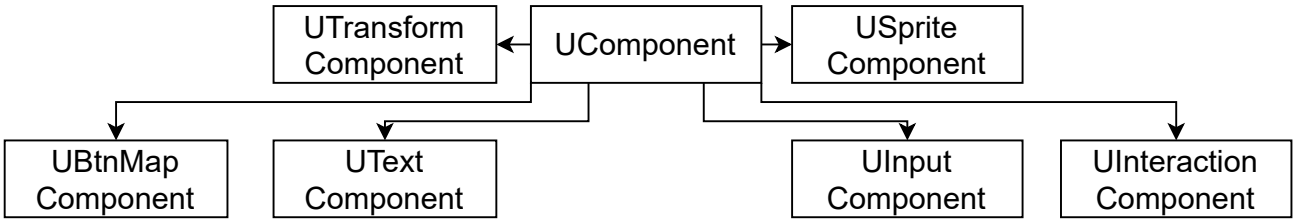
ACTORオブジェクトに入力処理スクリプト関数に関してのコンポーネント

※　自分が書けた関数をこのフレームワークで呼び出させる仕組みの一つ、本体の関数は基本的に使う必要ない。詳しい説明は json-sceneファイルの書き方とスクリプト関数の部分で確認お願いします。

AInteractionComponentが使える関数（基本）

ACTORオブジェクトに自身特有スクリプト関数を呼び出すためのコンポーネント

※　自分が書けた関数をこのフレームワークで呼び出させる仕組みの一つ、本体の関数は基本的に使う必要ない。詳しい説明は json-sceneファイルの書き方とスクリプト関数の部分で確認お願いします。



UTransformComponentが使える関数（基本）

UIオブジェクトに動く位置を判定するのコンポーネント

void SetPosition(Float3 _pos); & Float3 GetPosition();　このコンポーネントが所属してるUiObjectの位置をｘｙｚ軸の順で設定、取得する
void SetPosition(Float3 _pos); & Float3 GetPosition();　このコンポーネントが所属してるUiObjectのスピン角度をｘｙｚ軸の順で設定、取得する
void SetPosition(Float3 _pos); & Float3 GetPosition();　このコンポーネントが所属してるUiObjectの大きさをｘｙｚ軸の順で設定、取得する

void Translate(Float3 _pos); & void TranslateXAsix(float _posx); & void TranslateYAsix(float _posy);
現時点の位置を基づいて、更に渡された方向と距離に移動する
void Rotate(Float3 _angle); & void RotateXAsix(float _anglex); & void RotateYAsix(float _angley);
現時点のスピン角度を基づいて、更に渡された方向と距離に移動する
void Scale(Float3 _factor); & void ScaleXAsix(float _factorx); & void ScaleYAsix(float _factory);　現時点の大きさを基づいて、更に渡された方向と距離に移動する

USpriteComponentが使える関数（基本）

UIオブジェクトにスプライト表示に関してのコンポーネント

void SetOffsetColor(Float4 _color);　このコンポーネントが保存しているテクスチャを描画する時、各ピクセルの色にこのoffsetColorをかけていく
Float4 GetOffsetColor();　現時点のoffsetColorを取得

※　途中でテクスチャを変更するのは原則的お勧めではない

UBtnMapComponentが使える関数（基本）

UIオブジェクトにボタンを地図のように操作するコンポーネント

bool IsBeingSelected();　このボタンが選択されているかを判定

void SelectUpBtn(); & void SelectDownBtn(); & void SelectLeftBtn(); & void SelectRightBtn(); このボタンの上下左右ボタンを選択する
UiObject* GetUpBtn(); & UiObject* GetDownBtn(); & UiObject* GetLeftBtn(); & UiObject* GetRightBtn(); このボタンの上下左右ボタンを取得

UTextComponentが使える関数（基本）

UIオブジェクトに文字表示するためのコンポーネント

void SetTextPosition(Float3 _pos); テキストの位置を変更
void SetFontSize(Float2 _size); テキストの大きさを変更
void ChangeTextString(std::string _text); テキストの内容を変更する
void SetTextColor(Float4 _color); テキストの色を変更する

UInputComponentが使える関数（基本）

UIオブジェクトに入力処理スクリプト関数に関してのコンポーネント

※　自分が書けた関数をこのフレームワークで呼び出させる仕組みの一つ、本体の関数は基本的に使う必要ない。詳しい説明は json-sceneファイルの書き方とスクリプト関数の部分で確認お願いします。

UInteractionComponentが使える関数（基本）

UIオブジェクトに自身特有スクリプト関数を呼び出すためのコンポーネント

※　自分が書けた関数をこのフレームワークで呼び出させる仕組みの一つ、本体の関数は基本的に使う必要ない。詳しい説明は json-sceneファイルの書き方とスクリプト関数の部分で確認お願いします。

シーンに関して

SceneNode

一つのシーンを
SceneNodeの形で保存、
作動している

SceneManager

複数のシーンを一括で管
理するシステム

Camera

一つシーンに一つのカメ
ラが付いている、全て
ACTORオブジェクトの描
画はカメラベースで行う

SceneNodeが使える関数（基本）

単一のシーンを管理、代表するの節点

std::string GetSceneName(); このシーンの名前を取得
SceneManager* GetSceneManagerPtr() ; SceneManagerのポインタを取得
Camera* GetCamera(); このシーン所属のカメラを取得

ActorObject* GetActorObject(std::string _name); 名前よりこのシーンにあるACTORオブジェクトを取得
UiObject* GetUiObject(std::string _name); 名前よりこのシーンにあるUIオブジェクトを取得

Cameraが使える関数（基本）

視点と視界を定義するもの、全てのACTORオブジェクトは世界座標ではなくて、Cameraとの相対的座標で描画されている

void ResetCameraPos(Float2 _pos); カメラの位置をリセット
void TranslateCameraPos(Float2 _deltaPos); 現時点の位置よりカメラを移動させる
Float2 GetCameraPosition(); 現時点カメラの位置を取得

SceneManagerが使える関数（基本）

シーン管理する機能の対応

void SetShouldTurnOff(bool _value); ゲームを止めるかどうかを設定
void LoadSceneNode(std::string _name, std::string _path); 次のシーンをろーどする、引数1はシーンの名前、引数2はシーンjsonファイルのパス

JSONファイルの書き方

JSONとは? JSON入門 - とほほのWWW入門 (tohoho-web.com)

特別ファイル

- rom:/Configs/moji.json　誤って削除しないで!

- rom:/Configs/entry-scene.json　入口シーンを指定ためのファイル、中に入口にしたいシーンの名前とパスを記入してください。こちも誤って削除しないで!

- rom:/Configs/Scenes/load-scene.json　ローディングシーンを定義しているファイル、正しい書き方を沿って編集はできるが、削除は絶対ダメ!

シーンファイルの書き方

基本的には、UTF-8で記入、保存してくださいね（with BOM(signature)なしの方）

命名規則はソリューションファイルと同じフォルダ内にあるobj-name-rule.txtを参照してお願いします（コンポーネントの命名はしなくても大丈夫）

-- 最もシンプルなシーンファイルはこのように：実行すると背景の緑色以外何も見えないはずです

```
{
    "scene-name": "basic-scene",
    "actor": [],
    "ui": []
}
```

"scene-name"はつまりシーンの名前、シーンをロードすると一番の引数として渡すもの。ロード関数を使用する時、もし渡された名前はJSONファイル内の名前がずれたら、ロードは失敗する。
"actor"はACTORオブジェクトを定義する場所、"ui"はUIオブジェクトを定義する場所。今は空っぽなので、シーンの中には何もない

-- ACTORを二つ、UIを一つ追加なら、こう書くべき：

```
{
    "scene-name": "basic-scene",
    "actor": [
        {
            "actor-name": "test-1-actor",
            "update-order": 0,
            "parent": null,
            "components": []
        },
        {
            "actor-name": "test-2-actor",
            "update-order": 0,
            "parent": null,
```

```
            "components": []
        }
    ],
    "ui": [
        {
            "ui-name": "test-ui",
            "update-order": 0,
            "parent": null,
            "components": []
        }
    ]
}
```

つまり"actor"と"ui"の大かっこの中に中かっこで定義されているものを入れる。"actor/ui-name"はこのオブジェクトの名前を定義するところ、同一シーン内には複数の同じ名前持っているACTOR/UIオブジェクトが存在しないように注意してください。

"update-order"はこのACTOR/UIオブジェクトの更新順番を定義している、シーンシステムはいつも小さいものから大きいものまで更新する、同じ順番だとしたらランダムで行う。もし一つのオブジェクトはあるオブジェクト更新した後で更新したいなら、この"update-order"をぜひ活用してください。定義された順番と関係なく、ACTORオブジェクトの更新はいつもUIオブジェクの前で行う。

"parent"は同じ種類のオブジェクト間で親子関係を繋げる。原則的にはnullでいいです。

"component"はこのオブジェクトが持つべきコンポーネントを定義するところ。どう書くべきか後で説明します。

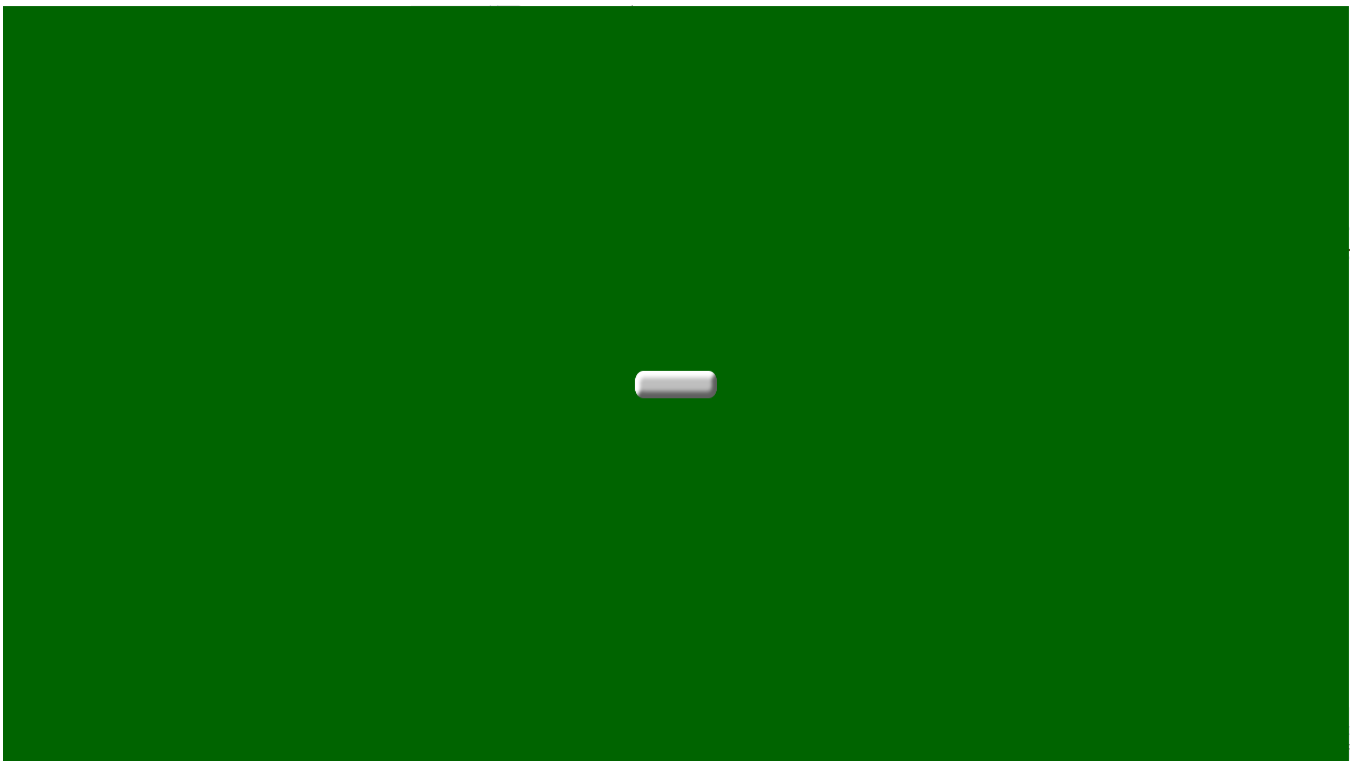
-- UIにテクスチャを表示すると、こう書くべき:

```
            "ui-name": "test-ui",
            "update-order": 0,
            "parent": null,
            "components": [
                {
                    "type": "transform",
                    "update-order": -1,
                    "init-value": [
                        0.0,
                        0.0,
                        0.0
                    ],
                    "position": [
                        null,
                        null,
                        null
                    ],
                    "rotation": [
                        null,
                        null,
                        null
                    ],
                    "scale": [
                        null,
                        null,
                        null
                    ]
```

```
            },
            {
                "type": "sprite",
                "update-order": 0,
                "draw-order": -2,
                "texture-path": "rom:/Assets/Textures/player.png",
                "texture-width": 150.0,
                "texture-height": 50.0
            }
        ]
```

つまり"components"のところに二つのコンポーネントを追加すればよい、一つはUTransformComponent、位置を定義する、もう一つはUSpriteComponent、表示するテクスチャを定義する。

実行するとこうなっているはずです。



-- 今私たちはまだデフォルトなカメラを使用している、できれば各シーンファイル内は明確にカメラを定義して方がいいと思います。定義するとこう書くべき:

```
    "scene-name": "basic-scene",
    "camera": [
        100.0,
        100.0,
        1920.0,
        1080.0
    ],
    "actor": [
```

前の二つ数値はカメラの位置を定義している、つまり今カメラの座標は[100 , 100]である、2D向けのフレームワークなので、 z 軸はいらないです。

後ろの二つ数値はカメラの視界、いつも1920 1080で設定して、変更なしでよいです。

-- 音声追加はこう書くべき:

```
    "scene-name": "basic-scene",
    "camera": [
        100.0,
        100.0,
        1920.0
```

```
            1080.0
        ],
        "sound": [
            {
                "name": "test-bgm",
                "path": "rom:/Assets/Sounds/bgm-success.wav"
            }
        ],
        "actor": [
```

"sound"の大かっこ内に中かっこで囲んでいる内容を必要な分を追加すればいい、"name"はゲーム内にこの音声唯一の名前、後で音声操作の部分で紹介させてください。

書き方は大体こうなっています。シーンに名前つけて、必要なオブジェクトを追加して、各オブジェクトに必要なコンポーネントを追加すれば完成。

各コンポーネントの書き方

ATransformComponent / UTransformComponent

"type" : "transform"と記入してください
"update-order" : 更新順番、より小さいとコンポーネントの間で
より早く更新される（この後は説明しません）
"init-value" : 普段はいつも 0.0, 0.0, 0.0で記入すればよい

"position" : 世界座標より位置の初期値、ｘｙｚ軸の順に記入
nullだったら全部0.0で初期化する

"rotation" : 世界座標よりスピン角度の初期値、ｘｙｚ軸の順に
記入、2Dなので、回したいならｚ軸にある数値を記入
してくださいね
nullだったら全部0.0で初期化する

"scale" : ｘｙｚ軸方向に大きさを調整できる、2Dなので、ｘｙ軸
だけで記入してください
nullだったら全部1.0で初期化する

```
{
    "type": "transform",
    "update-order": -1,
    "init-value": [
        0.0,
        0.0,
        0.0
    ],
    "position": [
        100.0,
        0.0,
        0.0
    ],
    "rotation": [
        null,
        null,
        null
    ],
    "scale": [
        null,
        null,
        null
    ]
},
```

ASpriteComponent / USpriteComponent

```
{
    "type": "sprite",
    "update-order": 0,
    "draw-order": -2,
    "texture-path": "rom:/Assets/Textures/player.png",
    "texture-width": 150.0,
    "texture-height": 50.0
},
```

※　ATransformComponent / UTransformComponentが必要

"type" : "sprite"と記入してください

"draw-order" : 描画順番、より小さいとより早く描画される、後で描画されるテクスチャにかぶる可能性があります

"texture-path" : テクスチャのパス、pngあるいはjpgでお願いします

"texture-width" : テクスチャが描画される時の幅さ

"texture-height" : テクスチャが描画される時の高さ

AAnimateComponent

```
{
    "type": "animate",
    "update-order": 0,
    "animates": [
        {
            "animate-name": "number",
            "animate-path": "rom:/Assets/Textures/number.png",
            "animate-stride": [
                0.2,
                0.2
            ],
            "max-count": 13,
            "repeat-flag": false,
            "frame-time": 1.0
        },
        {
            "animate-name": "run",
            "animate-path": "rom:/Assets/Textures/runman.png",
            "animate-stride": [
                0.2,
                0.5
            ],
            "max-count": 10,
            "repeat-flag": true,
            "frame-time": 0.1
        }
    ],
    "init-animate": "number"
}
```

※　ASpriteComponentが必要、本来のテクスチャはいらないでもつけるのは必要、empty.pngでもいい

"type" : "animate"と記入してください

"animates" : このオブジェクトに付けたいスプライート動画を定義する、例えば「stand」「run」とかな感じ

"animate-name" : このスプライート動画の名前、後でこの名前よりスプライート動画を設定できる

"animate-path" : スプライート動画を記録しているテクスチャのパス

"animate-stride" : 横と縦のUV値階差、例えばDX授業にいただいたrunman.pngが二行五列で分けられている、ですので、strideは1 / 2 = 0.2と1 / 2 = 0.5である

"max-count" : このスプライート動画はいくつの部分に分けられているを記入、runman.pngは計十コマなので、10で記入

"repeat-flag" : もし真で記入すると、この動画はずっとループされる、偽だとしたら一回終わったら止まる

"frame-time" : スプライート動画を次のコマを表示するの時間差、1.0でしたら１秒間１フレーム、0.1でしたら１秒間１０フレーム

"init-animate" : 最初はどのスプライート動画を表示するかを定義する

ATimerComponent

"type" : "timer"と記入してください

"timer" : 大かっこ内に付けたいタイマーの名前を登録、後でこの名前で特定のタイマーを取得できる

```
{
    "type": "timer",
    "update-order": 0,
    "timers": [
        "r",
        "g",
        "b"
    ]
},
```

ACollisionComponent

※　ATransformComponentが必要

"type" : "collision"と記入してください

"collision-type" : 円(circle)と四角形(rectangle)のどちらで
円と円、四角形と四角形、円と四角形の間で
当たり判定が行える

"collision-size" : 円だとしたら半径を記入、四角形だとしたら
幅さ(横)と高さ(縦)を記入

"show-flag" : 当たり判定のサイズと結果を可視化するかどうか

```
{
    "type": "collision",
    "update-order": 0,
    "collision-type": "circle",
    "collision-size": [
        100.0,
        100.0
    ],
    "show-flag": true
},
```

UBtnMapComponent

"type" : "btnmap"と記入してください

"left" : 左に他のボタンがあればそのボタンのオブジェクト名前で
記入(そのオブジェクトもUBtnMapComponentを追加必要)

"right" : 右にbalabala

"up / down" : 上にbalabala、下にbalabala

"default-select" : このボタンは最初から選択されたら真で記入
どちらのボタンオブジェクトのbtnmapが必ず
真にする、そして一つだけで真にする（複数で不可）

```
{
    "type": "btnmap",
    "update-order": 0,
    "left": "test-ui",
    "default-select": false
},
```

UTextComponent

```
{
    "type": "text",
    "update-order": 0,
    "moji-path": "rom:/Assets/Textures/moji.png",
    "init-text": "it's a great day, \nおはようこ゛さ゛います! \nきょうはさむいて゛す \nフ゛ルフ゛ルヒ°カチュウ nice \n 0321054.34654",
    "init-size": [
        40.0,
        40.0
    ],
    "init-position": [
        -940.0,
        -320.0,
        0.0
    ],
    "init-color": [
        0.0,
        0.0,
        0.0,
        1.0
    ]
}
```

"type" : "text"と記入してください

"moji-path" : 特定のフォントで作ったもじテクスチャのパスを記入、デフォルトはイメージの通りで

"init-text" : かな、英字、数値が記入できる濁点は単独の゛で、半濁点は単独の°（温度とかの度で）、もし仮名がある場合、そしてこの場合だけ、ファイルのエンディングをGB 2312に変更してください（怪しいすぎのは重々承知の上ですが、そこはどうしてもなおれないです。。。一つの解決策はここに全ての英字でUTF8で記入、この後コードの中にこのコンポーネントを取得して、ChangeTextStringを正しいテキストを設定する）\nという記号はEnterのように新しい行でかくこと

"init-position" : このテキストはどこに表示されるを記入

"init-color" : このテキストはどんな色で表示する

AInputComponent / UInputComponent

"type" : "input"と記入してください

"func-name" : 自分で書いた関数の名前を記入、
関数の戻り値と引数の型を注意

※　自分はこういう仕組みをスクリプト関数で呼んでいます。
後でスクリプト関数の部分で説明を行います。
ここはとりあえず正しい関数名記入を注意してください

```
{
    "type": "input",
    "update-order": 0,
    "func-name": "SceneSwitch"
}
```

AInteractionComponent/UInteractionComponent

"type" : "interaction"と記入してください

"init-func-name" : 自分で書いた初期化用関数の名前を
記入、関数の戻り値と引数の型
を注意

"update-func-name" : 自分で書いた更新用関数の
名前を記入、関数の戻り値と
引数の型を注意

"destory-func-name" : 自分で書いた終了用関数の
名前を記入、関数の戻り値と
引数の型を注意

※　自分はこういう仕組みをスクリプト関数で呼んでいます。
後でスクリプト関数の部分で説明を行います。
ここはとりあえず正しい関数名記入を注意してください

```
{
    "type": "interaction",
    "update-order": 0,
    "init-func-name": "TestInit",
    "update-func-name": "TestUpdate",
    "destory-func-name": "TestDestory"
}
```

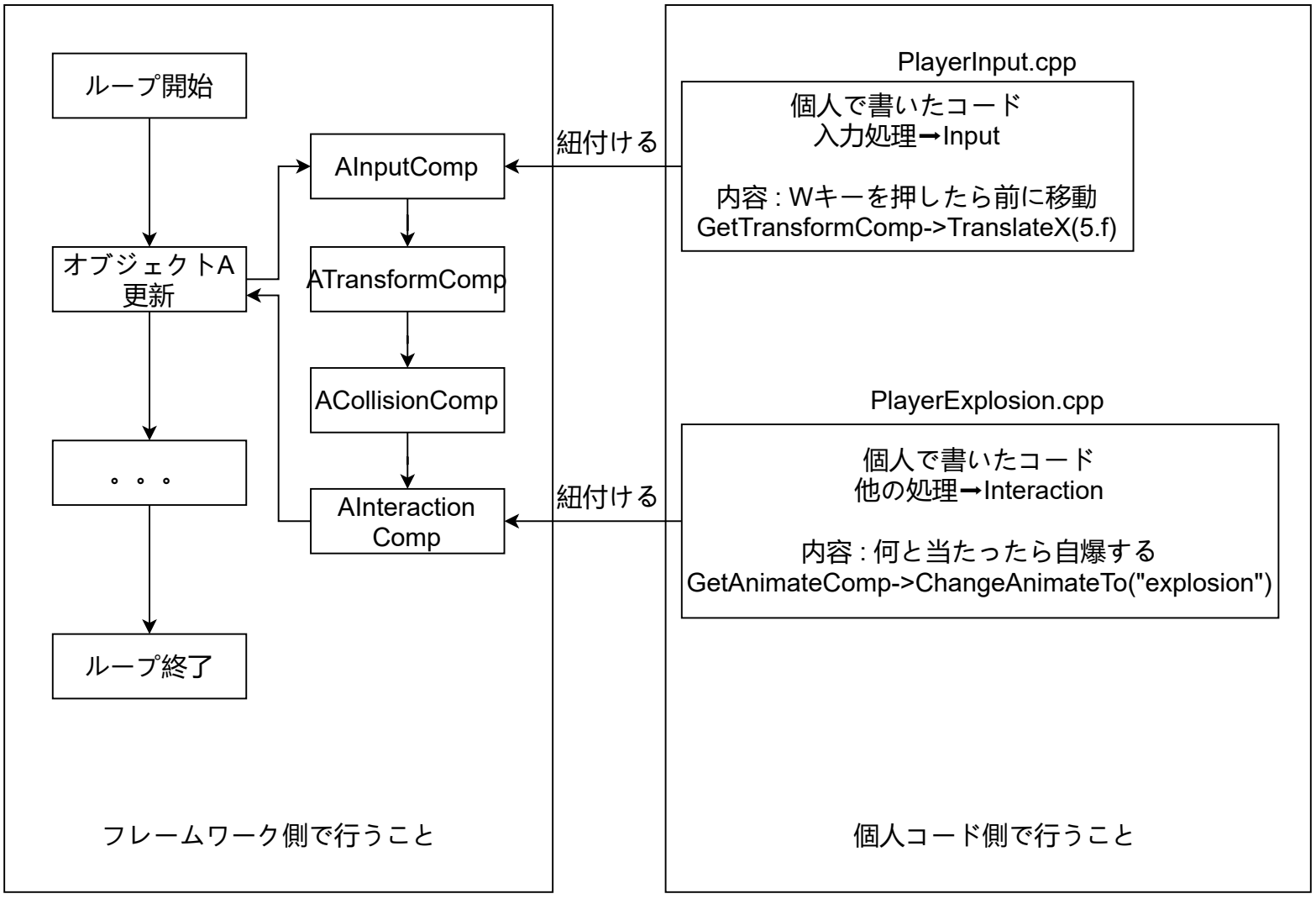
スクリプト関数に関して
(一番重要な部分です！！)

今まで紹介されたコンポーネントだけでは、シーンは動けなくなるでしょう、なんの操作するためのコンポーネント一つもないみたいですね。

そこで、AInputComponent / UInputComponent / AInteractionComponent / UInteractionComponent の出番だと思います。

この四つのコンポーネントは、フレームワーク自体のコードだけではなくて、個人で書いたコードも必要な時に自動的に呼び出すため作ったもの。

ちょっとUnityのscriptと少しだけ似ているところがあるんで、自分はこういう仕組みをスクリプト関数と読んでいます。（調子乗ってる。。。）



もしこの部分がないと位置はどうしても移動できない、例えかぶっても自爆できない。

フレームワークはあくまで流れ図に過ぎない、具体的にどう動作のは、必要な時、開発者の皆さんが書いたコードを呼び出して、皆さんの思い通りにゲームを動かす。

もし分かりにくいだとしたら、UnityのC# scriptの役割と使い方を少し復習して、こっちの方に代入すれば少々理解しやすいになるかもしれません。

スクリプト関数のルール

※　戻り値の型と引数の型要注意！必ず各コンポーネントのサンプルを沿って作ってお願いします！

AInputComponent : void 関数名自由(AInputComponent* _aic, float _deltatime);
　　例えば　void PlaneInput(AInputComponent* _aic, float _deltatime) { balabala; }

AInteractionComponent : 初期化 void 関数名自由(AInteractionComponent* _aitc);
　　例えば　void PlayerInit(AInteractionComponent* _aitc) { balabala; }

AInteractionComponent : 更新 void 関数名自由(AinteractionComponent* _aitc, float _deltatime);
　　例えば　void PlayerUpdate(AInteractionComponent* _aitc, float _deltatime) { balabala; }

AInteractionComponent : 終了 void 関数名自由(AIneractionComponent* _aitc);
　　例えば　void PlayerDestory(AInteractionComponent* _aitc) { balabala; }

UInputComponent : void 関数名自由(UInputComponent* _uic, float _deltatime);
　　例えば　void ButtonInput(UInputComponent* _uic, float _deltatime) { balabala; }

UInteractionComponent : 初期化 void 関数名自由(UInteractionComponent* _uitc);
　　例えば　void HpBarInit(UInteractionComponent* _uitc) { balabala; }

UInteractionComponent : 更新 void 関数名自由(UInteractionComponent* _uitc, float _deltatime);
　　例えば　void HpBarUpdate(UInteractionComponent* _aitc, float _deltatime) { balabala; }

UInteractionComponent : 終了 void 関数名自由(UIneractionComponent* _aitc);
　　例えば　void HpBarDestory(UInteractionComponent* _uitc) { balabala; }

※　フレームワークに登録する関数も用意するひつようです、引数の型も要注意

登録関数 : void 関数名自由(ObjectFactory* _factory);
　　例えば :

```
void PlayerFuncsRegister(ObjectFactory* _factory)
{
   auto aInputPoolPtr = _factory->GetActorInputPool(); // Actor入力関数を保存するところ
   auto aInitPoolPtr = _factory->GetActorInterInitPool(); // Actor初期化関数を保存するところ
   auto aUpdatePoolPtr = _factory->GetActorInterUpdatePool(); // Actor更新関数を保存するところ
   auto aDestoryPoolPtr = _factory->GetActorInterDestoryPool(); // Actor終了関数を保存するところ
   auto uInputPoolPtr = _factory->GetUiInputPool(); // Ui入力関数を保存するところ
   auto uInitPoolPtr = _factory->GetUiInterInitPool(); // Ui初期化関数を保存するところ
   auto uUpdatePoolPtr = _factory->GetUiInterUpdatePool(); // Ui更新関数を保存するところ
   auto uDestoryPoolPtr = _factory->GetUiInterDestoryPool(); // Ui終了関数を保存するところ

   aInitPoolPtr->insert(std::make_pair(FUNC_NAME(TestInit), TestInit));
   aUpdatePoolPtr->insert(std::make_pair(FUNC_NAME(TestUpdate), TestUpdate));
   aDestoryPoolPtr->insert(std::make_pair(FUNC_NAME(TestDestory), TestDestory));
   uInitPoolPtr->insert(std::make_pair(FUNC_NAME(TestUiInit), TestUiInit));
   uUpdatePoolPtr->insert(std::make_pair(FUNC_NAME(TestUiUpdate), TestUiUpdate));
   uDestoryPoolPtr->insert(std::make_pair(FUNC_NAME(TestUiDestory), TestUiDestory));
   aInputPoolPtr->insert(std::make_pair(FUNC_NAME(TempMove), TempMove));
   aInputPoolPtr->insert(std::make_pair(FUNC_NAME(SceneSwitch), SceneSwitch));
   uInputPoolPtr->insert(std::make_pair(FUNC_NAME(TempUiInput), TempUiInput));
}
```

ここは例として全ての種類のPoolを呼び出したけど、実際にコード書く時、必要な部分だけ呼び出せばオッケー。例えば、このcppファイル内にはActor入力処理関数しかないなら、auto aInputPoolPtr = _factor->GetActorInputPool();を書けば十分です。

下の半分は個人で書いた関数をフレームワークに登録するための文、こういうテンプレートで書いてください：

(関数保存するところ)->insert(std::make_pair(FUNC_NAME(個人で書いた関数名), 個人で書いた関数名));

登録関数を作ったら、Source Filesのフィルタにあ るFuncsRegister.cppに登録関数所在のヘッダーファイルをインクルードして、RegisterAllFuncsの関数内に書いて、_factoryを引数として渡す。

※　登録するときは十分な用心が必要。例えば、Actor入力処理関数は必ずGetActorInputPool()の中にinsertする、Ui更新処理関数は必ずGetUiInterUpdatePool()の中にinsertする。

登録完了後、jsonファイルにこれらの関数を実行させたいオブジェクトに関数の名前を記入すればオッケー。これで実行すると、皆さんが書いた関数もちゃんと実行されるはずです。

```
{
    "type": "interaction",
    "update-order": 0,
    "init-func-name": "TestInit",
    "update-func-name": "TestUpdate",
    "destory-func-name": "TestDestory"
}
```

```
{
    "actor-name": "scene-switch",
    "update-order": 0,
    "parent": null,
    "components": [
        {
            "type": "input",
            "update-order": 0,
            "func-name": "SceneSwitch"
        }
    ]
},
```

ほかのもの

音声方法

シーンJSONファイル内に登録した音声は"sound.h"をインクルードして、PlayBgm(音声の名前)とPlaySe(音声の名前)で再生できる。

画面の大きさ変更

00_BasicFunc/DxProcess.cppの21行目、最後の引数をtrueになったらフルスクリーンになる、falseだったら二個前の引数より決定される、1280,720とか、1920,1080とか、2560,1440とか

ログ出力

P_LOGというマクロを利用すればできる、使い方はスイッチのNN_LOGとほぼ同じ、でも文字列の前にログレベルを追加するが必要。例えば、LOG_ERROEとかLOG_DEBUGとか

入力デバイス

関数名はGMのcontroller.hとほぼ同じ、01_MiddleFunc/ControllerHelper.h/.cppを一旦確認してお願いいたします。