回転処理

操作キャラクターの向きを移動方法に向けるための
回転を行う処理を追加していきます。

3DWorld同様に、徐々に移動方向へ向きを変える遅延回転を行います。
これは、エネミーにも必要な機能になりますので、
CharactorBaseに実装を行います。

遅延回転処理をDelayRotate関数とし、
基底クラスCharactorBaseのUpdate内で呼び出すことも可能ですが、
今後を見据えて、更新処理(Update関数)を再設計していきます。

△あまり良くない設計

```
void CharactorBase::Update(void)
{
    // 移動方向に応じた遅延回転
    DelayRotate();                          ← 共通部分だから、
}                                               CharactorBaseに記述しよう！

void Player::Update(void)
{
    // 移動操作
    ProcessMove();

    ActorBase::Update();                    ← 回転したあとに
                                                移動やモデル更新したいから
    // 移動処理                                 ココに挿入！
    transform_.pos = VAdd(transform_.pos, movePow_);

    // 3Dの基礎情報更新
    transform_.Update();

    // アニメーション更新
    animationController_->Update();
}
```

問題点①

基底クラスの処理を呼び出す、ActorBase::Update() が
派生クラス側で自由に記述できてしまうため、
ActorBase::Update() 自体が、
何かの前処理なのか、後処理なのか、処理の位置付けが不明確となる。

今回の例でいくと、transform_.Update(); の後に呼び出すと、
モデル更新が1フレーム遅れとなる。

問題点②

3Dの基礎情報更新、アニメーション更新は、
CharactorBaseの共通処理にすべき。

<u>○派生クラス側でやることが明確な設計</u>

基底クラス側から、派生クラスに「これをやりなさい！」と指示を出し、
派生クラス側にあまり自由を与えず、指示された内容しか記述できないようにします。

これにより、派生クラス側で実装すべき内容や場所が明確になり、
基底クラス側で設計した処理順が守られるようになります。

このように、処理の流れ(テンプレート)を基底クラスで決めて、
具体的な中身は派生クラスに書かせることを
Template Method(テンプレートメソッド)パターンといいます。

実装例

CharactorBase.h

```
class CharactorBase : public ActorBase
{

public:

    ～ 省略 ～

    // 更新
    virtual void Update(void) override;

    // 解放
    virtual void Release(void) override;

protected:

    ～ 省略 ～

    // 更新系
    virtual void UpdateProcess(void) = 0;
    virtual void UpdateProcessPost(void) = 0;

    // 移動方向に応じた遅延回転
    void DelayRotate(void);

};
```

CharactorBase.cpp

```
void CharactorBase::Update(void)
{

    // 各キャラクターごとの更新処理
    UpdateProcess();

    // 移動方向に応じた遅延回転
```

```
    DelayRotate();

    // モデル制御更新
    transform_.Update();

    // アニメーション再生
    animationController_->Update();

    // 各キャラクターごとの更新後処理
    UpdateProcessPost();

}

void CharactorBase::DelayRotate(void)
{

    // 移動方向から回転に変換する
    Quaternion goalRot = Quaternion::LookRotation(moveDir_);

    // 回転の補間
    transform_.quaRot =
        Quaternion::Slerp(transform_.quaRot, goalRot, 0.2f);

}
```

Player.h

```
class Player : public CharactorBase
{

public:

    ～ 省略 ～

    // 更新
    void Update(void) override;              → 削除

protected:

    ～ 省略 ～
```

```
    // 更新系
    virtual void UpdateProcess(void) override;
    virtual void UpdateProcessPost(void) override;
```

Player.cpp

```
void Player::Update(void)
{

    ～ 省略 ～                                  → 関数ごと削除

}

void Player::UpdateProcess(void)
{

    // 移動操作
    ProcessMove();

    // 移動処理
    transform_.pos = VAdd(transform_.pos, movePow_);

}

void Player::UpdateProcessPost(void)
{
}
```

解説

○ 基底クラス側の処理フロー

① // 各キャラクターごとの更新処理
UpdateProcess(); 派生クラスで実装！

⇒ 移動やジャンプ、重力、衝突など、
基本的な処理はこのフェーズで行う。
次の遅延回転に繋がる移動方向もここで決める。

② // 移動方向に応じた遅延回転
DelayRotate();

⇒ 前回フェーズで決められた移動方向に合わせて、
ゆっくり回転を行う。

③ // モデル制御更新
transform_.Update();

⇒ DxLib経由で、大きさ、回転、位置の情報を
モデルに反映させる。

④ // アニメーション再生
animationController_->Update();

⇒ 最新のモデル情報を元にアニメーションを再生させる。

⑤ // 各キャラクターごとの更新後処理
UpdateProcessPost();　　派生クラスで実装！

⇒ もし、派生クラス側で、メイン処理フロー後に
何かやりたいことがあれば、記述する

○ 遅延回転(DelayRotate)

移動方向(単位ベクトル)から、クォータニオンに変換。
ゴールとなる回転に向けて、徐々に今の回転を近づけるよう、
球面補間(Slerp)を使用しています。

0.2の値は、徐々に近づける比率で、数値は、0.0～1.0を指定します。
値が大きいほど、早くゴール回転に到達します。