プロジェクト規約

目次

1.設計思想

基本事項

- 言語のバージョンは C++17 を使います。
- なるべく STL で記述できる部分は STL を使いましょう
- グローバル領域での using namespace を禁止します。
- std::any にポインタを格納する場合は、生ポインタを使用してください。
- 特に理由がない場合メモリの動的確保は std::unique_ptr を使いましょう
- 定数には constexpr を使いましょう。
- 関数・メソッドの戻り値や引数に適切に const をつけましょう。
- 場合によって属性(p5 にて解説)を付けたり、final や override などの修飾子を必ずつけましょう。
- ソースファイルは基本的に.hpp とし、ヘッダに記述することとします。
- 相互インクルードが起きる場合や、実装部分が多い場合にのみ cpp と分けます。
- アプリケーション以外の機能(Vec 等)の変更はマネージャーに報告してください。
- 外部ライブラリとして DxLib を用います。

Singleton パターンについて

- アプリケーション部分でシングルトンパターンを用いる場合、初期化順を制御したい場合のため必ず動的に生成しましょう。また、コードの見通しが悪くなり、バグの原因になりやすいため、シングルトンの内部処理において他のシングルトンクラスの使用を禁止にします。
- シングルトンの実装部分はインナークラスとして記述しましょう。

```
class Singleton final{
        class Implement{
                //本体
        };
public:
static Implement& get(){
                //インスタンス取得
        }
};
```

アーキテクチャー

- 今回のプロジェクトでは EntityComponentSystem(以下 ECS と略称)を用いて開発を行います。
- ECS は複数の独立したコンポーネントシステム(振る舞い)と、コンポーネントデータ(変数と考えてよいです)によってエンティティ(ゲームオブジェクト)を定義し、アプリケーションを作っていく開発方式です。
- 今回は実装の手間や学習コストを考慮し、オブジェクト指向ベースで実装を行います。
- ゲーム開発の手法でよく採用されているようです。

メリット

- この方式を使うことで、エンティティを柔軟に定義でき、動的に振る舞いを変更することが可能になります。
- 汎用性を考えて振る舞いを作れば、様々な振る舞いを組み合わせたり、他のエンティティにも同じ振る舞いを組み込むことができます。

デメリット

- 基本的に振る舞いを作っていく作業になるのでクラスや構造体が大量に増えてしまいます。
- 処理負荷がやや高い。
- 柔軟な記述が可能な反面、具体的な記述をしなければならない部分が出てきやすくなります。

2.命名規則

使ってはいけない記述

アンダースコア＋大文字で始まる名前	_HOGE _FUGA_
連続する 2 つのアンダースコアを含む名前	__hoge・my__fuga
グローバル領域においてアンダースコアで始まる名前	::_hoge

命名規則

ユーザー定義型名	パスカルケース	PascalCase
プライベート変数	キャメルケースに_	camelCase_
パブリック変数 メソッド 引数	キャメルケース	camelCase
グローバル変数 自動変数(スコープで死ぬ)	スネークケース	camel_case
グローバル関数 名前空間に属する関数	パスカルケース	PascalCase
定数 列挙型 マクロ	すべて大文字スネークケース	SNAKE_CASE

テンプレート

テンプレート引数は基本的に T を用います。

templete <class T>

明示的な場合はパスカルケースでしっかりつけるようにしましょう。

templete <class NameType>

可変引数は Args とします。

templete <class... TArgs>

文字列

日本語禁止　“./resource/image/player.png”このようにします。

3. 警告、エラー、例外処理

警告について

警告レベルは 4 とします。リリース時には必ず残さないようにコードで解決しましょう。

属性

- 値を返す関数・メソッドは原則として **nodiscard** 属性を付けましょう。

[[nodiscard]] int getVal() {return 1;}

これにより戻り値を無視した場合、警告が発生します。

- switch 文で break しない場合は **fallthrough** 属性を付けましょう。

switch (state){

case State::STATE1:

[[fallthrough]];

case State:: STATE2 break;

}

これにより、意図しない break かどうかが明示的になります。(環境によっては break がない場合警告が出ます)

- 場合によって使わない引数や変数には **maybe_unused** 属性を付けましょう。

void update([[maybe_unused]] int val)

なんらかのデザインパターンを用いたとき、まれにこのような場面が発生します。

原則として意味もなく付けてはいけません。

- 非推奨な機能、クラス、変数には[[**deprecated**]]属性をつけましょう。

[[deprecated]]void oldFunc() {}

または

[[deprecated("表示したい警告文")]]

void oldFunc() {}

エラー処理

- プログラムに不都合が生じるエラーに関しては **assert()** を用いてどこでエラーが発生しているか明示的にしましょう。

・例外やエラーが発生した場合にもゲームを続行させたい場合には、**try・catch** 文で行いましょう。

4. コメントのつけ方

コメントは Doxygen 形式で付けます。クラス名やメソッド名等は省略します。

ファイルへのコメント

※ファイルの先頭に記述

```
/**
 * @file  ファイル名.拡張子
 * @brief  簡単な説明
 * @author  書いた人
 * @date  日付（開始日）
*    @par  更新履歴
- 日付  書いた人
-# 変更点
 */
```

クラスへのコメント

```
/**
* @brief  簡単なクラスの説明
* @details  詳細なクラスの説明(あれば)
*/
```

関数へのコメント

```
/**
* @brief  簡単な説明（～する関数）
* @param a(引数名）引数の説明
* @param b(引数名）引数の説明
* @return  型名  戻り値の説明(void なら不要)
* @details  詳細な説明(あれば)
*/
```

マクロへのコメント

/**
* @brief 簡単な説明
* @details 詳細な説明(あれば)
*/

名前空間へのコメント

/**
* @brief 簡単な説明
* @details 詳細な説明(あれば)
*/

変数へのコメント

//! ○○で使う変数

列挙型へのコメント

/**
* @brief 簡単な説明
* @details 詳細な説明(あれば)
*/

プライベートなメンバやアルゴリズムに対してのコメント

原則として任意ですが簡単にでもいいのでコメントを付けてもらえると後から読みやすいのであると助かります。
ついていないものに関してはレビューの責任は持てません。
また、後から他人がコードを引き継ぐ場合もあるので、特に複雑なものに関してはお願いします。

その他

@note（覚え書き）、@bug（バグ）、@warning（警告）なども必要に応じて書くとよいでしょう。結局のところレビュー時に把握できれば大丈夫です。
また、簡単な関数やメソッド等は変数と同じコメントのつけ方で良いでしょう。

5.バージョン管理

- 管理は原則的にマネージャーが行います。ブランチのマージやコードレビュー等です。
- リモートは Github にします。
- 原則として 1 作業 1 ブランチとします。
- ファイルの追加は実際のフォルダ構成を守ってフィルターに追加してください。
- フィルターのコンフリクトは、変更点をプッシュせず、マージ時に追加しなおすことで解決します。
- ブランチをマスターから作る場合はマネージャーに事後でもいいので報告してください。
- リポジトリはチーム用のものを使います。
- 作業が終わったらプルリクエストかマネージャーに報告してください。
- Github 上の Issue にバグや ToDo 等をまとめます。作業振り分けなどはこれを使います。また、質問やバグなどのメモとして各自で自由に使ってください。終了したものに関しては close してください。

6.ファイル・フォルダの構成

プロジェクトファイルがある階層を root とします。

構成図

※以下の図はあくまでも参考用です

```
root/
   ├ main.cpp
   ├ src/
   |     ├ ECS/
   |     |
   |     ├ Scene/
   |     |       ├Game.h
   |     |       └Game.cpp
   |     ├ Components/
   |     |           └BasicComponents.hpp
   |     └ Utility/
   |           └Vec.hpp
   ├ resource/
   |           ├ image/
   |           ├ sound/
   |           └ score/
   |
   └ DxLib/
```

7.困ったときは...

~~ggrks~~

アルゴリズムで困ったとき

- 先生やチームメンバーに聞きましょう。

実装方法で困ったとき

- プロジェクトマネージャーや、リソース作成者に聞きましょう。

命名に困ったとき

- 周りの人に見てもらいましょう。

Git で困ったとき

- プロジェクトマネージャーと相談しましょう

作業ができないとき

- プロジェクトマネージャーに報告し、別の人に頼むといった対応をしましょう。
  完全に作業が止まってしまうのはなるべく避けましょう。

理由もなく返事、連絡がないとき

- 鬼電します(^q^)