カメラ処理の効率化手法

– ザインパターンの応用事例 –

2014 年 3 月 13 日　初稿

板垣 衛

■ 改訂履歴

稿	改訂日	改訂者	改訂内容
初稿	2014 年 3 月 13 日	板垣 衛	（初稿）

■ 目次

■ 概略

デザインパターンの応用事例の一つとして、カメラ処理の開発効率を向上させるプログラミング手法を説明する。

■ 目的

本書は、何かと無駄が生じがちなカメラ処理の構造を見直し、シンプルな処理に切り分けることを目的とする。

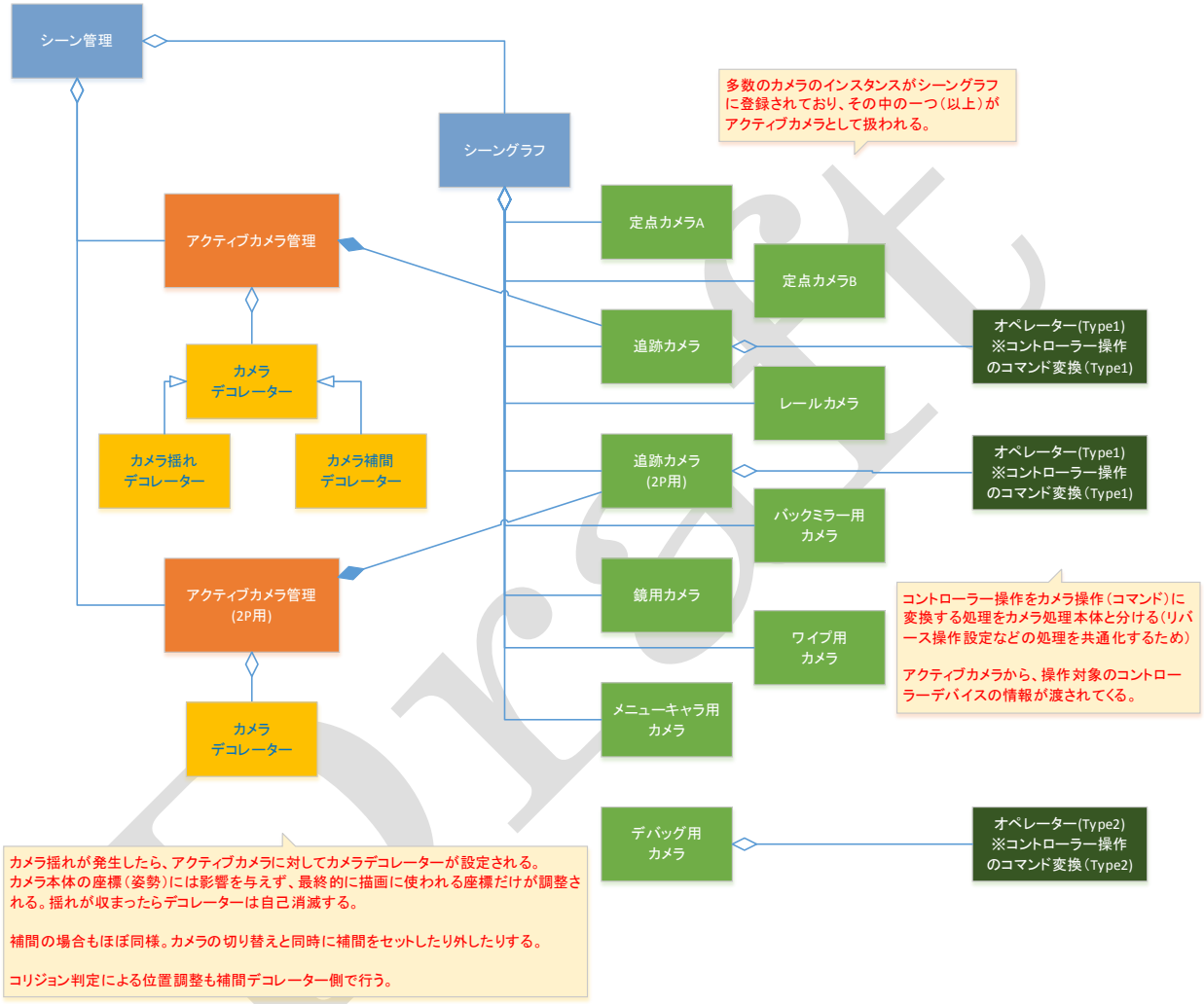
■ 対処すべき問題点

一つのゲームの中では、多数のカメラ処理を切り替えて使用する。プレイヤー追跡型のカメラやレールカメラ、定点カメラ、イベントシーン用カメラなど様々である。問題となるのは、このような処理の対応として、ごり押しのコピペや巨大なカメラクラス、無造作な継承などが行われることである。具体的に説明する。

- 【問題】一つの巨大なカメラ処理（クラス）で構成し、内部で状況に応じて挙動を変えている。
  - 無駄なコピペコードの発生を抑えて共通化することはできるが、コードが複雑になりがち。新しい処理を加える際に、他のカメラに影響を及ぼし易い。
- 【問題】カメラ揺れやコントローラー操作の処理を多数のカメラにコピペして使用している。
  - コピペが抜けていると、一部のカメラの時だけカメラが揺れないということもある。また、バグ修正や挙動の修正が必要になった時、全部を直さなければならず、チェック作業も含めて非効率。
- 【問題】カメラ揺れや補間などの基本処理を親クラスの処理として記述して継承する。
  - カメラ揺れの影響を受けたくないカメラや、補間をしたくないカメラの制御に難があったり、親クラスの処理が座標を書き換えてしまうために処理を作りにくかったりといった事がある。

■ 対策

コントローラー操作、カメラ揺れ、補間といった処理は、なるべくカメラ本体の処理から切り離して扱う。

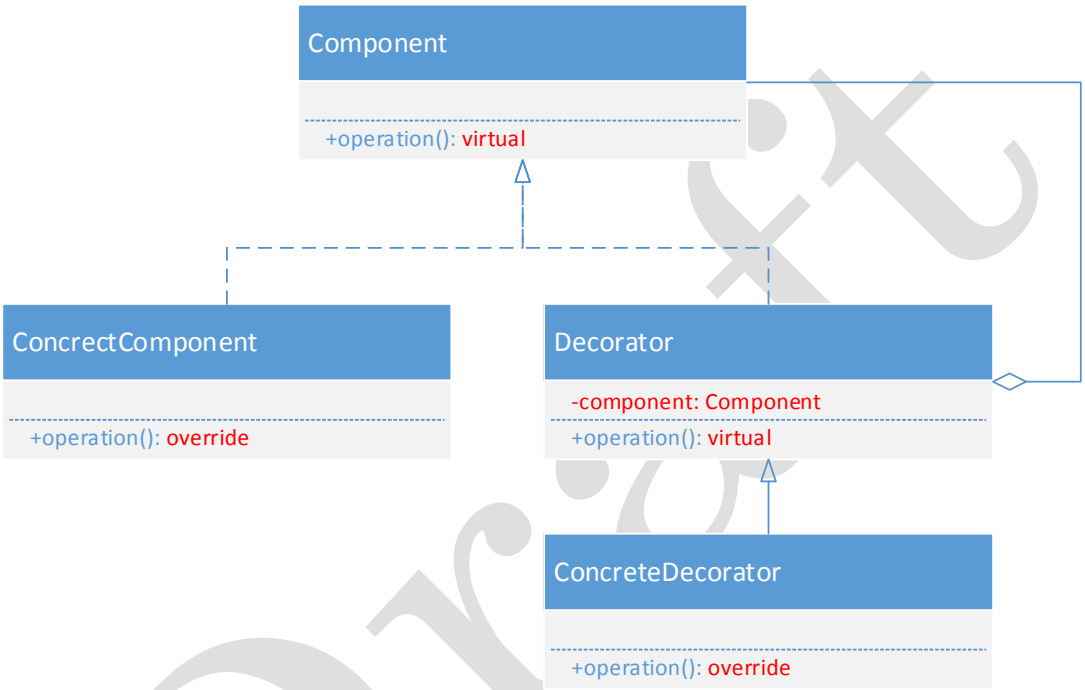


▼ カメラデコレーター（デコレーターパターン）

カメラ揺れ、補間といった、カメラの動作を装飾・補助する処理には、デザインパターンの「デコレーターパターン」を利用して、カメラ本体の処理と分離する。

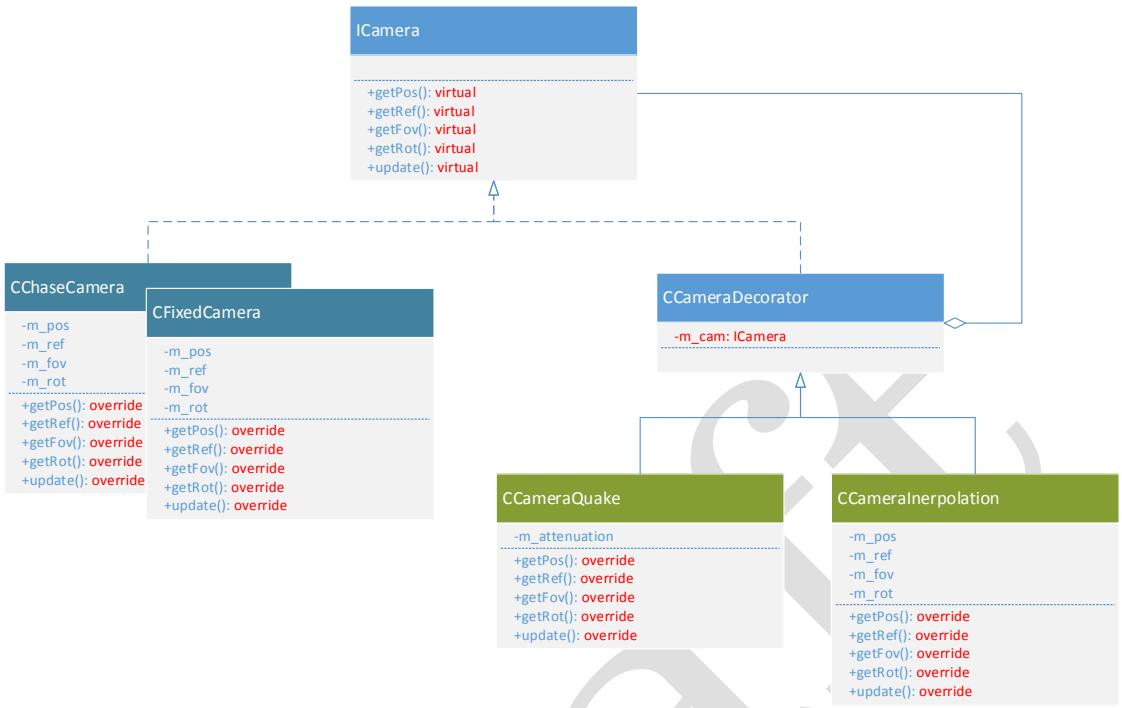
まず、一般的なデコレーターパターンを示す。

一般的なデコレーターパターンのクラス図：



これを元に、「カメラデコレーター」を設計する。

カメラデコレーターのクラス図：

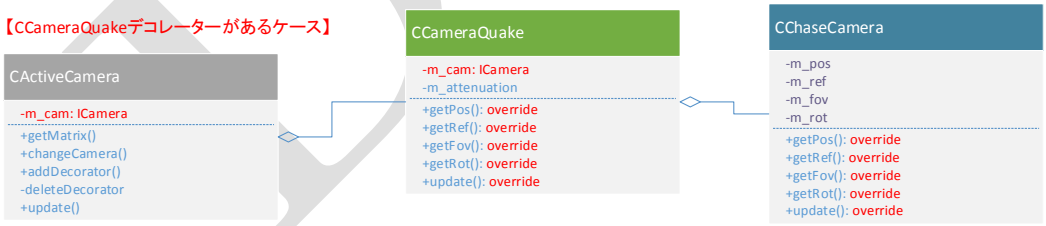


インスタンスのイメージ：

【カメラデコレーターがないケース】

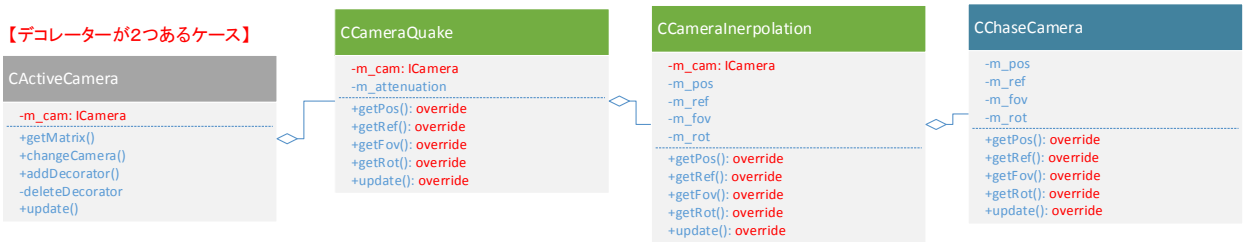
update() 実行時は、CChaseCamera の update() が実行される。

【CCameraQuakeデコレーターがあるケース】



update() 実行時は、CCameraQuake の update() が実行され、その中の処理の最初に CChaseCamera の update() が実行される。
また、CCameraQuake のほとんどの仮想メンバー関数は、透過的に CChaseCamera のメンバー関数を呼び出す（処理の委譲）。

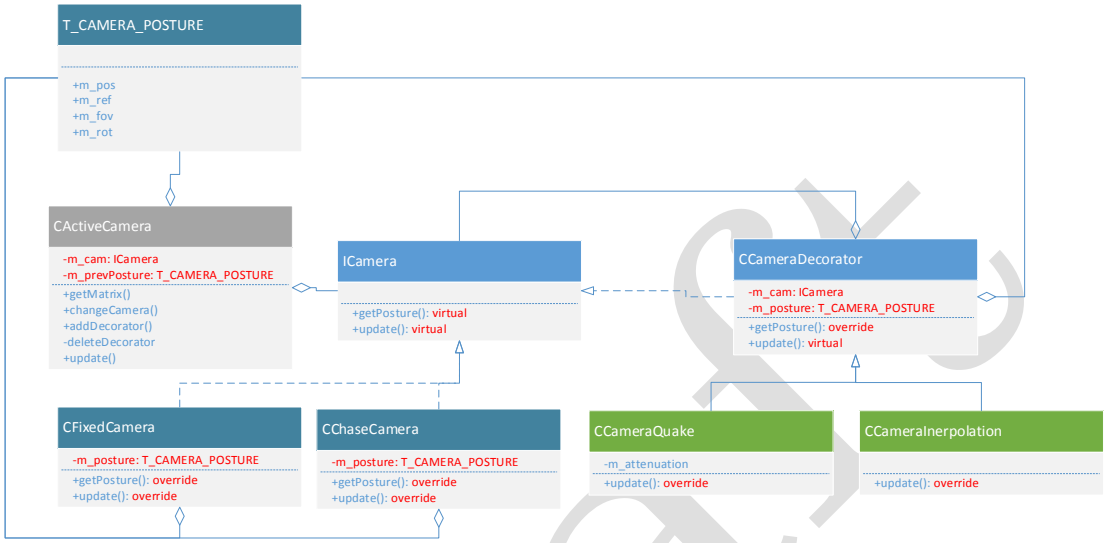
【デコレーターが2つあるケース】



このようにして、いくらでもデコレーターを連結できる。その上、カメラを扱う側から見れば、一つのカメラと変わらない。
カメラ本体のクラスは、補間や揺れの影響で変化する座標を気にせず処理できる上、連結の逆順に結果（座標）が加工されていく。

図には、カメラの姿勢の情報を各メソッドで扱うように描いているが、実際には一つの構造体にまとめて、少しでも仮想メソッドを減らすほうが良い。また、前のフレームの情報の記録としても扱うべき。

カメラの姿勢情報を加えたクラス図：



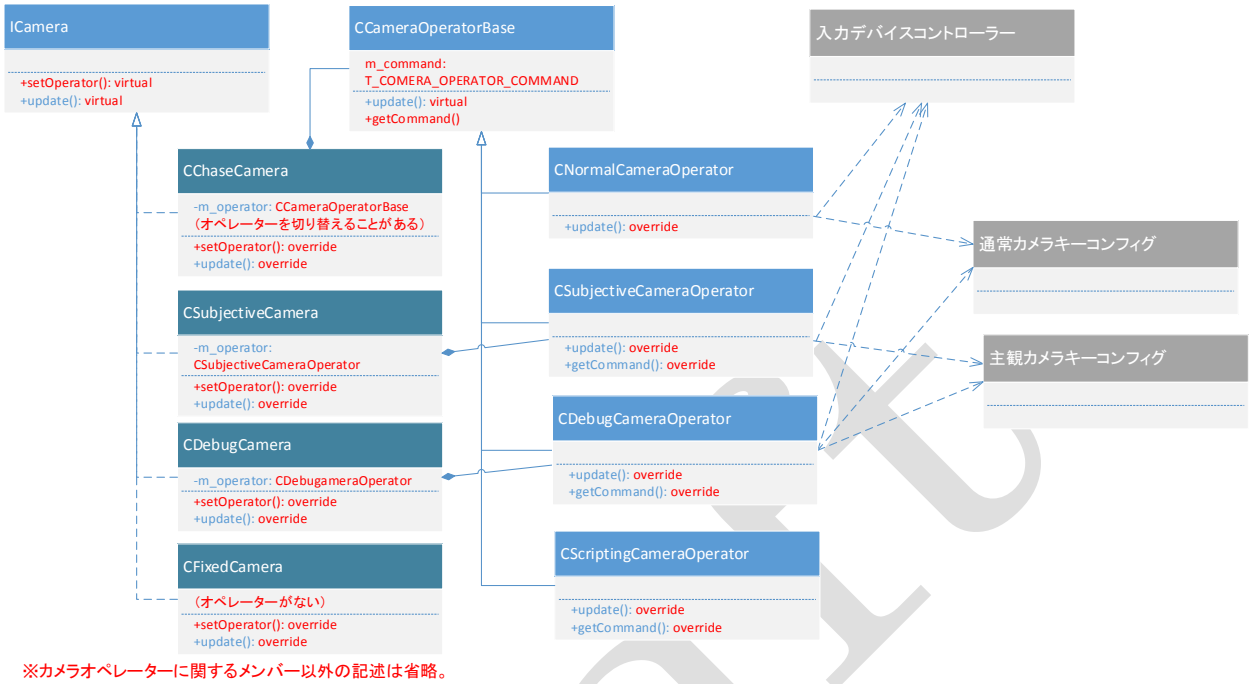
▼ カメラオペレーター

入力デバイスなどによる、カメラ操作は専用の別クラスに切り離して扱う。

カメラ本体が直接入力デバイスを扱うようなことはせず、「カメラオペレーター」に任せるようにする。

カメラオペレーターは、入力デバイスの情報を「カメラ操作コマンド」に変換して返す。これにより、「リバース操作設定」や「カメラ移動量設定」などの「キーコンフィグ」を、カメラ側が一切扱わなくて良い状態にする。

カメラオペレーターのクラス図：

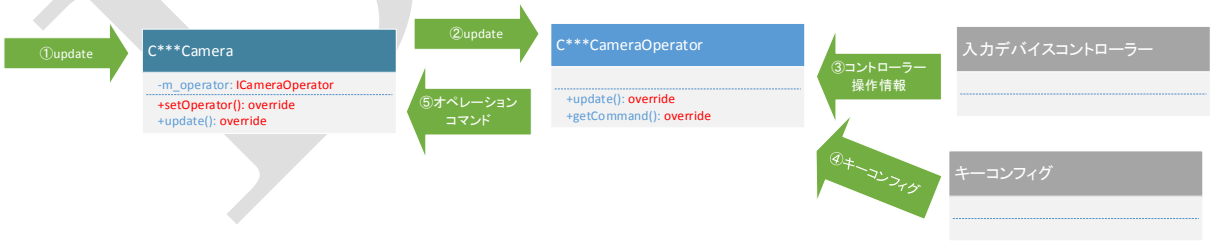


※カメラオペレーターに関するメンバー以外の記述は省略。

幾つものカメラを扱うような場合には、オペレーターを共通利用することで、処理の共通化を図る。

また、オペレーターのバリエーションを用意することにより、カメラクラス本体に手を入れずに、入力デバイス以外の方法でもカメラを操作させることができる。例えば、あらかじめ記録されたカメラ再生パターンを再生することなどが可能。

カメラオペレーターの使用イメージ：



【注意】カメラオペレーションコマンドを実際のカメラの動作に反映させる際は、「前のフレームの」（つまりユーザーが見ている状態の）カメラの姿勢に基づく必要がある点に注意。また、それはカメラクラス自体が保持する姿勢ではだめ。デコレーターの影響を加味した姿勢情報を扱う必要がある。更新処理（update）時にパラメータで受け渡すなどする。

カメラオペレーションコマンドの例：

- 視点移動方向 2D ベクトル（基本：-1.0, -1.0 ～ 1.0, 1.0）
- 注視点移動方向 2D ベクトル（基本：-1.0, -1.0 ～ 1.0, 1.0）
- 平行移動方向 2D ベクトル（基本：-1.0, -1.0 ～ 1.0, 1.0）
  - 2D ベクトルは、アナログスティックや十字ボタンを倒した方向を表す。値は正規化されているものとする。
  - 2D ベクトルの符号は、キーコンフィグのリバース設定が反映される。
- 視点移動動量（基本：～ 1.0）
- 注視点移動量（基本：～ 1.0）
- 平行移動量（基本：～ 1.0）
  - 移動量 = 1.0 を、カメラ側が規定する標準の移動量とする。
  - 移動量は、アナログスティックの倒した量が反映されるものとする。基本的には、倒してない状態を 0.0、最大に倒した状態を 1.0 とした時の割合となる。なお、アナログスティックの遊びの範囲の補正は入力デバイスコントローラー側で行われる。例えば、0.0～0.2 を 0.0 に補正し、0.2～1.0 を 0.4～1.0 に補正するといった調整である。
  - デジタルボタン（十字ボタン）が移動に使われた場合、基本的には移動量を 1.0 とする。いずれにしてもオペレーター側の処理次第。
  - 移動量には、キーコンフィグのカメラ移動速度を反映して乗算した値が返される。つまり、標準よりも速い移動が指定された場合、1.0 を超える値となり得る。
- 視点前後移動量（基本：-1.0 ～ +1.0）
- 注視点前後移動量（基本：-1.0 ～ +1.0）
- 画角変化量（基本：-1.0 ～ +1.0）
- ロール回転量（基本：-1.0 ～ +1.0）

※どのようなコントローラー操作がどのコマンドに反映されるかはオペレーター側の処理次第。キーコンフィグをどのように扱うかも同様。

※どのコマンドをどのように扱うかはカメラ側の処理次第。対応しないコマンドは無視される。例えば、デバッグ用カメラには全てのコマンドが必要になる。

■■以上■■

■ 索引

索引項目が見つかりません。

カメラ処理の効率化手法

以　上