コンピュータアニメーション特論

第12回　キャラクタアニメーション(5)

九州工業大学 情報工学研究院　尾下真樹

今日の内容

- 前回までの復習
- 動作接続･遷移
  - 動作接続･遷移の原理
  - サンプルプログラム
  - 動作接続のプログラミング
  - 動作遷移のプログラミング
  - 動作接続･遷移の拡張
  - 動作接続･遷移の応用
- 動作変形

キャラクタ・アニメーション

- CGにより表現された人体モデル（キャラクタ）のアニメーションを実現するための技術
- キャラクタ・アニメーションの用途
  - オフライン・アニメーション（映画など）
  - オンライン・アニメーション（ゲームなど）
    - どちらの用途でも使われる基本的な技術は同じ（データ量や詳細度が異なる）
    - 後者の用途では、インタラクティブな動作を実現するための工夫が必要になる
- 人体モデル・動作データの処理技術

全体の内容

- 人体モデル（骨格・姿勢・動作）の表現
- 人体モデル・動作データの作成方法
- サンプルプログラム
- 順運動学、人体形状変形モデル
- 姿勢補間、キーフレーム動作再生、動作補間
- 動作接続・遷移、動作変形
- 逆運動学、モーションキャプチャ
- 動作生成・制御

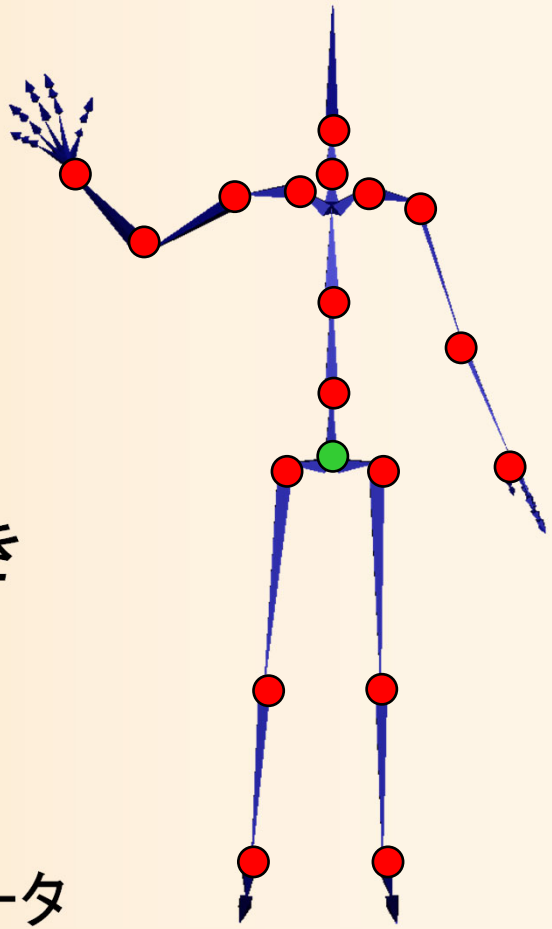
今日の内容

- 前回までの復習
- 動作接続･遷移
  - 動作接続･遷移の原理
  - サンプルプログラム
  - 動作接続のプログラミング
  - 動作遷移のプログラミング
  - 動作接続･遷移の拡張
  - 動作接続･遷移の応用
- 動作変形

前回までの復習

骨格・姿勢・動作の表現

- 人体の骨格の表現
  - 多関節体モデルによる表現
  - 複数の体節と関節
    - 関節は2つの体節の間を接続
- 姿勢の表現
  - 全関節の回転 ＋ 腰の位置・向き
- 動作の表現
  - 姿勢の時間変化
    - 一定間隔 or キーフレーム動作データ



骨格・姿勢・動作のデータ構造

- 骨格・姿勢・動作の
  構造体・クラスの定義
  （SimpleHuman.h/cpp）

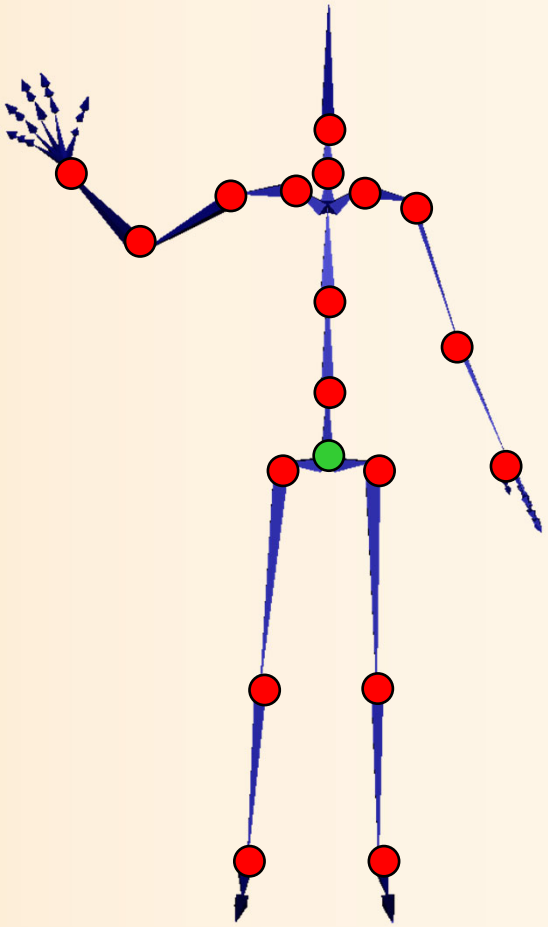
```
// 人体モデルの体節を表す構造体
struct  Segment

// 人体モデルの関節を表す構造体
struct  Joint

// 人体モデルの骨格を表すクラス
class  Skeleton

// 人体モデルの姿勢を表すクラス
class  Posture

// 人体モデルの動作を表すクラス
class  Motion
```

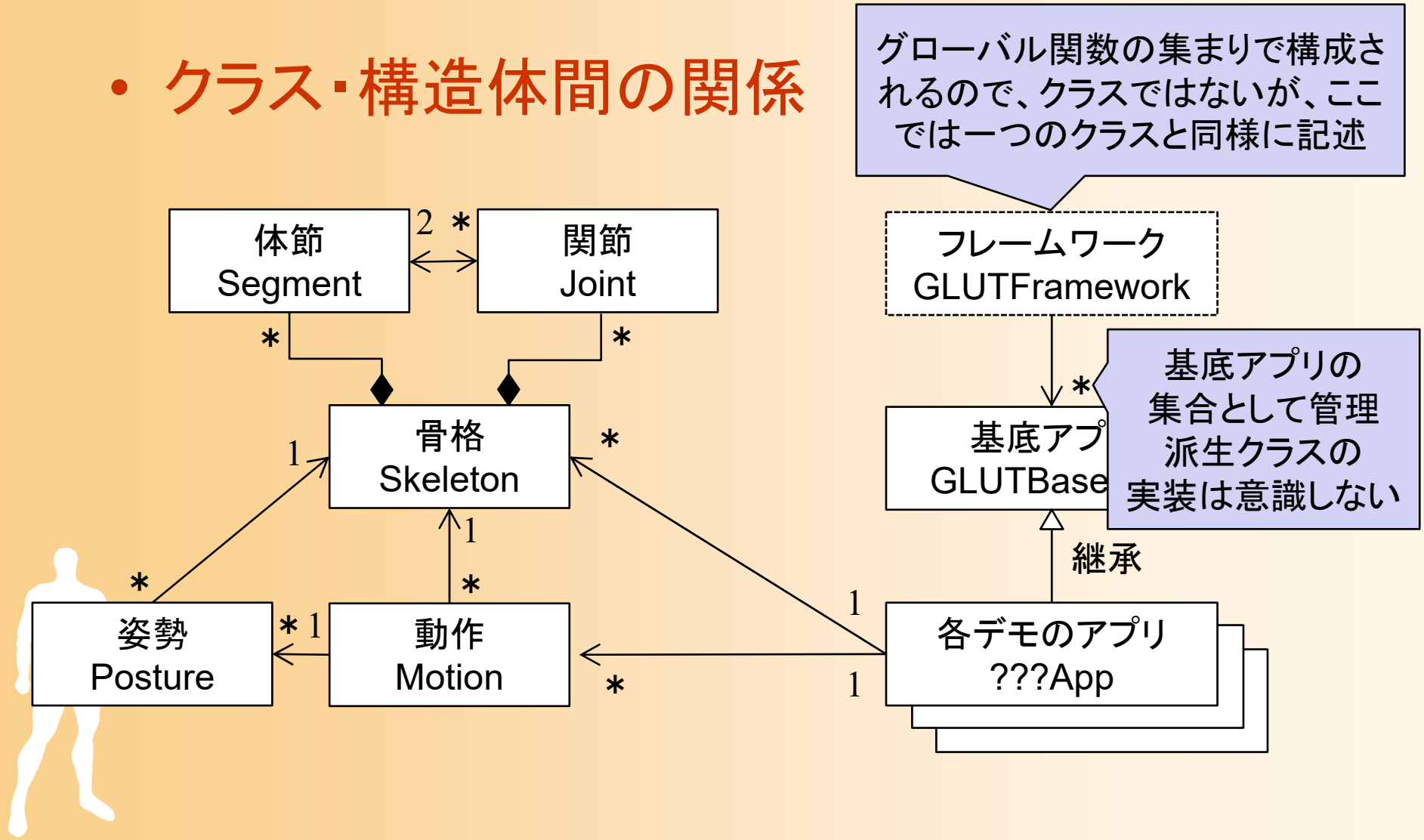


サンプルプログラム

- デモプログラムの一部のサンプルプログラム
  - 骨格・姿勢・動作のデータ構造定義（SimpleHumn.h/cpp）
  - BVH動作クラス（BVH.h/cpp）
  - アプリケーションの基底クラスとGLUTコールバック関数（SimpleHumanGLUT.h/cpp）
    - アプリケーションの基底クラス GLUTBaseAppの定義・実装
      - 各イベント処理のためのメソッドの定義を含む
      - 本クラスを派生させて各アプリケーションクラスを定義
    - 複数のアプリケーションの管理と、現在のアプリケーションのイベント処理を呼び出すGLUTコールバック関数
  - メイン処理（SimpleHumanMain.cpp）
  - 各アプリケーションの定義・実装（???App.h/.cpp）
    - 主要な処理を各自で実装（レポート課題）

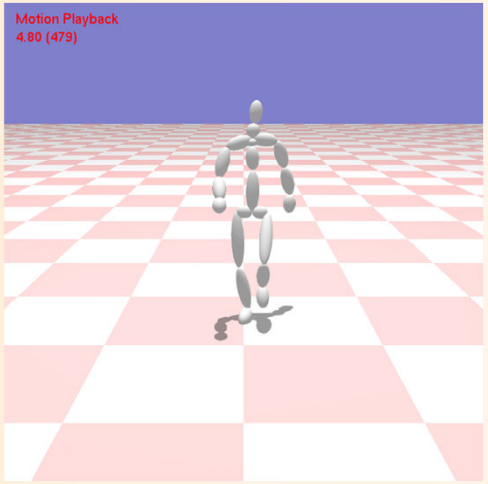
クラス図

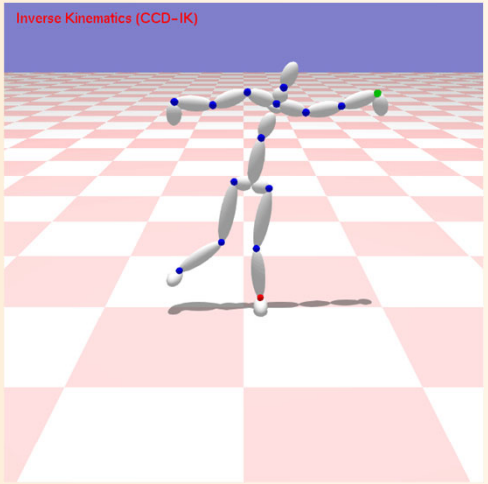
- クラス・構造体間の関係



サンプルプログラム

- 複数のアプリケーションを含む
  - マウスの中ボタン or m キーで切り替え
- 動作再生
- キーフレーム動作再生
- 順運動学計算
- 姿勢補間
- 動作補間（2つの動作の補間）
- 動作接続・遷移
- 動作変形
- 逆運動学計算（CCD-IK）



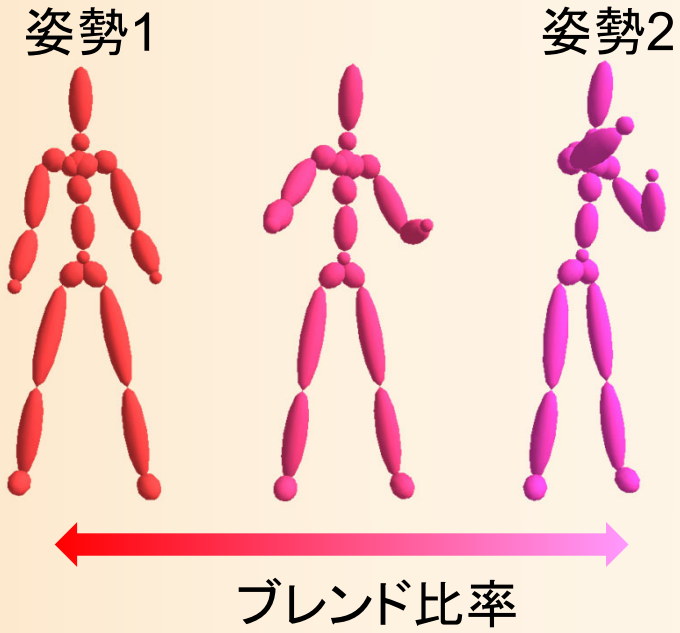


前回までの内容

- 人体モデル（骨格・姿勢・動作）の表現
- 人体モデル・動作データの作成方法
- サンプルプログラム、動作再生
- 順運動学、人体形状変形モデル
- 姿勢補間、キーフレーム動作再生、動作補間
- 動作接続・遷移、動作変形
- 逆運動学、モーションキャプチャ
- 動作生成・制御

姿勢補間

- 基礎技術
  - 2つの姿勢の補間
    - 混合(Blending)、
      補間(Interpolation)、
      合成(Synthesis)など
      いくつかの呼び方がある

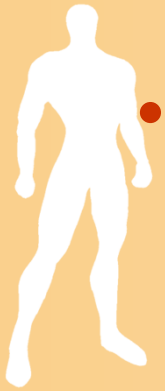


- 姿勢補間の応用例
  - キーフレーム動作再生、動作補間、
    動作接続・遷移、動作変形

動作接続・遷移

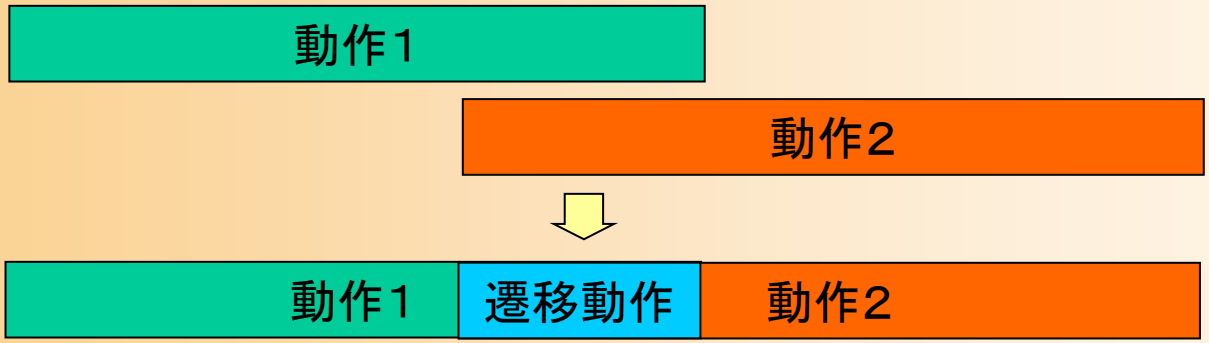
動作接続･遷移

- 動作接続･遷移の原理
- サンプルプログラム
- 動作接続のプログラミング
- 動作接続･遷移のプログラミング
- 動作接続･遷移の拡張
- 動作接続･遷移の応用



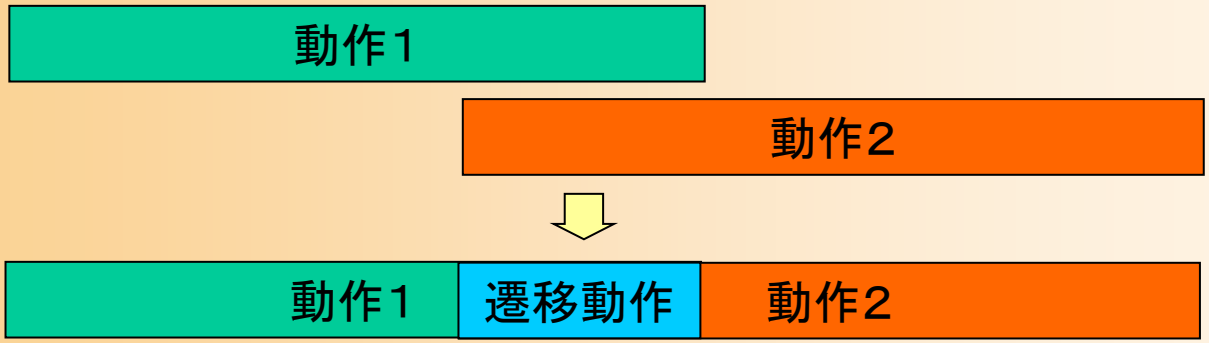
動作接続・遷移

- 動作接続・遷移・合成（トランジション、ブレンド）
  - 2つの既存の動作をつなげて新しい動作を生成
  - 前後の動作の間を滑らかにつなぐ手法
    - 時間に応じた比率で前後の動作の姿勢を補間
    - 前の動作の終了部分と、次の動作の開始部分が、同じような動作である場合に適用可能



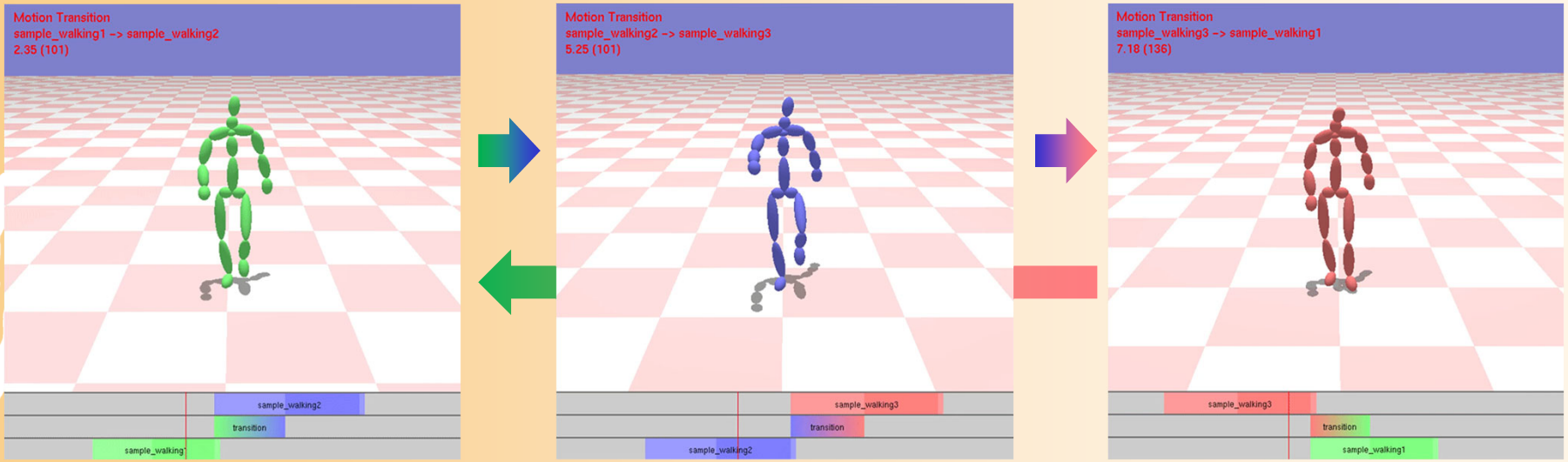
動作接続・遷移

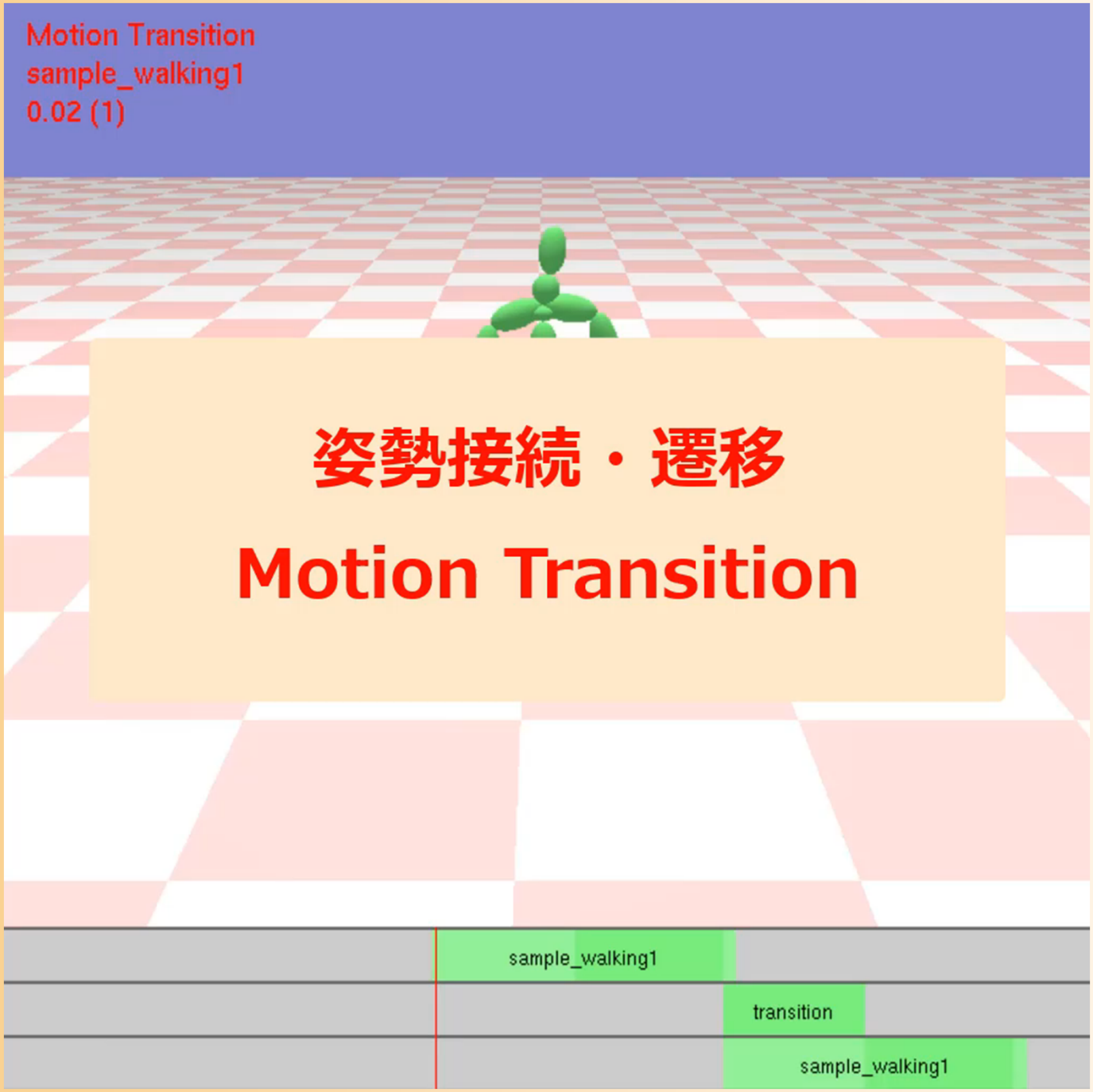
- 動作接続・遷移・合成（トランジション、ブレンド）
  - 接続・遷移・合成などの呼び方がある
  - 厳密に定義をすると、以下のように定義できる
    - 接続は、動作間の位置やタイミングを合わせて接続
    - 遷移は、動作接続時に滑らかな遷移を実現
    - 接続と遷移の組み合わせが必要になる



デモプログラム

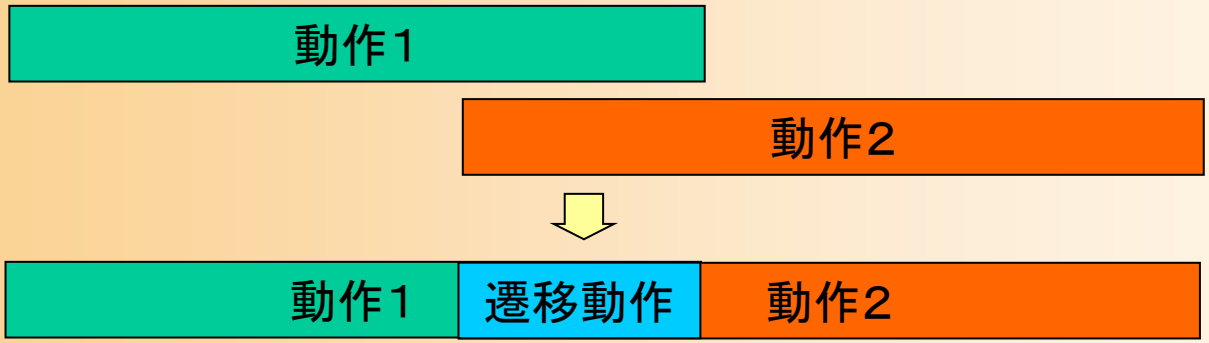
- 動作接続・遷移アプリケーション
  - 動作接続・遷移を含む動作再生
    - 動作接続、動作接続・遷移を切り替え（Dキー）
  - マウス操作（左クリック）で、次の動作を変更
    - 変更なしの場合は、同じ動作に接続・遷移（繰り返し）




Motion Transition
sample_walking1
0.02 (1)
姿勢接続・遷移
Motion Transition
sample_walking1
transition
sample_walking1

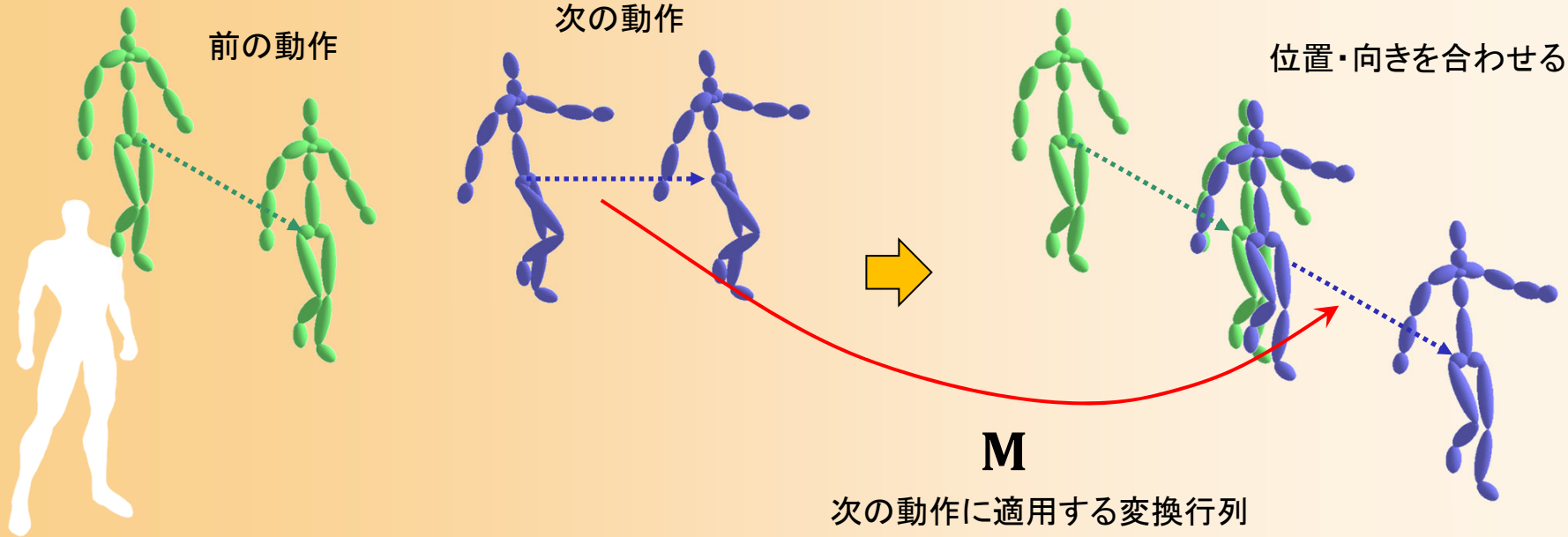
動作接続・遷移の概要

- 動作接続
  - 前の動作の終了部分と、次の動作の開始部分の、位置・向きやタイミングを合わせる
- 動作遷移（遷移動作中の姿勢補間）
  - 時刻 t に対応する、前後の動作の時刻 t_i と重み w_i を決定し、両動作の姿勢を補間



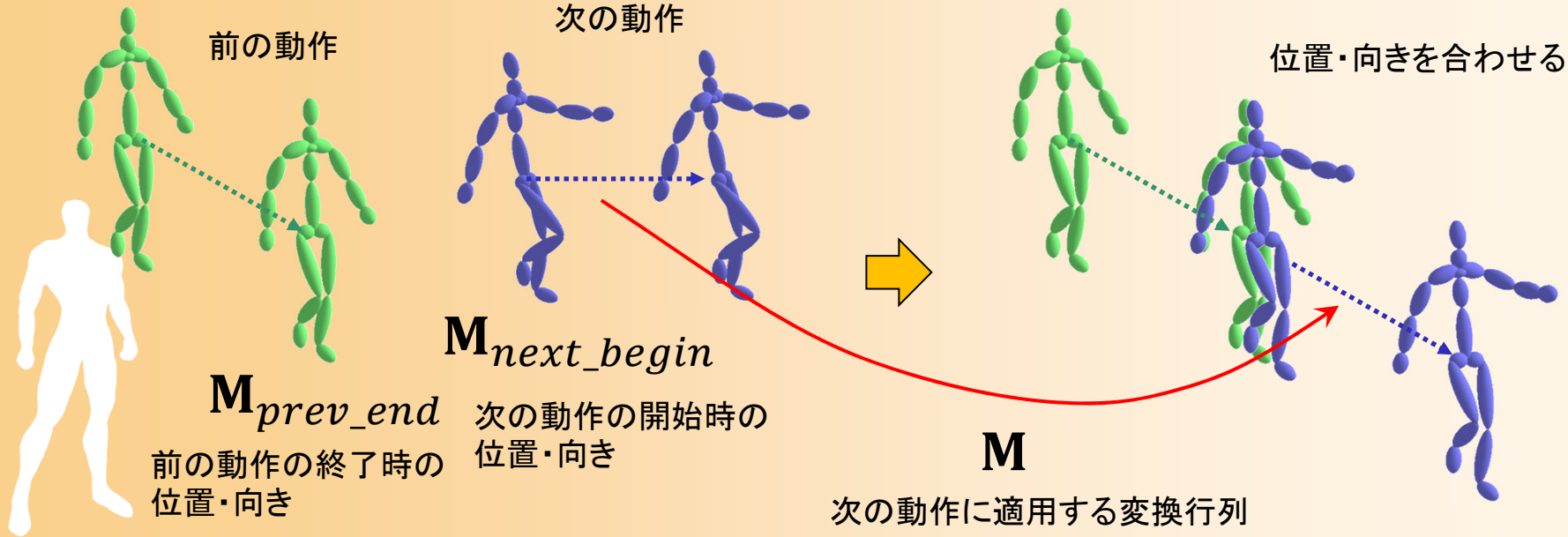
動作接続の実現方法

- 前の動作の終了時の姿勢の位置・向きに、次の動作の開始時の位置・向きがつながるように、次の動作に座標変換を適用
  - 腰の位置・向きを基準に座標変換を計算



動作接続の実現方法

- 前の動作の終了時の姿勢の位置・向きに、次の動作の開始時の位置・向きがつながるように、次の動作に座標変換を適用
  - 腰の位置・向きを基準に座標変換を計算



動作接続の位置・向き

- 2つの動作の位置・向きを合わせる
  - 前の動作の終了時の位置・向きと、次の動作の開始時の位置・向きが一致するように、次の動作の各姿勢に変換行列を適用

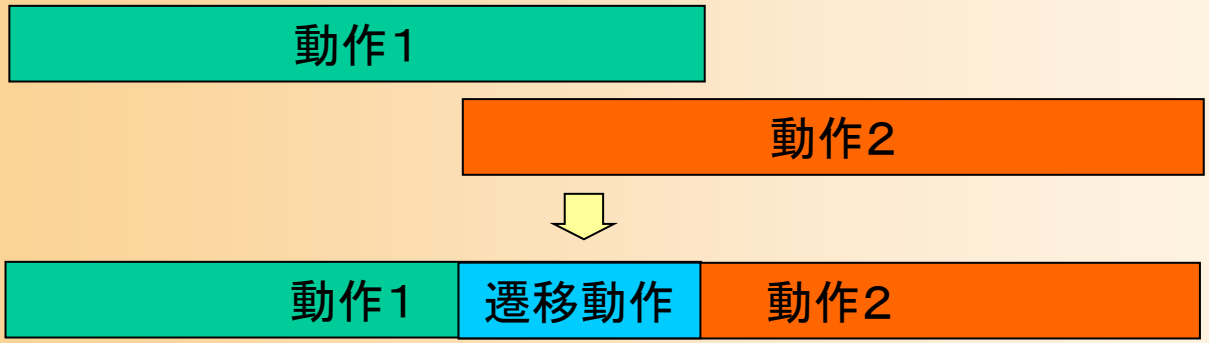
$$\mathbf{M} = \mathbf{M}_{prev\_end}\left(\mathbf{M}_{next\_begin}\right)^{-1}$$

次の動作に適用する変換行列　　前の動作の終了時の位置・向き　　次の動作の開始時の位置・向き

- 実際には上の計算方法では、上下回転・左右回転・上下位置のずれを含む場合、接続した動作の方向・位置がおかしくなる
  - 各行列から水平回転・水平移動の成分のみを取り出して計算すると、より適切な変換行列が計算できる

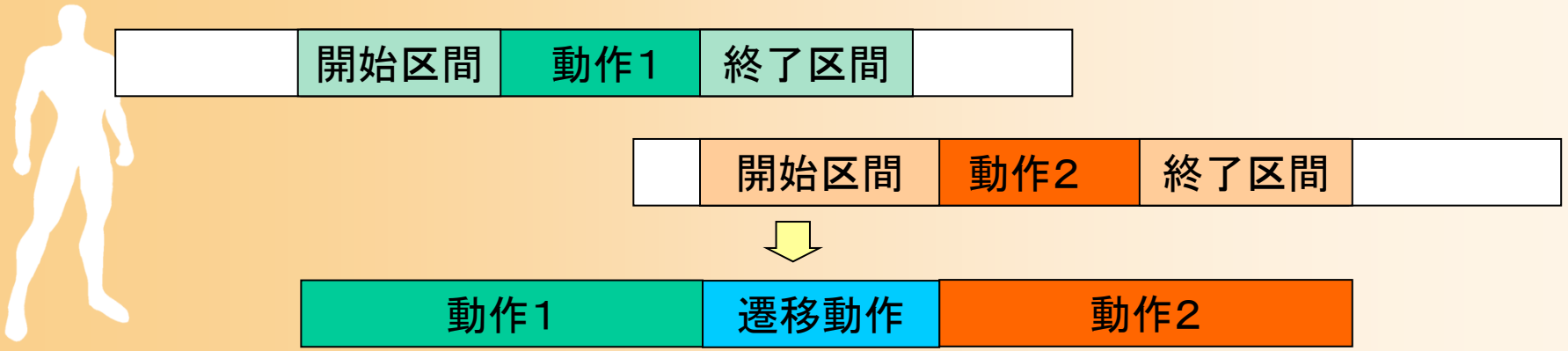
動作遷移の実現方法

- 動作遷移のタイミングを決定
  - どのタイミングで次の動作を開始するか
  - どのタイミングで前後の動作の補間を行うか
- 動作遷移を含む動作再生
  - タイミングにもとづいて、遷移動作区間では、重みを変えながら、前後の動作の姿勢を補間



動作遷移のタイミング

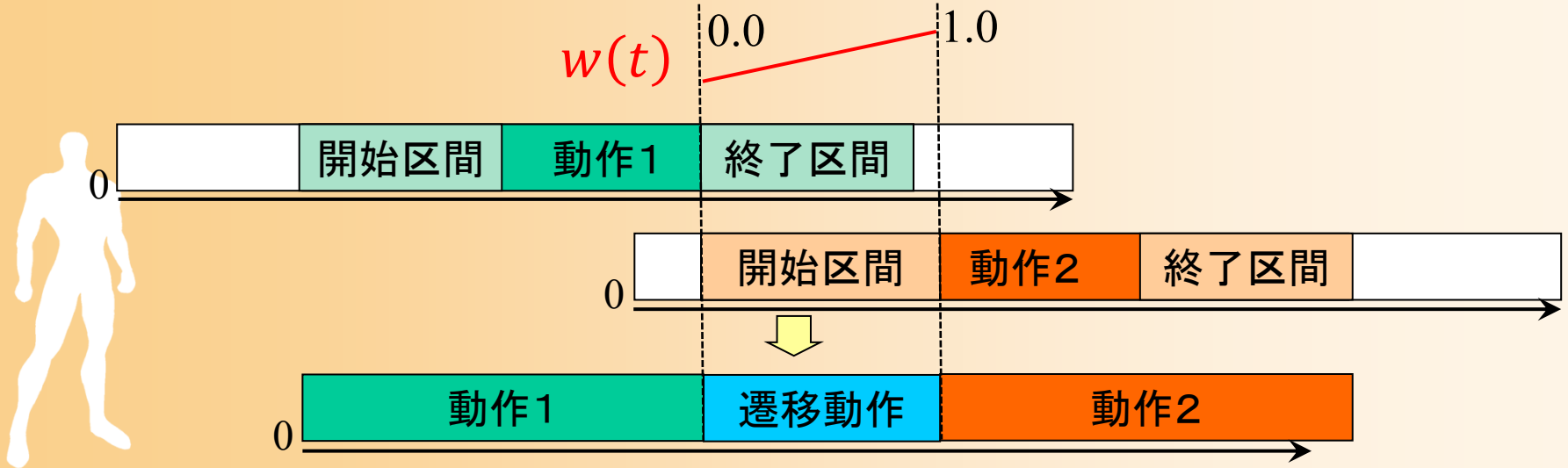
- 基本的には、前の動作の主要な部分が終わってから、次の動作を開始する必要がある
  - ただし、逆に、次の動作の開始が遅すぎると、滑らかな遷移を行うことができない
- 動作データに、主要な部分と、それ以外の開始・終了区間の情報が設定されていれば、主要な部分以外で動作遷移を行うように、タイミングを決められる



動作遷移の再生

- 遷移中は、両方の動作から姿勢を取得して、補間
  - 現在時刻 t を、各動作の時刻 t_{next} , t_{prev} に変換
  - 現在時刻 t にもとづいて、補間の重み $w(t)$ を計算
    - $0.0 \rightarrow 1.0$ に単調増加するような重みを計算

$$\mathbf{p}(t) = w(t)\mathbf{P}_{next}(t_{next}) + \big(1 - w(t)\big)\mathbf{P}_{prev}\big(t_{prev}\big)$$



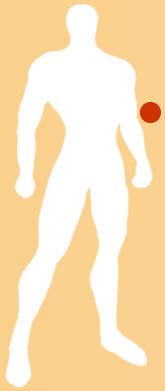
今日の内容

- 前回までの復習
- 動作接続･遷移
  - 動作接続･遷移の原理
  - サンプルプログラム
  - 動作接続のプログラミング
  - 動作遷移のプログラミング
  - 動作接続･遷移の拡張
  - 動作接続･遷移の応用
- 動作変形

動作接続・遷移（続き）

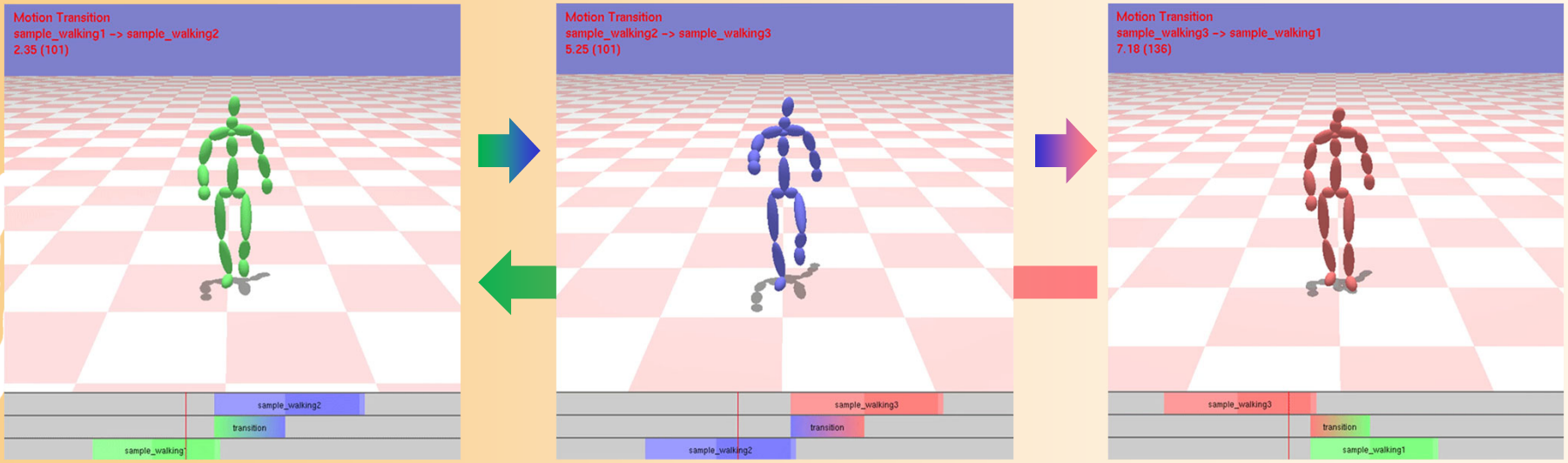
動作接続･遷移

- 動作接続･遷移の原理
- サンプルプログラム
- 動作接続のプログラミング
- 動作接続･遷移のプログラミング
- 動作接続･遷移の拡張
- 動作接続･遷移の応用



プログラミング演習

- 動作接続・遷移アプリケーション
  - 動作接続・遷移を含む動作再生
    - 動作接続、動作接続・遷移を切り替え（Dキー）
  - マウス操作（左クリック）で、次の動作を変更
    - 変更なしの場合は、同じ動作に接続・遷移（繰り返し）



動作接続・遷移アプリケーション

- MotionTransitionApp
  - 初期化時に、再生する複数の動作の読み込みと設定
  - マウス操作（左クリック）に応じて、次の動作を変更
  - アニメーション処理（アニメーション関数）
    - 動作接続・遷移の初期化・姿勢取得処理を呼び出し
  - タイムラインの描画（Timeline.h/cpp）
- MotionConnection、MotionTransition
  - 動作接続のための変換行列の計算（各自作成）
    - ComputeConnectionTransformation関数
  - 動作遷移のための姿勢の計算（各自作成）
    - MotionTransition::Init関数、MotionTransition::GetPosture関数

動作接続・遷移アプリケーション(2)

- クラス定義(MotionTransitionApp)

```
class  MotionTransitionApp : public GLUTBaseApp
{
  protected:
    // 動作遷移・接続(を含む動作再生)の入力情報
    // 動作データリスト
    vector< MotionInfo * >  motion_list;
    // 現在の再生動作番号
    int  curr_motion_no;
    // 現在の動作の位置・向きに対する変換行列
    Matrix4f  curr_motion_mat;
    // 現在の動作の再生開始時刻(グローバル時刻)
    float  curr_start_time;
    // 次の再生動作番号
    int  next_motion_no;
    // 実行待ちの動作番号
    int  waiting_motion_no;
 protected:
    // 動作遷移のための変数
    // 動作接続・遷移
    MotionTransition *  transition;
    ...
```

動作接続・遷移アプリケーション(2)

- メンバ変数定義(MotionTransitionApp)
  - 入力情報
    - 動作データのリスト
    - 現在の動作、次の動作、待ち動作の番号
    - 現在の動作の開始位置・向きと開始時刻
  - 動作接続・遷移のための情報
    - 動作接続・遷移機能(MotionTransitionクラス)
  - 動作再生のための変数
  - 情報表示用の変数
    - タイムライン描画機能(Timelineクラス)

動作接続・遷移アプリケーション(2)

- メンバ関数(MotionTransitionApp)
  - 初期化処理で、サンプル動作の読み込み
  - マウスクリック処理で、待ち動作を変更
  - アニメーション処理で、動作接続・遷移を含む動作再生
    - MotionTransitionクラス or MotionConnectionクラスを利用
  - 描画処理で、現在姿勢やタイムラインを描画


```
class  MotionTransitionApp : public GLUTBaseApp
{
    virtual void  Initialize();
    virtual void  Start();
    virtual void  Display();
    virtual void  MouseClick( int button, int state, int mx, int my );
    virtual void  Keyboard( unsigned char key, int mx, int my );
    virtual void  Animation( float delta );
```

動作接続・遷移クラス(1)

- 動作接続・遷移クラスのメンバ変数

```
//  動作接続・遷移クラス
class  MotionTransition
{
  protected:
    // 動作遷移の入力情報
    // 前の動作
    const MotionInfo *  prev_motion;
    // 後の動作
    const MotionInfo *  next_motion;
    // 前の動作の変換行列
    Matrix4f  prev_motion_mat;
    // 前の動作の開始時刻(グローバル時刻)
    float  prev_begin_time;

    // 動作遷移のための情報

    // 動作遷移中の姿勢補間のための情報
```

動作のメタ情報を表す
MotionInfo構造体を使用

動作接続・遷移クラス(2)

- 動作接続・遷移クラスのメンバ関数

```
//  動作接続・遷移クラス
class  MotionTransition
{
    // 動作遷移の状態
    enum  MotionTransitionState {
        MT_PREV_MOTION, MT_IN_TRANSITION, MT_NEXT_MOTION };

    // 動作接続・遷移の初期化
    virtual bool  Init(
        const MotionInfo * prev_motion, const MotionInfo * next_motion,
        const Matrix4f & prev_motion_mat, float prev_begin_time );

    // 動作接続・遷移の姿勢計算
    virtual MotionTransitionState  GetPosture( float time,
        Posture * posture );
}
```

動作遷移の状態を表す
GetPosture関数の戻り値として使用
前の動作の再生中、動作遷移中、
次の動作の再生中のいずれかの値

これらのメンバ関数の
一部の処理を作成

動作接続・遷移クラス(3)

- 動作接続クラス(MotionConnection)の定義
  - 動作接続・遷移クラス(MotionTransition)を継承
    - 説明用に動作接続に機能を限定したクラスを作成

```
//  動作接続クラス
class  MotionConnection : public MotionTransition
{
   // 動作接続・遷移の初期化
   virtual bool  Init(
      const MotionInfo * prev_motion, const MotionInfo * next_motion,
      const Matrix4f & prev_motion_mat, float prev_begin_time );

   // 動作接続・遷移の姿勢計算
   virtual MotionTransitionState  GetPosture( float time,
      Posture * posture );
}
```


これらのメンバ関数の
処理は作成済み

動作接続･遷移の切り替え

- キー操作で、どちらを使用するかを切り替え
  - MotionTransition * 型の変数 transition は、MotionTransition と MotionConnection のどちらのオブジェクトも参照できる

```
class  MotionTransitionApp : public GLUTBaseApp
{
    ...
    // 動作接続･遷移
    MotionTransition *  transition;
    ...
};
void  MotionTransitionApp::Start()
{
    ...
    if ( enable_transition )
        transition = new MotionTransition();
    else
        transition = new MotionConnection();
    ...
```

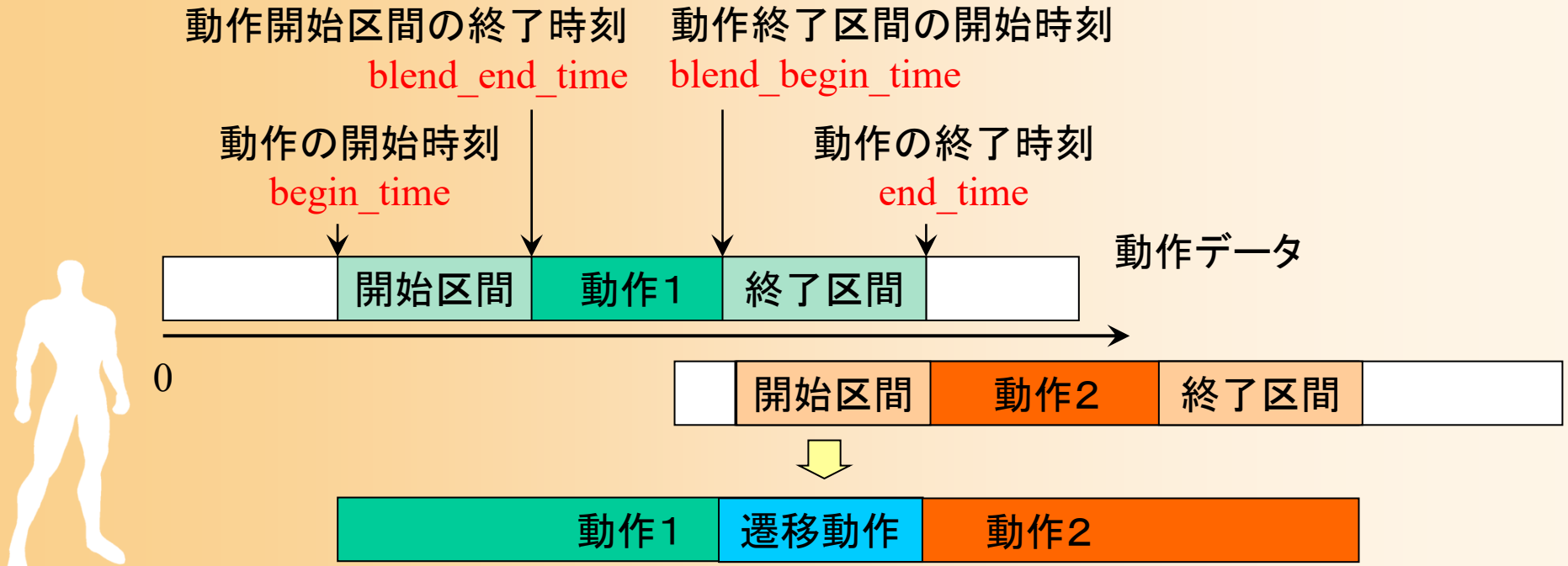
動作接続・遷移に用いる動作情報

- 動作のメタ情報の構造体（MotionTransition.h）

```
// 動作のメタ情報を表す構造体
struct  MotionInfo
{
    // 動作情報
    Motion *  motion;
    // 動作の開始・終了時刻（動作のローカル時間）
    float  begin_time;
    float  end_time;
    // ブレンド区間の終了・開始時刻（動作のローカル時間）
    float  blend_end_time;
    float  blend_begin_time;
    //次の動作への動作接続のための基準部位
    int  base_segment_no;
    ....
```

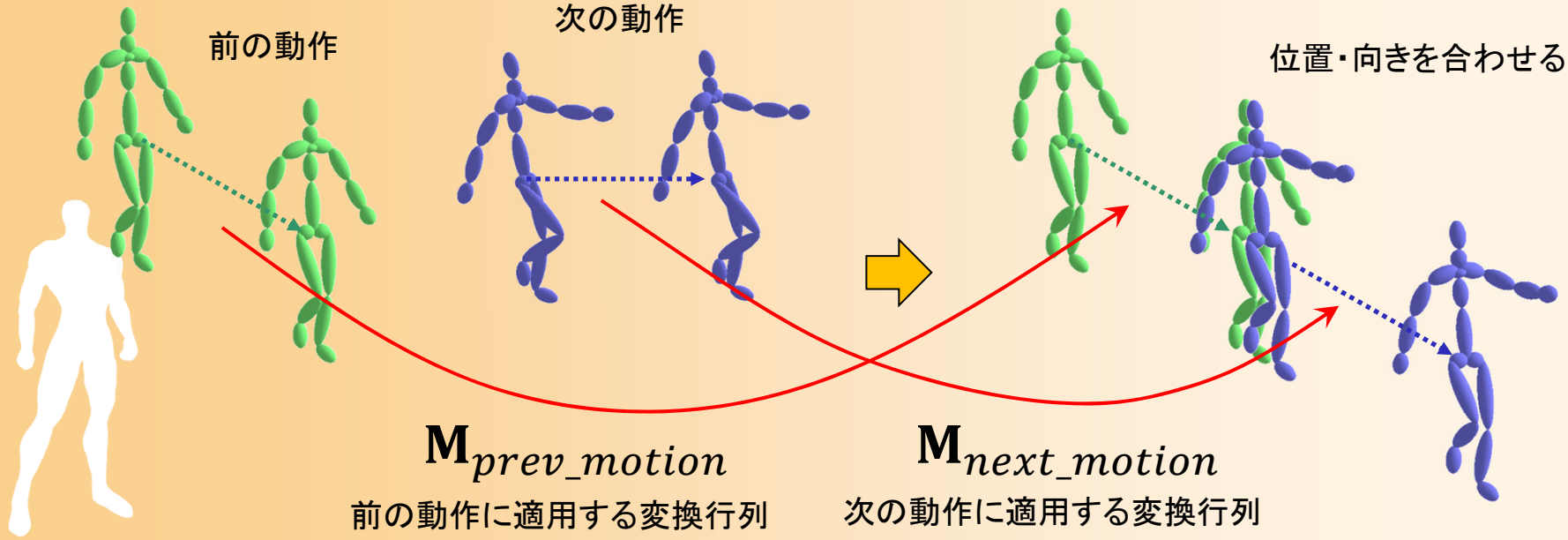
動作接続・遷移に用いる動作情報

- 動作のメタ情報が持つ時刻の情報
  - 動作の開始・終了時刻（動作のローカル時間）
  - ブレンド終了・開始時刻（動作のローカル時間）



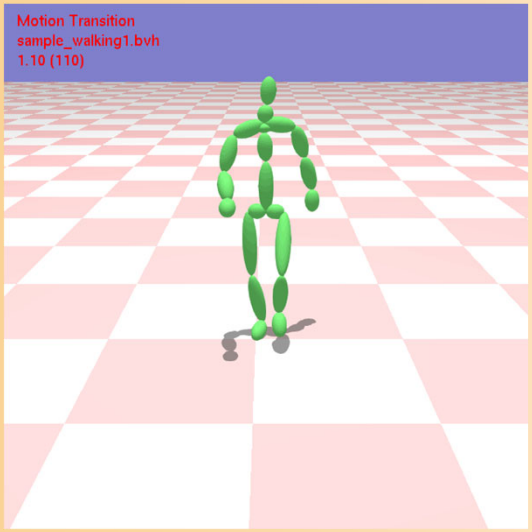
動作接続に必要な情報

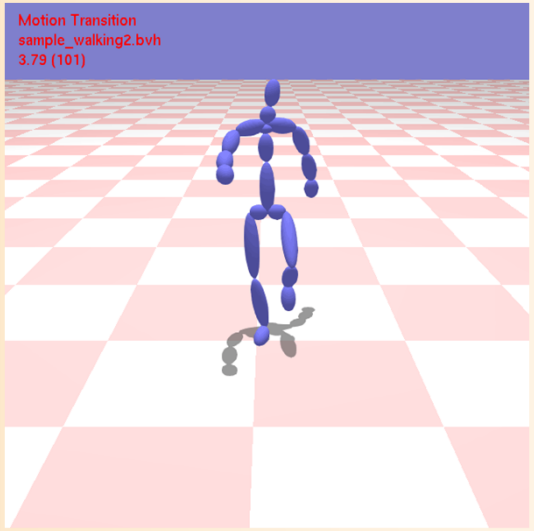
- 動作接続の変換行列の情報
  - 前の動作に適用する変換行列（prev_motion_mat）
  - 次の動作に適用する変換行列（next_motion_mat）



サンプル動作(1)

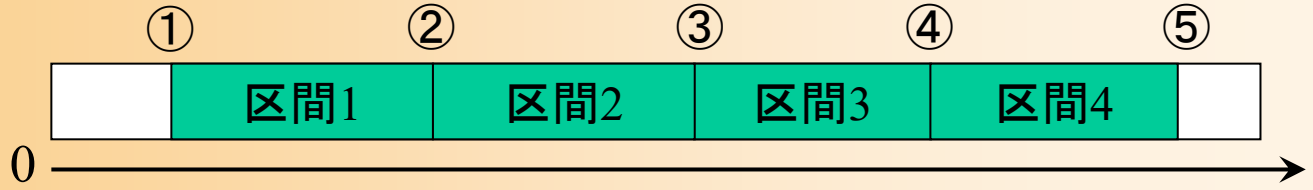
- デモプログラム(サンプルプログラム)では、異なるスタイルの歩行動作を使用
  - 繰り返し歩行動作中の1サイクル分を使用
  - 3つの異なるスタイルの歩行動作を使用





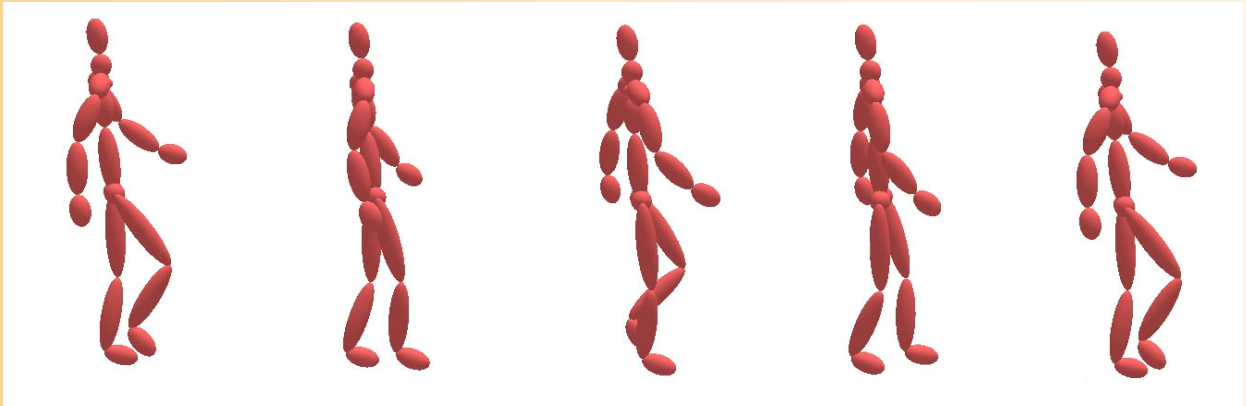
サンプル動作(2)

- デモプログラム(サンプルプログラム)では、異なるスタイルの歩行動作を使用
  - 繰り返し歩行動作中の1サイクル分を使用
  - 5つのキー時刻を設定
    1. 右足を上げ始める(動作開始)
    2. 右足を着く(ブレンド区間終了)
    3. 左足を上げ始める
    4. 左足を着く(ブレンド区間開始)
    5. 右足を上げ始める(動作終了)



サンプル動作(3)

- デモプログラム(サンプルプログラム)では、異なるスタイルの歩行動作を使用
  - 繰り返し歩行動作中の1サイクル分を使用
  - 5つのキー時刻を設定



サンプル動作(4)

- デモプログラム(サンプルプログラム)では、異なるスタイルの歩行動作を使用
  - 繰り返し歩行動作中の1サイクル分を使用
  - 5つのキー時刻を設定
  - 最初と最後の区間を、ブレンド区間とする
    - 本来は、動作遷移時は、前後の動作で同じ動き行う区間同士をブレンドすることが望ましいが、動作接続を分かりやすくするために、このブレンド区間を使用
    - 実際には、動作終了時のブレンド区間は短いため、次の動作の動作開始時のブレンド区間が、動作遷移中の主な動作になる

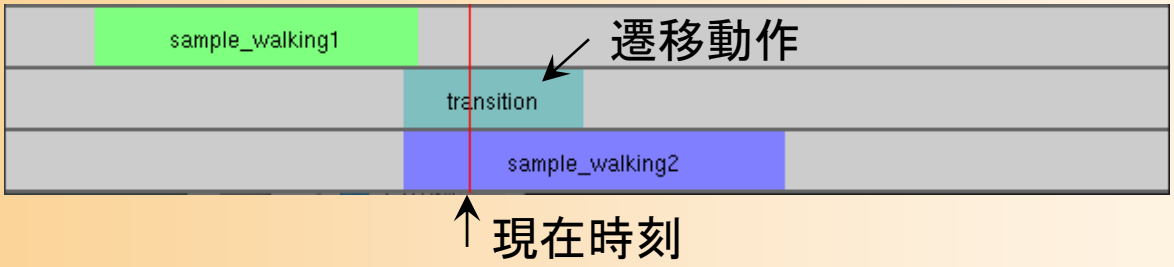
サンプル動作の読み込み

- LoadSampleMotions関数で読み込み
  - 動作に関する情報を関数内に記述
  - BVHファイルを読み込み、動作データの配列を出力

```
// サンプル動作セットの読み込み
const Skeleton *  LoadSampleMotions(
    vector< MotionInfo * > &  motion_list, const Skeleton * body )
{
    const int  num_motions = 3;
    const int  num_keytimes = 5;
    const char *  sample_motions[ num_motions ] = {
        "sample_walking1.bvh", "sample_walking2.bvh",
        "sample_walking3.bvh" };
    const float  sample_keytimes[ num_motions ][ num_keytimes ] = {
        { 2.35f, 3.00f, 3.08f, 3.68f, 3.74f },
        { 1.30f, 2.07f, 2.12f, 2.88f, 2.94f },
        { 1.20f, 2.00f, 2.08f, 2.80f, 2.86f } };
    ...
```

タイムラインの描画

- 現在の動作、次の動作、遷移区間の時間を可視化
  - Timeline クラスを利用（Timeline.h/cpp）
    - タイムラインの設定（行数、全体の時間範囲）
    - 各要素の設定（開始・終了時間、色、ラベル）
  - UpdateTimeline 関数
    - アニメーション処理から呼び出し
    - 引数として渡された情報を、タイムラインに設定
  - 現在時刻の位置を固定して、アニメーションの進行に合わせて、表示する時間の範囲を変更

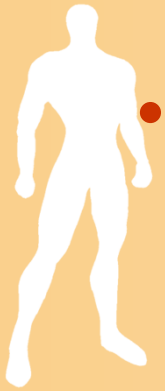


プログラミング演習の手順

- ２段階に分けて実装する
  - 動作接続のみの実現
    - ComputeConnectionTransformation グローバル関数
      - MotionConnection::Init 関数や MotionTransition::Init 関数から呼ばれる
  - 動作接続＋動作遷移の実現
    - MotionTransition::Init メンバ関数
    - MotionTransition::GetPosture メンバ関数
- キー操作（Dキー）で、モードを切替可能
  - 動作接続の作成後、動作接続のみで動作確認

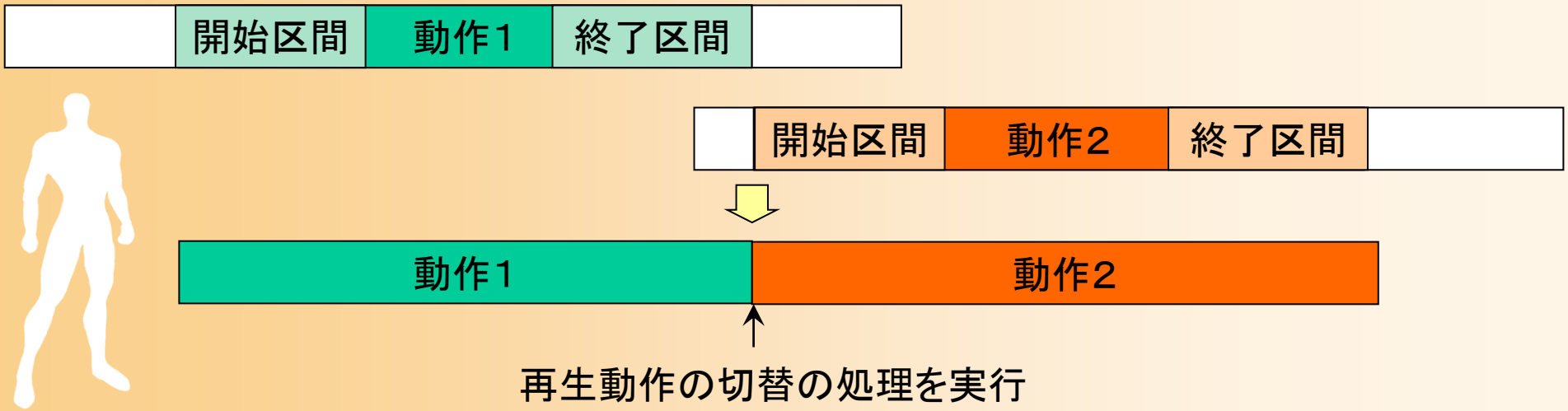
動作接続･遷移

- 動作接続･遷移の原理
- サンプルプログラム
- 動作接続のプログラミング
- 動作接続･遷移のプログラミング
- 動作接続･遷移の拡張
- 動作接続･遷移の応用



動作接続のタイミング

- 動作接続のタイミング
  - 前の動作の終了時刻と次の動作の開始時刻を合わせる
    - 開始･終了ブレンド区間は無視
  - 切替時、次の動作に適用する変換行列を計算



動作接続のプログラミング(1)

- 動作接続クラスのメンバ関数
  - 初期化
  - 姿勢計算

```
//  動作接続クラス
class  MotionConnection : public MotionTransition
{
   // 動作接続・遷移の初期化
   virtual bool  Init(
      const MotionInfo * prev_motion, const MotionInfo * next_motion,
      const Matrix4f & prev_motion_mat, float prev_begin_time );

   // 動作接続・遷移の姿勢計算
   virtual MotionTransitionState  GetPosture( float time,
      Posture * posture );
}
```

動作接続のプログラミング(2)

- 動作接続クラスのメンバ関数
  - 初期化
    - 次の動作の開始時刻(next_begin_time)の設定
      - ブレンド区間の開始・終了時刻の設定は不要
    - 動作接続のための変換行列(next_motion_mat)の計算
      - ComputeConnectionTransformation関数を呼び出し
  - 姿勢計算
    - 入力された時刻が、前の動作と次の動作のどちらに対応するかを判定
    - 前の動作 or 次の動作の姿勢を取得して、座標変換を行い、出力する

動作接続のプログラミング（3）

- 動作接続のための変換行列の計算

```
// 2つの姿勢の位置・向きを合わせるための変換行列を計算
//（next_frame の位置・向きを、prev_frame の位置向きに合わせる
    ための変換行列 trans_mat を計算）
void  ComputeConnectionTransformation(
   const Matrix4f & prev_frame, const Matrix4f & next_frame,
   Matrix4f & trans_mat )
{
```

$$\mathbf{M} = \mathbf{M}_{prev\_end}\left(\mathbf{M}_{next\_begin}\right)^{-1}$$

各自作成

```
        trans_mat   prev_frame    next_frame
}

// 姿勢の位置・向きに変換行列を適用
void  TransformPosture( const Matrix4f & trans, Posture & posture );
```

動作接続のプログラミング（4）

- 動作接続のための変換行列の計算の改良
  - 実際には、前のスライドの計算方法では、変換行列に上下回転・左右回転・上下位置のずれを含む場合、接続した動作の方向がおかしくなる
  - 水平方向の向きの成分のみの回転行列を用いるようにすることで、より適切な変換行列を計算できる

$$\mathbf{M} = \mathbf{M}_{prev\_end}\left(\mathbf{M}_{next\_begin}\right)^{-1}$$

水平方向の回転成分のみの回転行列に変換する

動作接続のプログラミング（5）

- 水平方向の回転を表す回転行列への変換
  - 水平方向の回転行列は下に示すような形になる

$$\mathbf{M} = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix} \quad \cdots ①$$

  - 他の回転が混ざっている場合も、水平方向の回転行列とみなして、水平方向の回転角度を計算する

$$\mathbf{M} = \begin{pmatrix} x_x & 0.0\cdots & z_x \\ 0.0\cdots & 0.9\cdots & 0.0\cdots \\ x_z & 0.0\cdots & z_z \end{pmatrix} \qquad \theta = \tan^{-1}\frac{z_x}{z_z} \qquad \cdots ②$$

  - もとの回転行列から、式② で計算した回転角度を、式① に代入することで、水平方向の回転行列に変換できる

動作接続のプログラミング（5）

- 水平方向の回転を表す回転行列への変換
  - 水平方向の回転行列は下に示すような形になる

$$\mathbf{M} = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix} \cdots$$

vecmathの関数（Matrix3fのrotY関数）で計算可能

  - 他の回転が混ざっている場合も、水平方向の回転行列とみなして、水平方向の回転角度を計算する

$$\mathbf{M} = \left(\right.$$

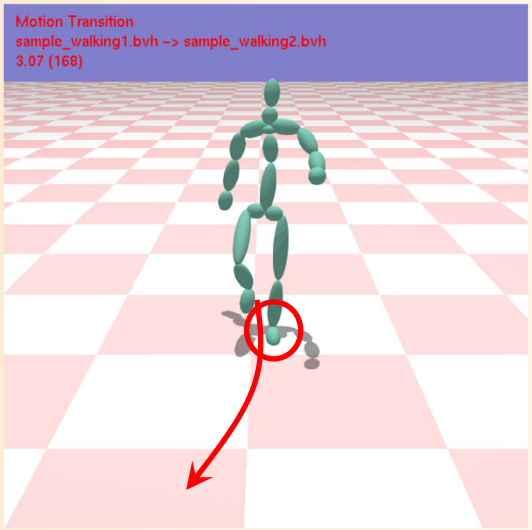
C言語の標準関数（atan2関数）で計算可能

$$\theta = \tan^{-1}\frac{z_x}{z_z} \quad \cdots ②$$

  - もとの回転行列から、式② で計算した回転角度を、式① に代入することで、水平方向の回転行列に変換できる

動作接続の改良(1)

- 水平方向の位置・向きだけを合わせるように修正しても、まだ問題が起きる
  - 水平方向の向きが意図した通りにならない
    - 例: 正面方向に歩行動作を繰り返すように接続したいが、横方向にずれていく
  - 動作遷移を行っても支点の足の位置がずれる
    - 例: 動作接続・遷移のときに、軸足(支点)となっている足が地面の上で固定されず、横に滑る



動作接続の改良(2)

- 解決方法
  - 水平方向の向きが意図した通りにならない
    → 水平方向の向きを別のパラメタで指定する
  - 動作遷移を行っても支点の足の位置がずれる
    → 腰(ルート)ではなく、任意の体節の位置を合わせる
- 変換行列計算の入力と処理を変更

```
//  2つの姿勢の位置・向きを合わせるための変換行列を計算
//（next_pose の基準部位 base_segment の水平位置を prev_pose に合わせて、
//  水平向き next_ori を prev_ori に合わせる）
void  ComputeConnectionTransformation(
    const Posture & prev_pose, float prev_ori, const Posture & next_pose,
    float next_ori, int base_segment, Matrix4f & trans_mat )
```

動作接続の改良(3)

- 変換行列計算の入力と処理を変更
  - サンプルプログラムでは、以下の拡張版の関数を定義して、呼び出すようになっている
    - 動作データに接続の基準部位(軸足)の情報を入力
    - 前の動作と後の動作の水平向きの情報を入力
  - 最初は、拡張版の関数から、基本的な処理を行う関数を呼び出し

```
//  2つの姿勢の位置・向きを合わせるための変換行列を計算
//（next_pose の基準部位 base_segment の水平位置を prev_pose に合わせて、
//  水平向き next_ori を prev_ori に合わせる）
void  ComputeConnectionTransformation(
    const Posture & prev_pose, float prev_ori, const Posture & next_pose,
    float next_ori, int base_segment, Matrix4f & trans_mat )
```

動作接続の改良(4)

```
void  ComputeConnectionTransformation(
    const Posture & prev_pose, float prev_ori, const Posture & next_pose,
    float next_ori, int base_segment, Matrix4f & trans_mat )
{
    ....
    // 上記の処理が未実装であれば、標準的な処理を呼び出す
    ComputeConnectionTransformation( prev_pose, next_pose, trans_mat );
}

bool  MotionTransition::Init( ・・・ )
{
    ....
    base_segment_no = prev_motion->base_segment_no;
    ComputeConnectionTransformation(
        *prev_motion_posture, prev_ori, *next_motion_posture, next_ori,
        base_segment_no, next_motion_mat );
    ....
}
```

追加の入力は使用しない関数を呼び出し

動作に設定された基準体節番号の情報を使用

前の動作の終了時の水平向き(ワールド座標系)
次の動作の開始時の水平向き(動作データのワールド座標系)

動作接続の改良(5)

```
void  ComputeConnectionTransformation(
    const Posture & prev_pose, float prev_ori, const Posture & next_pose,
    float next_ori, int base_segment, Matrix4f & trans_mat )
{
    // 基準部位としてルート体節が指定された場合は、
    // 2つの姿勢のルートの位置を取得
    if ( base_segment == 0 )
    {
    }
    // 基準部位としてルート体節以外が指定された場合は、
    // 順運動学計算により、2つの姿勢の指定部位の位置を計算
    else
    {
    }
    // 前の姿勢の位置・向きを表す変換行列を計算
    // 次の姿勢の位置・向きを表す変換行列を計算
    // 座標変換を計算
}
```

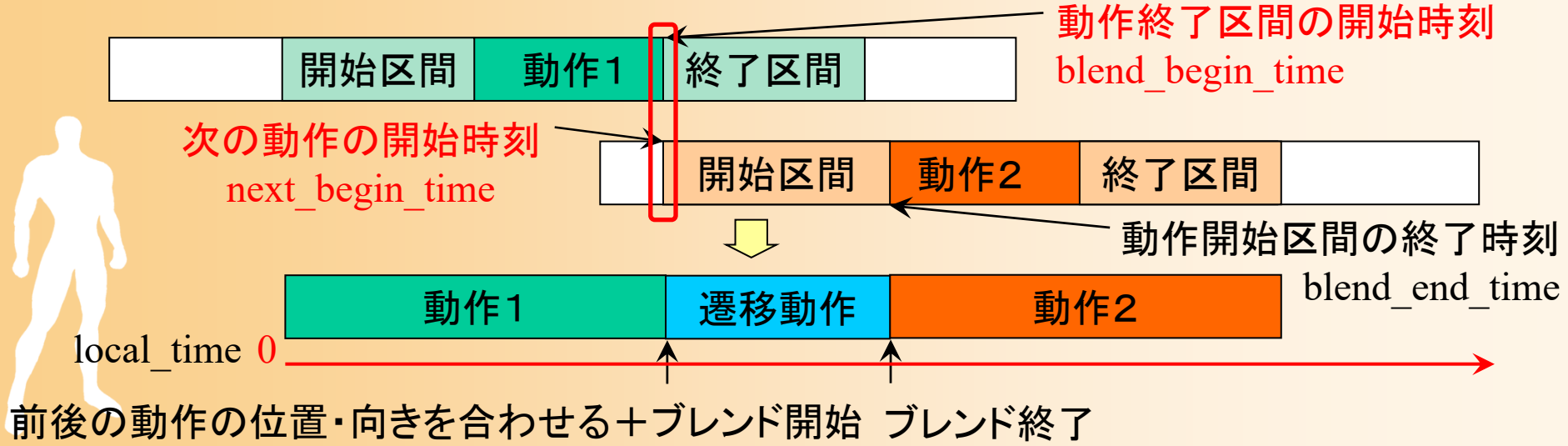
入力にもとづいて
変換行列の
位置・水平向きを計算

動作接続･遷移

- 動作接続･遷移の原理
- サンプルプログラム
- 動作接続のプログラミング
- 動作接続･遷移のプログラミング
- 動作接続･遷移の拡張
- 動作接続･遷移の応用

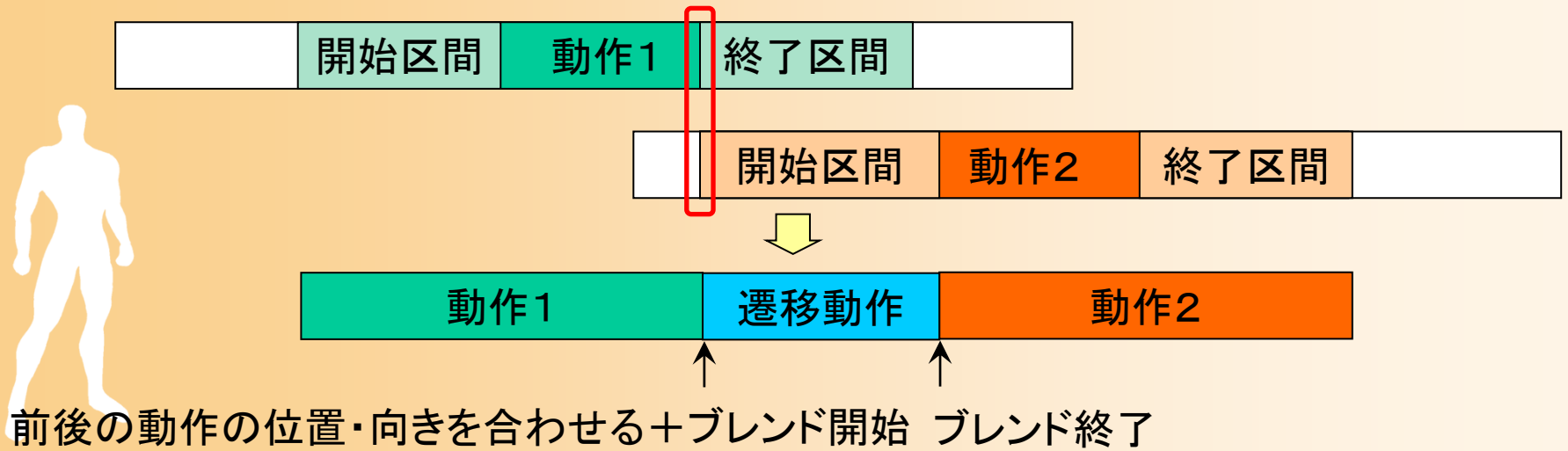
動作遷移のタイミング

- 動作遷移の区間の決定や姿勢補間の方法にはさまざまなやり方がある
  - 今回は、以下の2つのタイミングを合わせる
    - 前の動作の終了区間の開始時刻
    - 次の動作の開始区間の開始時刻（＝動作の開始時刻）



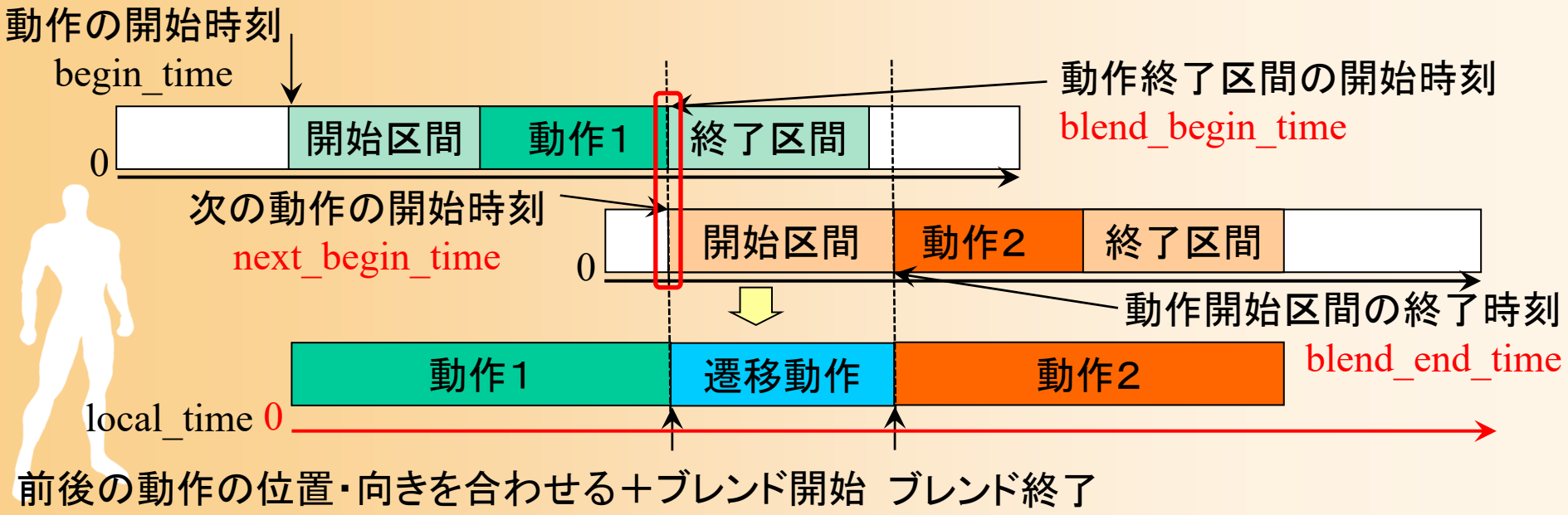
動作遷移の再生処理(1)

- 前後の動作にもとづいて、動作遷移のタイミングを決定
- 動作遷移中は、両方の動作の姿勢を補間
- 動作遷移が終了したら、次の動作に切替



動作遷移の再生処理(2)

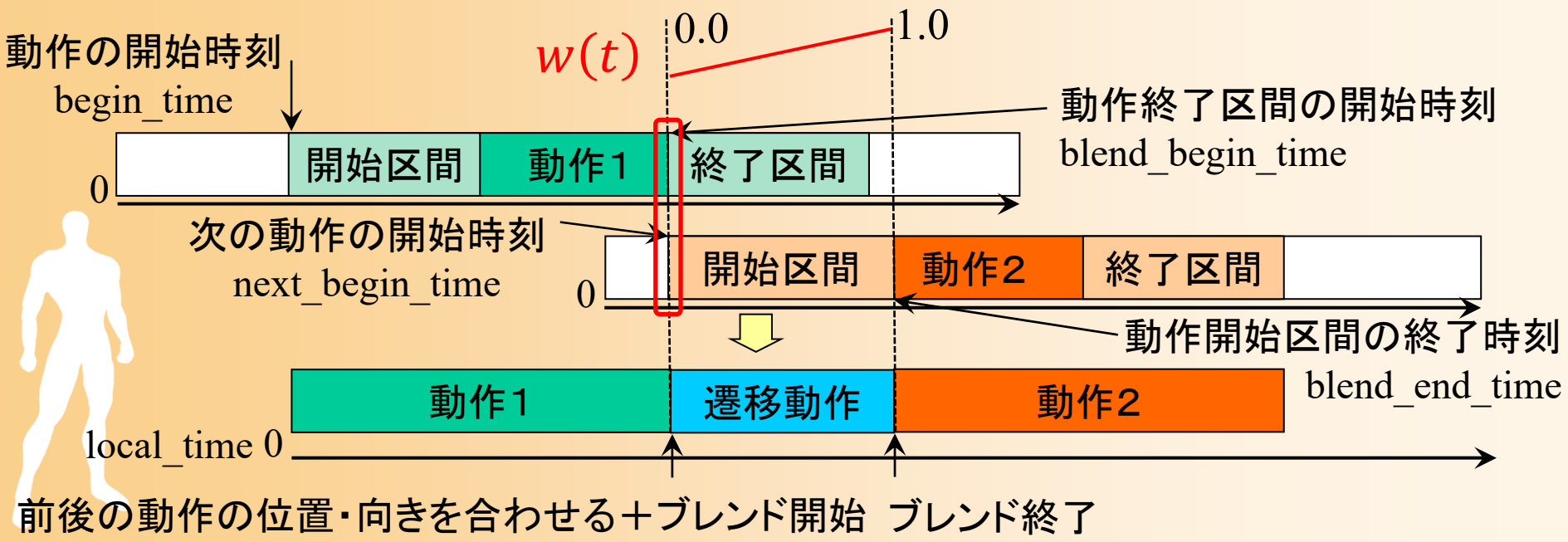
- 動作遷移のタイミングを決定
  - 次の動作の開始時刻(next_begin_time)
  - 動作ブレンドの開始時刻(blend_begin_time)
  - 動作ブレンドの終了時刻(blend_end_time)
    - 現在の動作の開始時間を基準とするローカル時刻で表す



動作遷移の再生処理(3)

- 遷移中は、両方の動作から姿勢を取得して、補間
  - 現在時刻 t を、各動作の時刻 t_{next} , t_{prev} に変換
  - 現在時刻 t にもとづいて、補間の重み $w(t)$ を計算

$$\mathbf{p}(t) = w(t)\mathbf{P}_{next}(t_{next}) + \big(1 - w(t)\big)\mathbf{P}_{prev}\big(t_{prev}\big)$$



動作接続・遷移のプログラミング(1)

- 動作接続・遷移クラスのメンバ関数
  - 初期化
  - 姿勢計算

```
//  動作接続・遷移クラス
class  MotionTransition
{
    // 動作接続・遷移の初期化
    virtual bool  Init(
        const MotionInfo * prev_motion, const MotionInfo * next_motion,
        const Matrix4f & prev_motion_mat, float prev_begin_time );

    // 動作接続・遷移の姿勢計算
    virtual MotionTransitionState  GetPosture( float time,
        Posture * posture );
}
```

動作接続・遷移のプログラミング(2)

- 動作接続クラスのメンバ関数
  - 初期化
    - 次の動作の開始時刻(next_begin_time)の設定
    - ブレンド区間の開始・終了時刻(blend_begin_time, blend_end_time)の設定
    - 動作接続のための変換行列の計算
  - 姿勢計算
    - 入力された時刻が、前の動作、遷移中、次の動作、のどの状態に対応するかを判定
    - 前の動作・次の動作の姿勢を取得して座標変換
    - 動作遷移中であれば、姿勢補間を行い、出力する

赤字は動作接続からの追加

動作接続・遷移のプログラミング(3)

- 動作接続・遷移の初期化
  - 動作接続・遷移のタイミングを決定

```
//  動作接続・遷移の初期化
bool  MotionTransition::Init(
    const MotionInfo * prev_motion, const MotionInfo * next_motion,
    const Matrix4f & prev_motion_mat, float prev_begin_time )
{
    ...
    // 次の動作を開始する時刻
    transition->next_begin_time = ???;
    // 動作遷移のための動作ブレンドを行う開始時刻
    transition->blend_begin_time = ???;
    // 動作遷移のための動作ブレンドを行う終了時刻
    transition->blend_end_time = ???;
    ...
}
```

各自作成

動作接続・遷移のプログラミング(4)

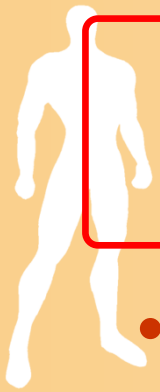
- 姿勢補間の重みの計算
  - 現在時刻に応じて、前後の動作の姿勢補間の重みを計算し、姿勢補間を適用

```
//  動作接続・遷移の姿勢計算
MotionTransition::MotionTransitionState  MotionTransition::GetPosture(
    float time, Posture * posture )
{
    // 前の動作の姿勢を取得、座標変換を適用
    ・・・
    // 後の動作の姿勢を取得、座標変換を適用
    next_motion_local_time = ???;
    ・・・
    // 姿勢補間の重みを計算
    blend_ratio = ???;
    // 前後の動作の姿勢を補間
    MyPostureInterpolation( ・・・ );
    ・・・
```

各自作成

今日の内容

- 前回までの復習
- 動作接続･遷移
  - 動作接続･遷移の原理
  - サンプルプログラム
  - 動作接続のプログラミング
  - 動作遷移のプログラミング
  - 動作接続･遷移の拡張
  - 動作接続･遷移の応用
- 動作変形



動作接続・遷移（続き）

動作接続･遷移

- 動作接続･遷移の原理
- サンプルプログラム
- 動作接続のプログラミング
- 動作接続･遷移のプログラミング
- 動作接続･遷移の拡張
- 動作接続･遷移の応用

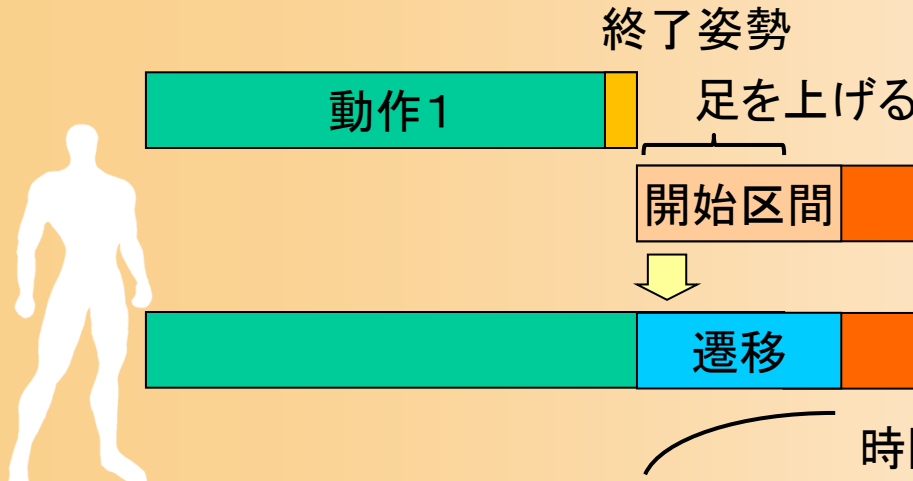
動作接続・遷移の改良・拡張

- ここまでは、基本的な動作接続・遷移のみ
- 実際には、様々な改良・拡張が必要となる

- 動作遷移のタイミングの自動決定
  - 姿勢の類似度の評価
  - 動作の解析
- 動作遷移時の姿勢補正

別の動作遷移のタイミングの例

- 動作遷移の区間の決定や姿勢補間の方法にはさまざまなやり方がある
  - 前の動作の終了姿勢を次の動作の開始部分とブレンドする方法もある
  - 動作開始時のブレンド区間のみを使用

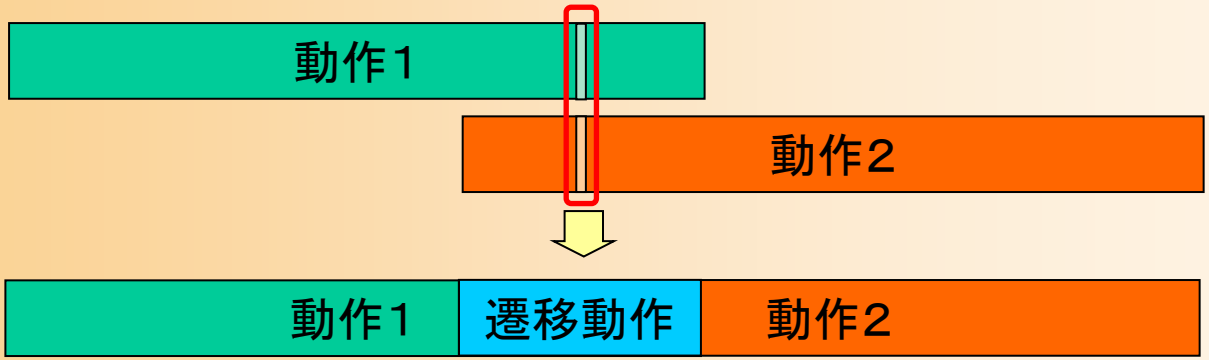


動作遷移中の姿勢ブレンド

$$\mathbf{p}(t) = w(t)\,\mathbf{P}_{next}(t_{next}) + \big(1 - w(t)\big)\,\mathbf{P}_{prev\_end}$$

動作遷移のタイミングの自動決定

- 適切なタイミングを自動的に決定する方法
  - 2つの動作中の最も近い姿勢同士を合わせる
    - 前の動作と次の動作の中から、最も近い姿勢の組を見つけ出す必要がある
  - 足が地面に着く・離れるタイミングを合わせる
    - 各動作の足の離着のタイミングを解析する必要がある



姿勢間の類似度の評価(1)

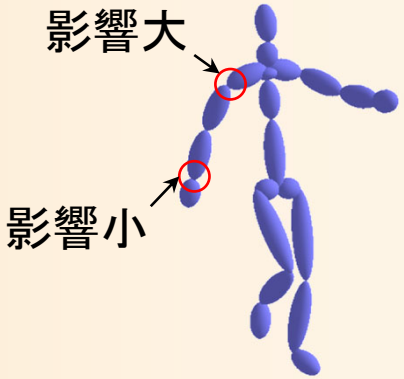
- 姿勢表現にもとづいて評価することは難しい
  - 全関節角度の差の平均値による評価

$$D = \frac{1}{n}\sum_{i}\left|\theta_i^1 - \theta_i^2\right|$$

n　関節数

θ_i^1　θ_i^2　各姿勢のｉ番目の関節の回転角度

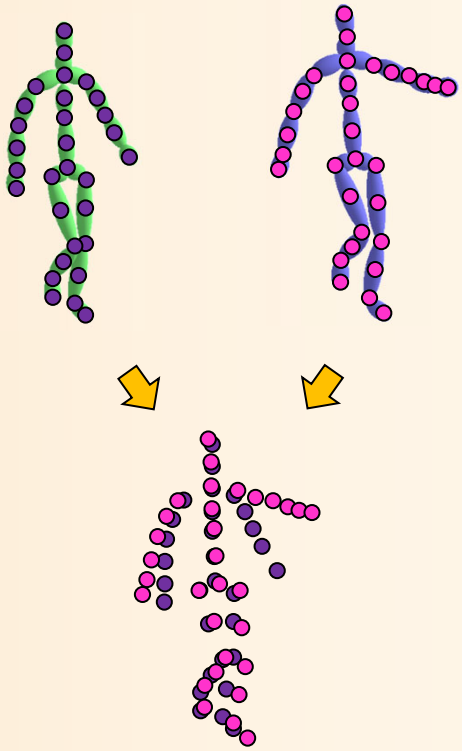
  - 関節によって、姿勢の見た目に与える影響には違いがあるため、このような計算方法は不適切
    - 関節の重み付けを行うことも難しい



- 姿勢を点群で表現して、その点同士の平均距離によって評価する方法が用いられる

姿勢間の類似度の評価(2)

1. 姿勢を点群で表現
   - 関節・体節位置
   - 形状変形モデルの頂点、など
2. 位置・方向が一致するように
   一方の点群を移動・回転
   - 2次元平面上での正規分布を求めて、
     分散が一致するように移動・回転
3. 点同士の距離の平均を計算



$$D = \frac{1}{m}\sum_{i}\left|\mathbf{p}_i^1 - \mathbf{p}_i^2\right|$$

m 点の数

$\mathbf{p}_i^1$ $\mathbf{p}_i^2$ 各姿勢の i 番目の点の座標

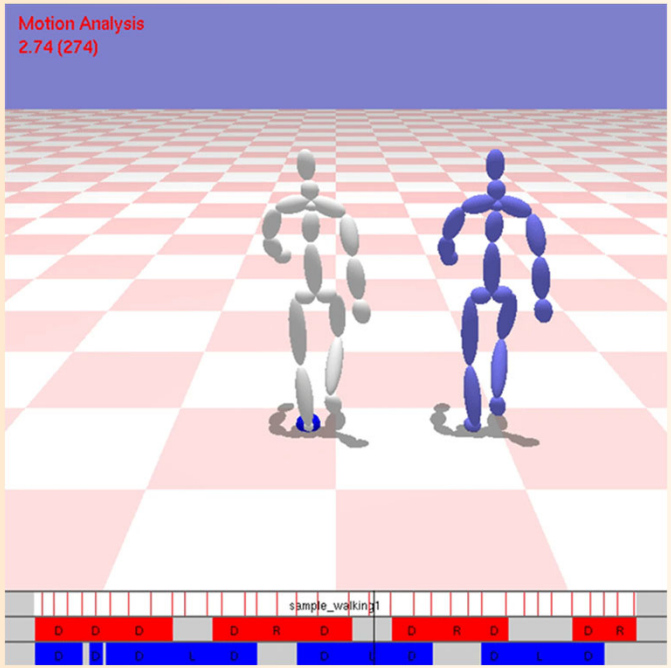
Lucas Kovar, Michael Gleicher, Frédéric H. Pighin, **Motion Graphs**, *ACM Transactions on Graphics (SIGGRAPH 2002)*, Vol. 21, Issue 3, pp. 473-482, *2002*.

動作の解析

- 動作中の重要なタイミングを検出
  - 動作が止まるタイミング
    - 末端部位の速度・加速度が0になる点、など
      - 関節によって動作のタイミングは異なるため、末端部位などの重要な部位のみに注目するか、何らかの計算方法で全身の動作を考慮する、などの対応が必要になる
  - 足が地面に着く・離れるタイミング
    - 右足・左足の速度・加速度が0で、かつ、足(踵やつま先)が地面の上にある(一定値よりも低い高さにあるタイミング)ときに、地面に着いていると判定

デモプログラム

- 動作解析アプリケーション
  - 右足・左足が地面に着く・離れるタイミングを検出
    - タイムライン上に可視化（左足：赤、右足：青）
  - キー時刻を検出して、キーフレーム動作を生成
    - タイムライン上に、検出したキー時刻を可視化
    - 入力BVH動作（青）、生成キーフレーム動作（白）を再生




Motion Analysis
0.04 (4)
動作解析
Motion Analysis
sample_walking1
D D D D R D D R D D R
D D D L D D L D D L D

動作解析のプログラム(1)

- 解析結果を格納するためのデータ構造

```
// 両足の地面との接触状態
enum SupportStateEnum
{
    NO_SUPPORT = 0 , // 両足とも地面に接触していない
    RIGHT_FOOT_SUPPORT = 1 , // 右足が地面に接触している
    LEFT_FOOT_SUPPORT = 2 , // 左足が地面に接触している
    DOUBLE_SUPPORT = 3 , // 両足が地面に接触している
    NUM_SUPPORT_STATES = 4
} ;

// 動作における地面との接触状態のキーフレーム情報
struct  MotionSupportKey
{
    // 時刻
    float  time ;
    // 両足の地面との接触状態
    SupportStateEnum  state;
} ;
```

動作解析のプログラム(2)

- 動作解析
  - 動作データの各フレームに対して
    - 各部位の位置を計算(姿勢から順運動学計算)
    - 各部位の速度を計算(前後の位置から計算)
    - 右足・左足が地面に着いているかを判定
      - 速度が閾値以下で、高さが閾値以下
    - キーフレームかどうかを判定
      - 全部位の速度の和が閾値以下で極小値になる

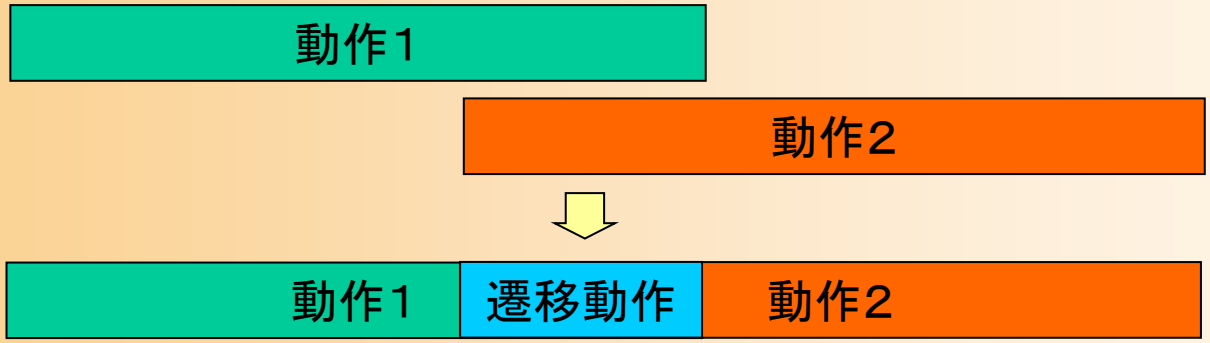

```
// 動作解析を適用、キー時刻・地面との接触状態の変化を抽出
void  MotionAnalysisForSupportStates( const Motion & motion,
    const int segment_no[ NUM_PRIMARY_BODY_PARTS ],
    vector< float > & keytimes, vector< MotionSupportKey > & support_keys )
```

動作遷移時の姿勢補正(1)

- 前後の動作の姿勢を補間するだけでは、動作遷移中の動作が不自然になる場合がある
  - 前後の動作で地面に着いている足の位置が異なると、動作遷移中に足が地面の上を滑るような動作になってしまう
- 逆運動学計算により足の位置を補正することで、不自然な動作になることを防ぐ

動作遷移時の姿勢補正(2)

- 逆運動学計算により足の位置を補正
  - 足が接地している間の足の位置を決定
  - 足が接地している間は、足の位置を固定
  - 足が接地する前・後の一定時間は、足の位置が動作遷移中の位置になるように、姿勢を変形

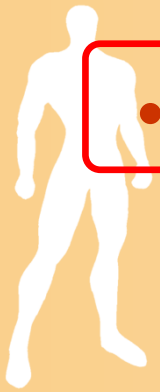


動作接続･遷移の利用時の注意

- 自然な動作接続･遷移の実現のためには、適切な遷移区間や姿勢修正方法を設定しておく必要がある
  - 前後の動作の組み合わせごとに適切な設定が必要
- 姿勢が大きく異なる動作間に適用することは難しい

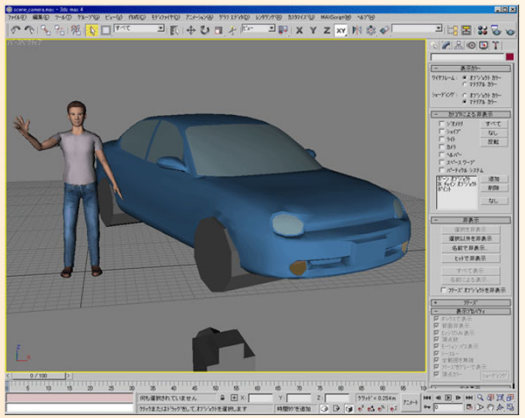
動作接続･遷移

- 動作接続･遷移の原理
- サンプルプログラム
- 動作接続のプログラミング
- 動作接続･遷移のプログラミング
- 動作接続･遷移の拡張
- 動作接続･遷移の応用



動作接続・遷移の応用(1)

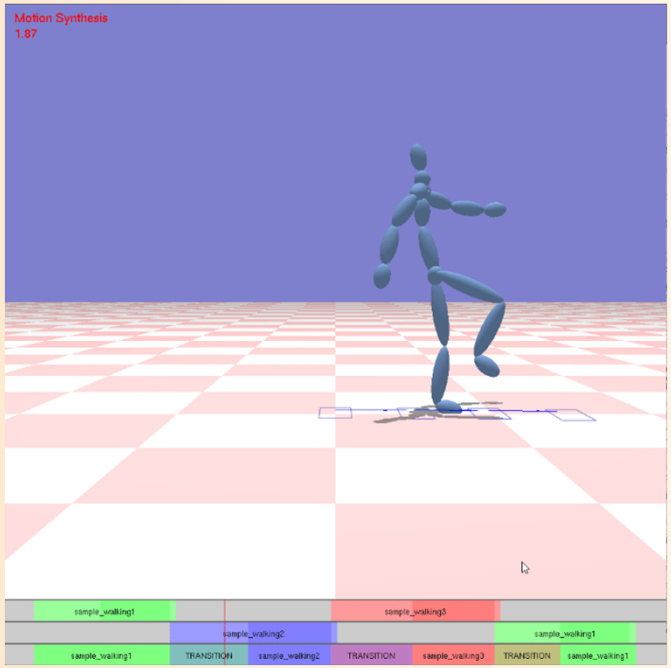
- 動作合成への応用
  - 多くのアニメーション制作システムでは、時間軸上に動作データを配置して、前後の動作間で接続・遷移を行うことで、動作合成を行える



    - 動画編集や音楽編集と同様のインターフェース
    - 複数の動作データを組み合わせてアニメーションを制作できる
  - 自然な動作を生成するためには、動作接続・遷移の方法や姿勢を手動で調整する必要がある

デモプログラム

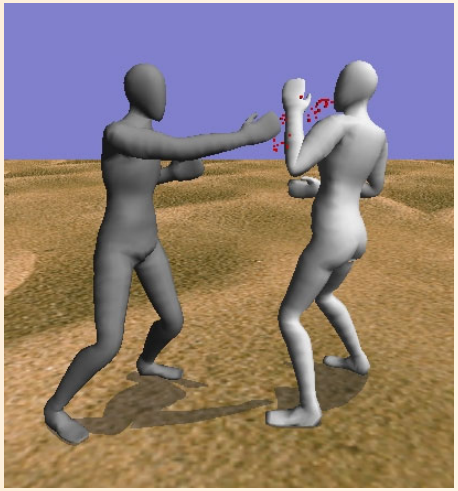
- 動作合成アプリケーション
  - 動作接続・遷移の応用により、複数の動作データをつなげて、連続的な動作を合成する
  - 動作データを時間軸上に配置
    - 数字キーを押すことで、対応する動作データを追加
  - 動作データを時間軸上でドラッグすることで、時間を変更




Motion Synthesis
0.05
動作合成
State Synthesis
sample_walking1
sample_walking2
sample_walking1
TRANSITION
sample_walking2

動作接続・遷移の応用(2)

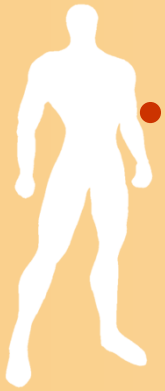
- オンライン・アニメーションへの応用
  - コンピュータゲームなどの用途で、キャラクタの連続的な動作を生成するときにも、動作接続・遷移が用いられる
    - あらかじめ用意された短い動作をつなげて再生することで、連続的な動作を実現する



  - 動作接続・遷移の情報を含む、動作状態機械のような仕組みと合わせて用いられる
    - 詳しくは後日の講義で説明

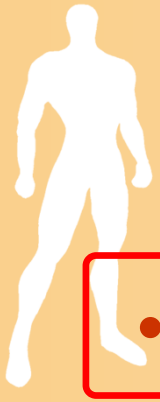
動作接続･遷移

- 動作接続･遷移の原理
- サンプルプログラム
- 動作接続のプログラミング
- 動作接続･遷移のプログラミング
- 動作接続･遷移の拡張
- 動作接続･遷移の応用



今日の内容

- 前回までの復習
- 動作接続･遷移
  - 動作接続･遷移の原理
  - サンプルプログラム
  - 動作接続のプログラミング
  - 動作遷移のプログラミング
  - 動作接続･遷移の拡張
  - 動作接続･遷移の応用
- 動作変形



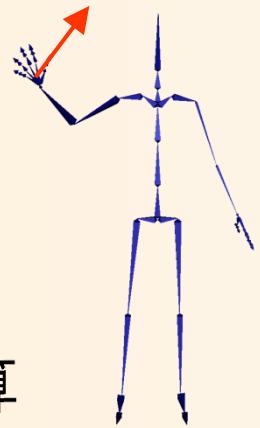
動作変形

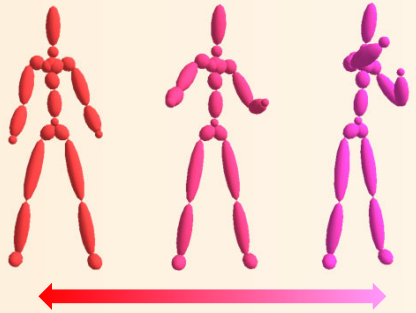
動作データの変形

- 既存の動作データを利用して、新しい動作を生成する技術
  - 動作遷移・接続、動作補間（これまでの授業で説明）
  - タイム・ワーピング
  - モーション・ワーピング
  - モーション・リターゲッティング
- 市販のアニメーション制作ソフトウェアでは、これらの機能を使うことで動作データの変形を行える
  - オンラインアニメーションでも、動作データを動的に変形するために、これらの技術が用いられることがある

動作変形に用いる基礎技術

- 順運動学
  - 任意の関節の回転を直接変更
- 逆運動学
  - 任意の体節・関節の位置・向きを変更
    - 条件を満たすように複数関節の回転を計算
- 姿勢補間
  - 2つ、または、複数の姿勢を補間
    - 複数関節の回転を計算





※ これらの応用・組合せにより動作変形を実現

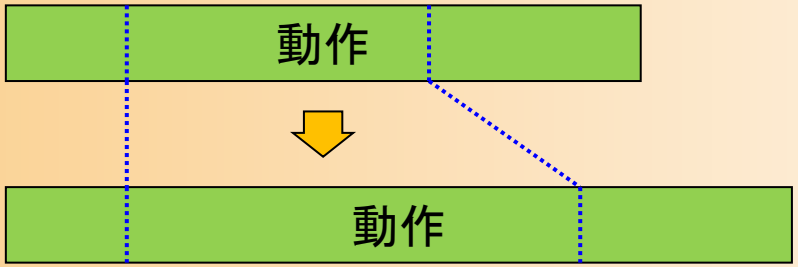
姿勢・動作に関する制約

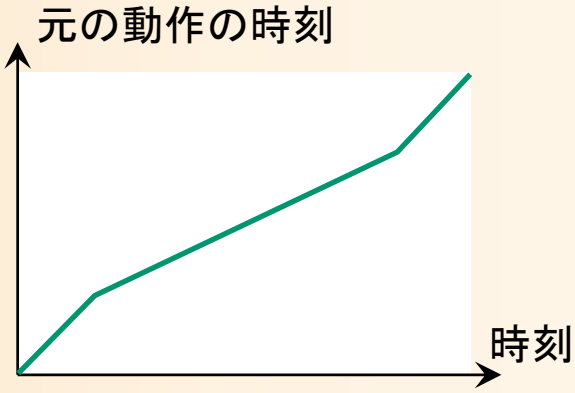
- 姿勢に関する制約
  - 関節の回転範囲
  - 自己衝突の回避
  - 静的なバランスの保持
- 動作に関する制約
  - 支点（足）を地面に固定
  - 前後の姿勢の連続性
  - 動的なバランスの保持

- これらの制約を守るように姿勢や動作を生成・変形する必要がある

タイム・ワーピング

- 動作データの再生時間を変化させることで、動作の速度を変更する
  - タイムワープ関数により時間変化を指定
    - 例：動作の一部分の時間を変化





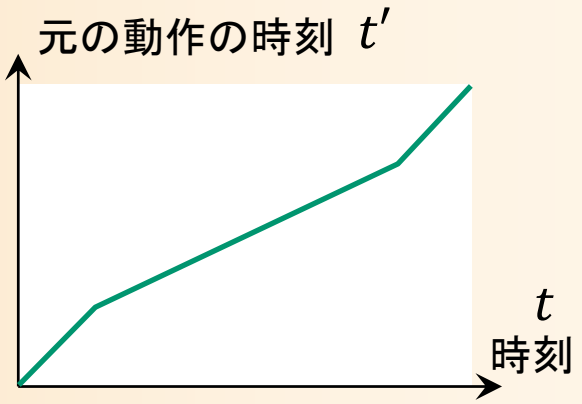
  - 再生速度が大幅に変化したり、再生速度が急激に変化したりすると、不自然な動作になるので、タイムワープ関数を滑らかにするなど工夫が必要

タイム・ワーピングの実現方法

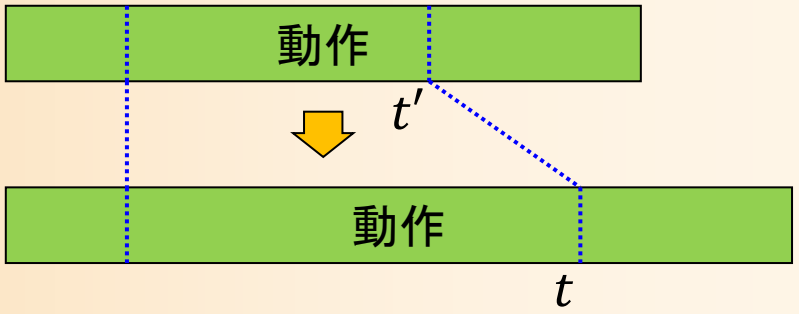
- 動作データの再生時間を変化させることで、動作の速度を変更する
  - タイムワープ関数を定義
    - 必ず単調増加する関数とする

      $t' = f(t)$

      $t = f^{-1}(t')$　※ 逆関数も存在

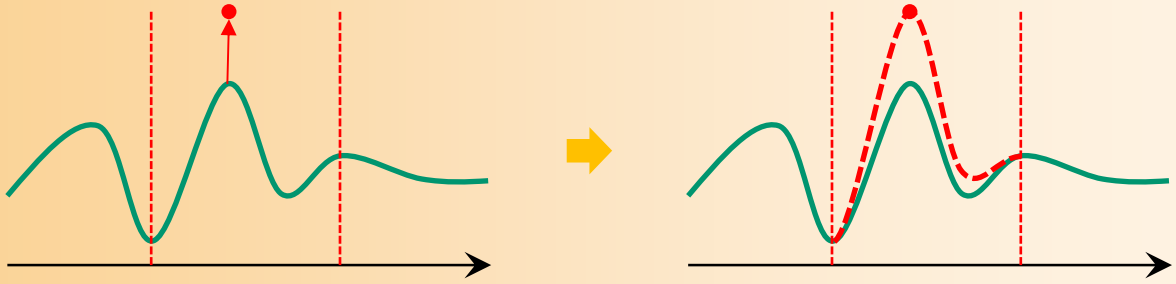


  - 動作再生
    - 時刻 t において、元の動作の時刻 t' の姿勢を表示



モーション・ワーピング

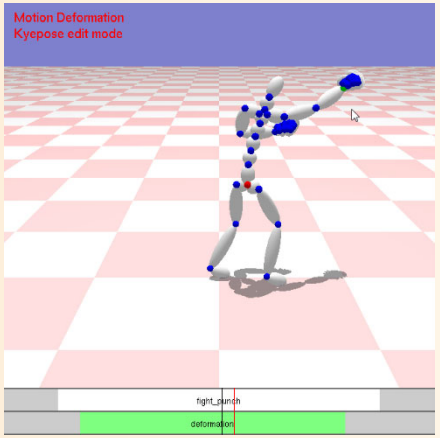
- 動作データ中のキーフレームの姿勢変化に合わせて前後の動作を滑らかに変化させる
  - キーフレームの時刻、変形後の姿勢、変形を適用する区間の開始・終了時刻を指定
  - 区間中の各時刻の姿勢に、姿勢ブレンドを適用
    - 時刻に応じてブレンド比率を変化（ブレンド比率関数）



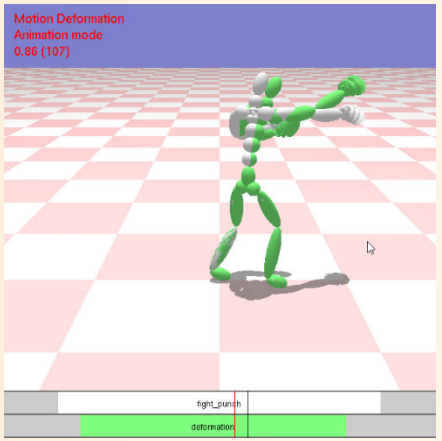
  - 姿勢変化が大きいと、不自然になる可能性がある
    - 変形の時間区間やブレンド比率関数の設定が重要

デモプログラム

- モーション・ワーピング
  - パンチ動作を元の動作として読み込み、キー時刻を設定
  - キー姿勢の変形モード
    - 逆運動学計算（CCD法）を使用
  - 変形動作の再生モード
    - 動作中の各フレームの姿勢を変形
  - 動作変形の時間範囲の変更
    - タイムライン上で時間を指定



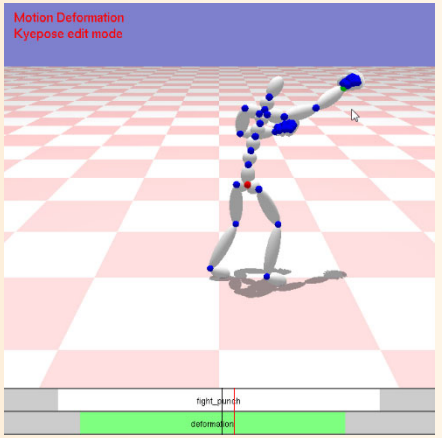
キー姿勢の変形モード



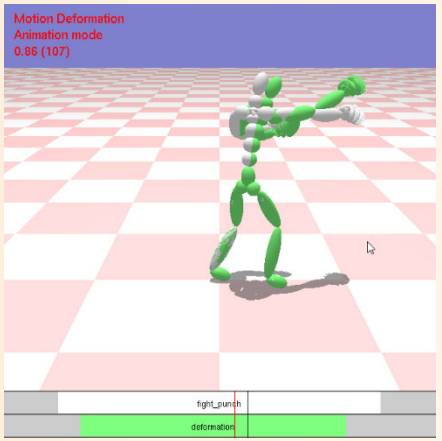
変形動作の再生モード

デモプログラム

- 操作方法
  - スペースキーで、変形動作の再生モードと、キー姿勢の変形モードを切替
  - キー姿勢の変形モードでは、任意の関節をドラッグすることで、姿勢を変形
    - 詳細は逆運動学計算（CCD法）のデモプログラムの操作方法の説明を参照
  - タイムライン上でクリックすることで、動作変形を適用する範囲の開始・終了時間を設定
  - 変形動作の再生中に、Dキーで、変形前の動作（白）と変形後の動作（緑）の描画の切替



キー姿勢の変形モード

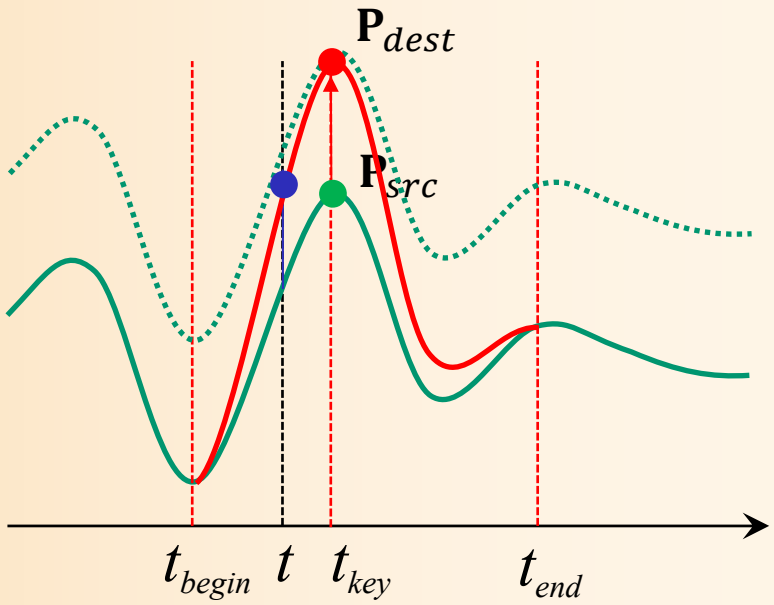


変形動作の再生モード


Motion Deformation
Animation mode
0.03 (3)
動作変形
Motion Deformation
fight_punch
deformation

モーション・ワーピングの方法

- 動作中の姿勢の変形
  - キー時刻（t_{key}）において、変形前の姿勢（$\mathbf{P}_{src}$）が変形後の姿勢（$\mathbf{P}_{dest}$）になるように、各時刻の姿勢を変形
    - 時間 t に応じて、変形の重み $w(t)$ を計算
    - 重み $w(t)$ に応じて、元の動作（緑実線）と、元の動作を一定量変化させた動作（緑点線）を補間（赤点線）



Andrew Witkin, and Zoran Popovic,
Motion Warping, SIGGRAPH 95, 1995.

モーション・ワーピングの方法

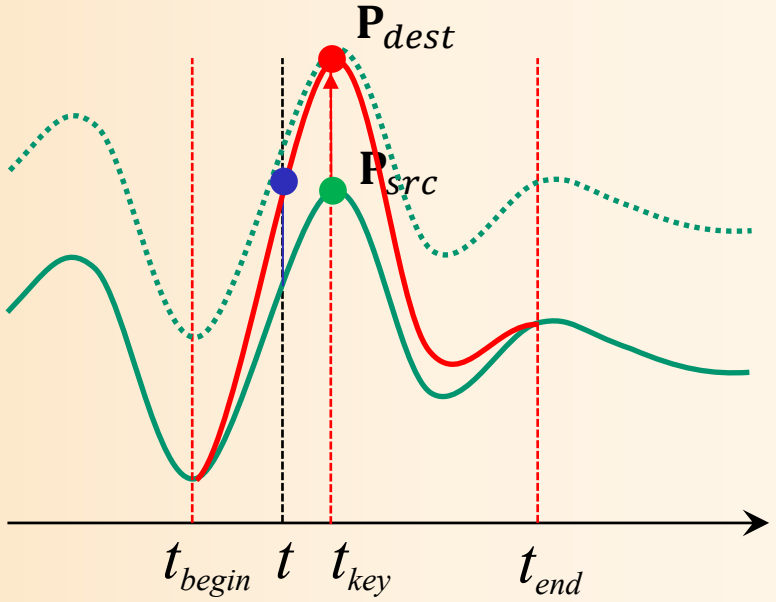
- 動作中の姿勢の変形
  - キー時刻(t_{key})において、変形前の姿勢($\mathbf{P}_{src}$)が変形後の姿勢($\mathbf{P}_{dest}$)になるように、各時刻の姿勢を変形

$$\mathbf{P}(t)' = w(t)(\mathbf{P}_{dest} - \mathbf{P}_{src} + \mathbf{P}(t)) + \big(1 - w(t)\big)\mathbf{P}(t)$$

または

$$\mathbf{P}(t)' = w(t)(\mathbf{P}_{dest} - \mathbf{P}_{src}) + \mathbf{P}(t)$$

※ 関節回転の表現方法に応じて計算
（例：回転行列による表現であれば、
差・和は（逆）行列の積により計算）



モーション・ワーピングの方法

- 動作中の姿勢の変形
  - 関節の回転が回転行列により表されている場合は、行列の積により計算

$$\mathbf{R}(t)' = w(t)\left(\mathbf{R}_{dest}\ {\mathbf{R}_{src}}^{t}\ \mathbf{R}(t)\right) + \left(1 - w(t)\right)\mathbf{R}(t)$$

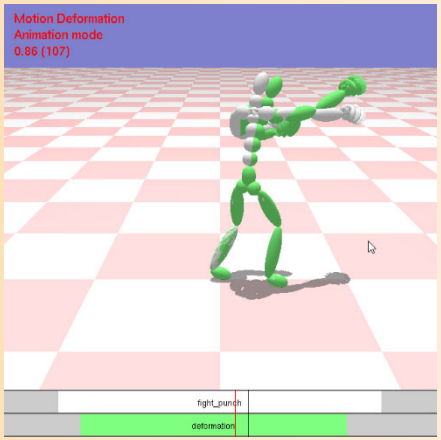
または

$$\mathbf{R}(t)' = w(t)\left(\mathbf{R}_{dest}\ {\mathbf{R}_{src}}^{t}\right)\mathbf{R}(t)$$

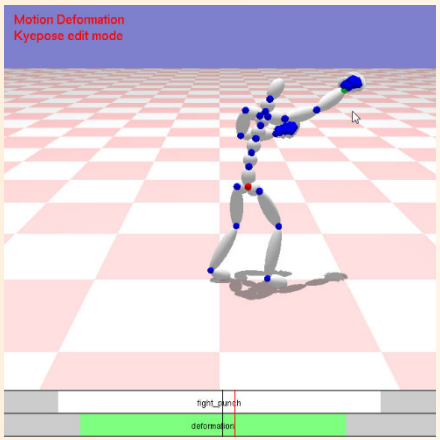
※ 回転行列の逆行列は、転置行列と同じになる
※ 1つ目の式を用いる場合は、四元数を使った姿勢補間が利用可能

プログラミング演習

- 動作変形アプリケーション
  - 動作変形情報(キー時刻・姿勢)にもとづいて、動作を変形しながら再生
  - 動作変形処理(各自実装)
  - キー時刻・姿勢の操作機能
    - 逆運動学計算(CCD法)の処理を呼び出し



変形動作の再生モード



キー姿勢の変形モード

動作変形アプリケーション

- MotionDeformationApp（一部未実装）
  - 基底クラス（InverseKinematicsCCDApp）を継承
    - 逆運動学計算（CCD法）によるキー姿勢編集のため
  - 動作変形の初期化
  - 動作変形処理（各自実装）
- MotionDeformationEditApp
  - 基底クラス（MotionDeformationApp）を継承
    - 動作変形情報（キー時刻・姿勢）の編集機能を追加
  - こちらのアプリケーションが実行される

モーション・ワーピングのプログラム(1)

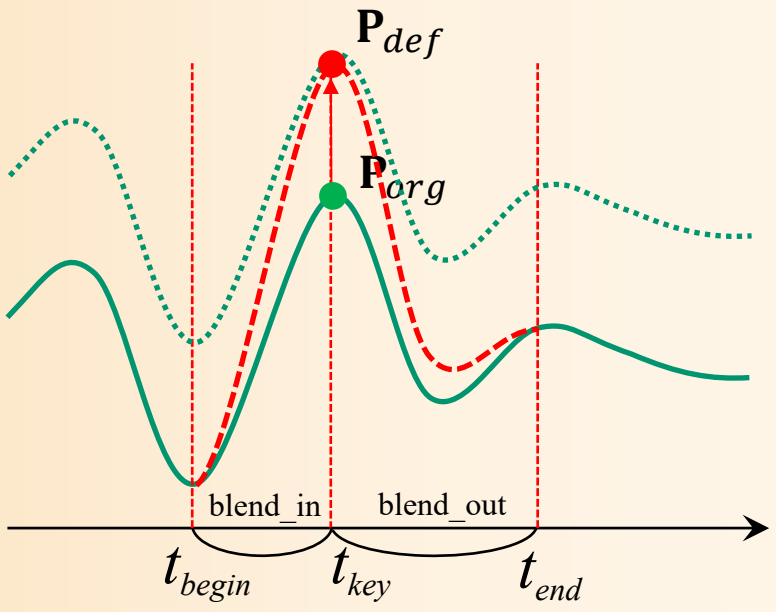
- 動作変形のための情報の定義

```
// 動作変形(動作ワーピング)の情報
struct  MotionWarpingParam
{
    // 姿勢変形を適用するキー時刻
    float       key_time;

    // 変形前のキー時刻の姿勢
    Posture     org_pose;

    // 変形後のキー時刻の姿勢
    Posture     key_pose;

    // 姿勢変形の前後のブレンド時間
    float       blend_in_duration;
    float       blend_out_duration;
};
```



モーション・ワーピングのプログラム(2)

- 動作変形情報の初期化
  - プログラムの初期化処理から呼び出し
  - 元のBVH動作ファイル名や時間情報を関数内に直接記述
    - 変形後の姿勢を含むBVH動作を読み込み
    - もしくは、逆運動学計算により変形後の姿勢を作成


```
// 入力動作・動作変形情報の初期化
void  MotionDeformationApp::InitMotion( int no )
{
    LoadBVH( "fight_punch.bvh" );
    BVH *  key_bvh = new BVH( "fight_punch_key.bvh" );
    Motion *  key_pose_motion = CoustructBVHMotion( key_bvh );
    InitDeformationParameter( *motion, 0.80f, 0.70f, 0.50f, deformation );
    deformation.key_pose = *key_pose_motion->GetFrame( 0 );
```

モーション・ワーピングのプログラム(3)

- 動作変形処理(1)
  - 時刻に応じて重みを計算し、動作変形を適用
    - 重みは、キー時刻で 1 となるように計算
      - 変形区間の開始時刻からキー時刻の間で単調増加
      - キー時刻から変形区間の終了時刻の間で単調増加
      - 具体的な関数は任意(線形補間など)

```
//  動作変形(動作ワーピング)の適用後の姿勢の計算
float  ApplyMotionDeformation( float time, const MotionWarpingParam & deform,
    const Posture & input_pose, Posture & output_pose )
{
    // 動作変形(動作ワーピング)の重みを計算
    float  ratio = ???;

    // 姿勢変形
    PostureWarping( input_pose, deform.org_pose, deform.key_pose, ratio, output_pose );
}
```

モーション・ワーピングのプログラム(4)

- 動作変形処理(2)
  - 姿勢変形処理

```
//  動作ワーピングの姿勢変形(2つの姿勢の差分(dest - src)に重み ratio をかけたものを
//  元の姿勢 org に加える )
void  PostureWarping( const Posture & org, const Posture & src, const Posture & dest,
    float ratio, Posture & p )
{
    // 各関節の回転を計算
    for ( int i = 0; i < body->num_joints; i++ )
    {
        p.joint_rotations[ i ] = ???;
    }
    // ルートの向きを計算
    p.root_ori = ???;
    // ルートの位置を計算
    p.root_pos = ???;
}
```

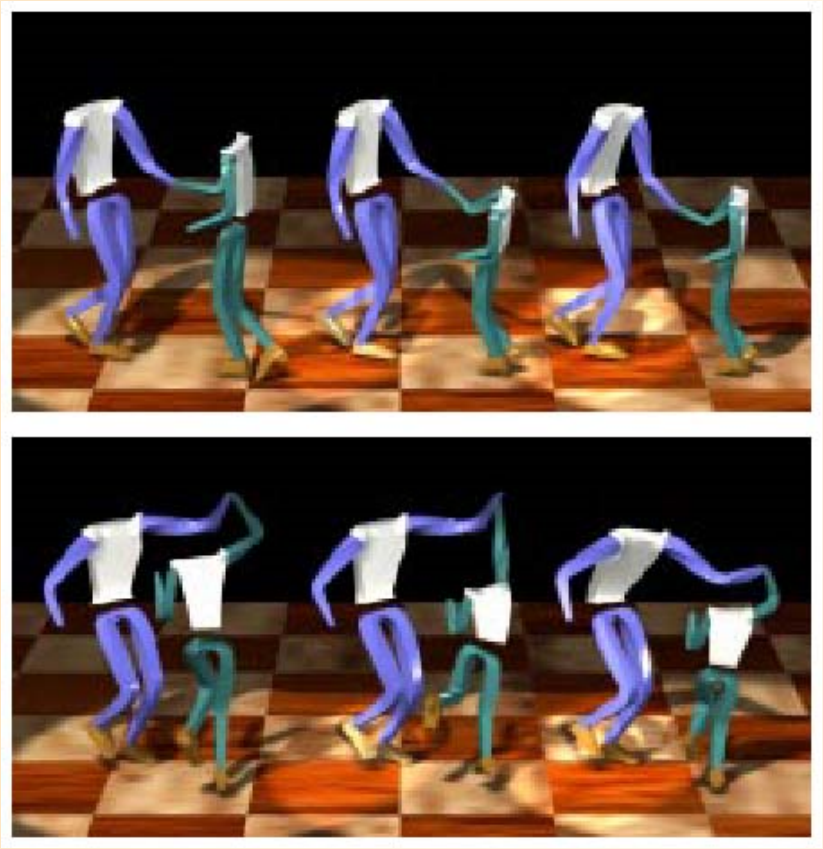
$$\mathbf{R}(t)' = w(t)\left(\mathbf{R}_{def}\ {\mathbf{R}_{org}}^{t}\ \mathbf{R}(t)\right) + \left(1 - w(t)\right)\mathbf{R}(t)$$

または

$$\mathbf{R}(t)' = w(t)\left(\mathbf{R}_{def} - \mathbf{R}_{org}\right) + \mathbf{R}(t)$$

モーション・リターゲッティング（1）

- 骨格の異なるキャラクタ間で動作データを変換
  - 例：モーションキャプチャデータを、元の俳優の骨格から、実際のキャラクタの骨格用に修正
  - 単純に関節角度をコピーするだけでは、足の位置がずれたりするため修正が必要



[Gleicher 98]

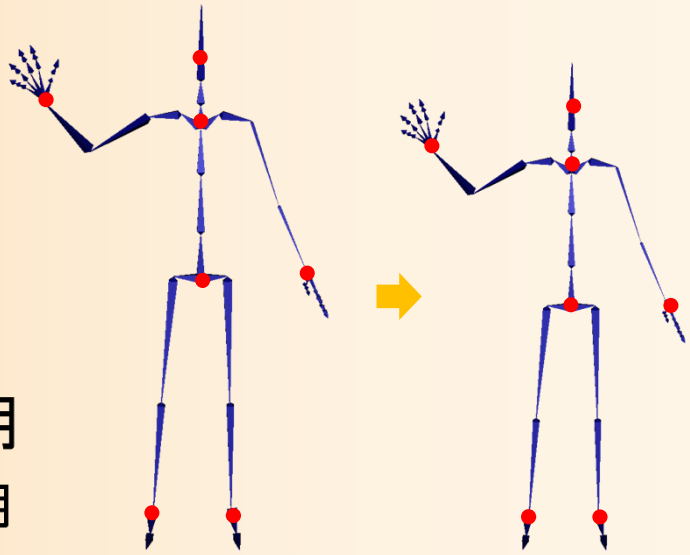
Michael Gleicher, **Retargetting Motion to New Characters**, SIGGRAPH 98, pp. 33-42, 1998.

モーション・リターゲッティング（2）

- モーション・リターゲッティングの計算方法
- 動作最適化問題として解くのが一般的
  - 下記の項目を含む、目標関数 f（動作）を定義
    - 変形後の各姿勢が元の姿勢の制約条件を満たす（足が地面に接しているときに固定したり、キャラクタ同士が接触しているときに接触を保つなど）
    - 変形後の各姿勢が元の姿勢になるべく近くなる
    - 動作中の前後の姿勢が連続的に変化するようにする
  - 目標関数 f（動作）がなるべく小さくなるように、動作全体を最適化（変形）

モーション・リターゲッティング（3）

- 簡易的なモーション・リターゲッティング
  - 部位の位置にもとづくリターゲッティング
  - 変換後の姿勢の各部位の位置（＋向き）を計算し、逆運動学計算により姿勢を変形
    - 例：身長の比率にもとづき部位の位置をスケーリング
    - 接触時は、接触位置から部位の位置を決定
    - 前後のフレームで姿勢の連続性を保つようにIKを適用
      - 解析的な逆運動学計算を適用
      - 前フレームの姿勢を変形



レポート課題

- キャラクタ・アニメーション(2)
  - サンプルプログラムの未実装部分(前半)は作成済み
  1. 順運動学計算
  2. 姿勢補間
  3. キーフレーム動作再生
  4. 動作補間
  - サンプルプログラムの未実装部分(後半)を作成
  5. 動作変形
  6. 動作接続・遷移
  7. 逆運動学計算(CCD法)

残りの課題は
次回説明

まとめ

- 前回までの復習
- 動作接続･遷移
  - 動作接続･遷移の原理
  - サンプルプログラム
  - 動作接続のプログラミング
  - 動作遷移のプログラミング
  - 動作接続･遷移の拡張
  - 動作接続･遷移の応用
- 動作変形

次回予告

- 人体モデル（骨格・姿勢・動作）の表現
- 人体モデル・動作データの作成方法
- サンプルプログラム
- 順運動学、人体形状変形モデル
- 姿勢補間、キーフレーム動作再生、動作補間
- 動作接続・遷移、動作変形
- 逆運動学、モーションキャプチャ
- 動作生成・制御